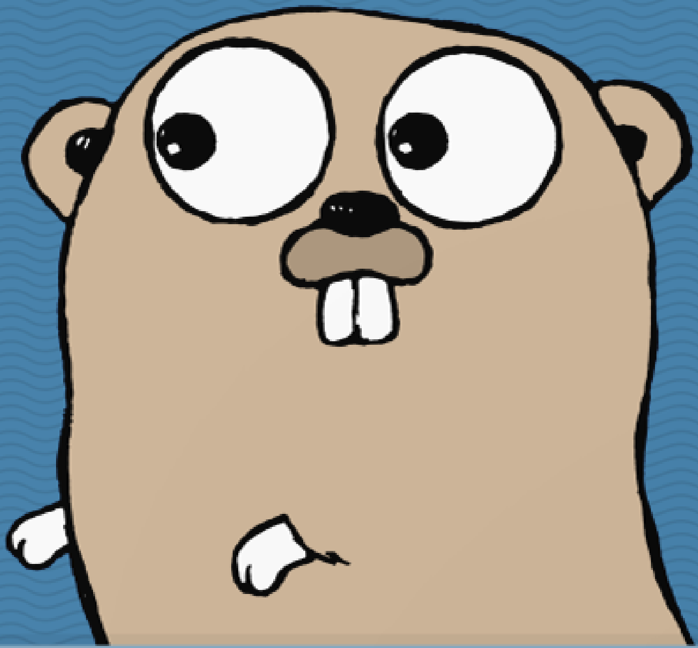
Alex Edwards

Let's Go!

Learn to build professional web applications with Go



A step-by-step guide to creating fast, secure and maintainable web applications

First 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 that 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!

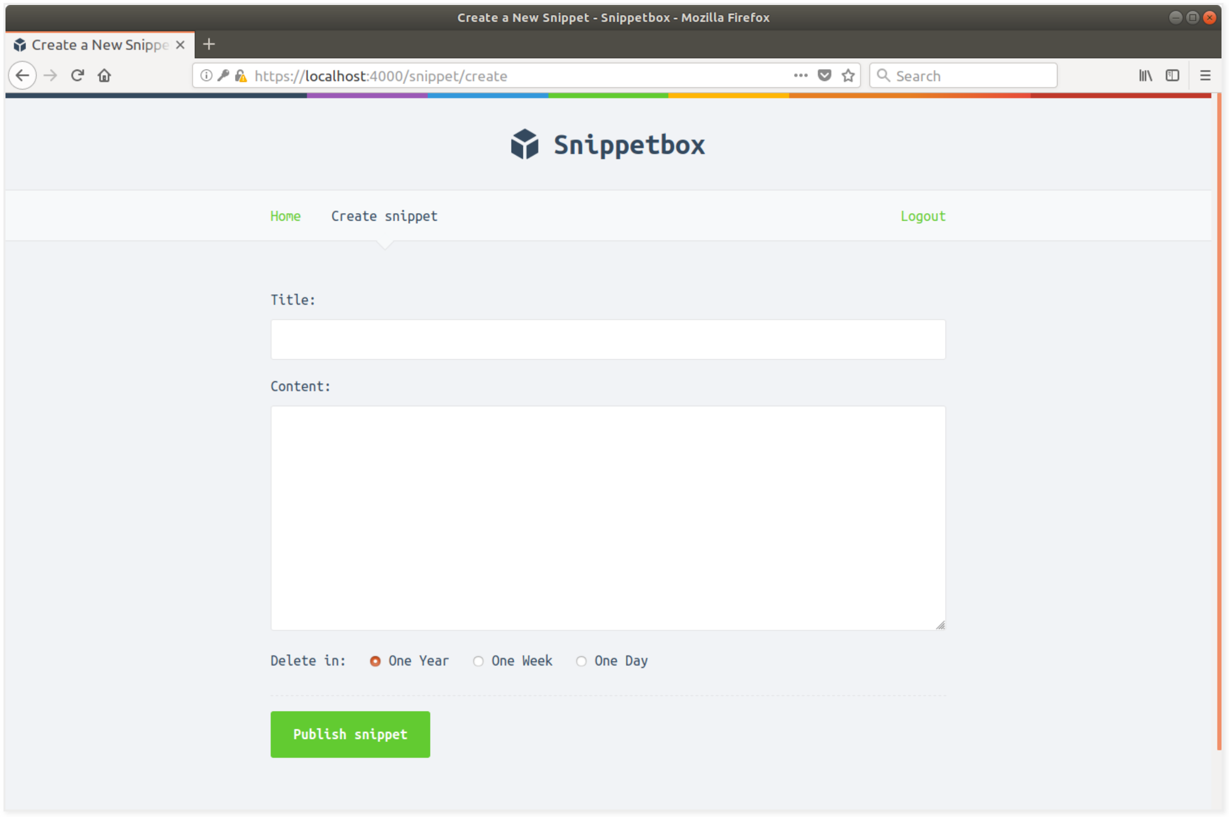
— Alex

Contents

Chapter 1.

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 is able 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.

Prerequisites

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.

In terms of software, you'll need a working installation of Go (version 1.11 or newer).

We'll also use Curl in a few places throughout the book to inspect the responses that our application sends. This should be pre-installed on most Macs, and Linux/Unix users should find in their package repositories as `curl`. Windows users can download it from here.

Conventions

Throughout this book code blocks are shown with a silver background like below. If the code is particularly long, 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 we're working on.

```
File: hello.go
```

```go
package main

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

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

Terminal (command line) instructions are show 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.

```
$ 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.

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.

About the Author

Hey, I'm Alex Edwards, a full-stack web developer. I began working with Go 6 years ago in 2013, and have been teaching people and writing about the language for nearly as long.

I've used Go to build a variety of production applications, from simple websites to high-frequency trading systems. I also maintain several open-source Go packages, including the popular session management system SCS.

I live near Innsbruck, Austria. You can follow me on GitHub, Instagram, Twitter and on my blog.

Copyright and Disclaimer

Chapter 2.

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:

- Setup a project repository 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.
- Send different HTTP responses, headers and status codes to users.
- Fetch and validate untrusted user input from URL query string parameters.
- Structure your project in a sensible and scalable way.
- Render HTML pages and use template inheritance to keep your markup DRY and free of boilerplate.
- Serve static files like images, CSS and JavaScript from your application.

Chapter 2.1.

Installing Go

If you'd like to code along with this book, you'll need Go version 1.11 installed on your computer.

As well as generally being the latest and greatest version of Go, the 1.11 release ships with support for *modules* — which are Go's new way to manage and version control any external packages used by your project.

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

```
$ go version
go version go1.11 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

Chapter 2.2.

Project Setup and Enabling Modules

Before we write any code, you'll need to first set up a project directory (also known as a *repository*) on your computer which follows the Go conventions.

This directory will act as the top-level 'home' for our Snippetbox project throughout this book — all the Go code we write will live in there, along with other project-specific assets like HTML templates and CSS files.

So where should this project directory live, and what should we call it?

Open a terminal window and run the `go env` command to get the information about your current Go installation and environment. The output should look something like this:

```
$ go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/home/alex/.cache/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/alex/go"
```

```
GOPROXY=""
GORACE=""
GOROOT="/usr/local/go"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
CC="gcc"
CXX="g++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-bu
```

The important thing to look at here is the `GOPATH` value — which in my case is the directory `/home/alex/go`.

For older versions of Go (pre-1.11) it was best practice to locate your project repositories within a `src` folder under your `GOPATH` directory.

But if you're running Go 1.11 and using modules to manage your dependencies — like we will be in this book — then this rule no longer applies. You can create your project repository anywhere on your computer, and it's actually simplest if you locate it *outside* of your `GOPATH` directory.

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

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

While we're at it, let's also add an empty `main.go` file to the project directory:

```
$ cd $HOME/code/snippetbox
$ touch main.go
```

Enabling Modules

The next thing we need to do is let Go know that we want to use the new modules functionality to help manage (and version control) any third-party packages that our project imports.

To do this we first need to decide what the *module path* for our project should be.

The important thing here is uniqueness. To avoid potential import conflicts with other people's packages or the standard library in the future, you want to choose a module path that is globally unique and unlikely to be used by anything else. In the Go community, a common convention is to namespace your module paths by basing them on a URL that you own.

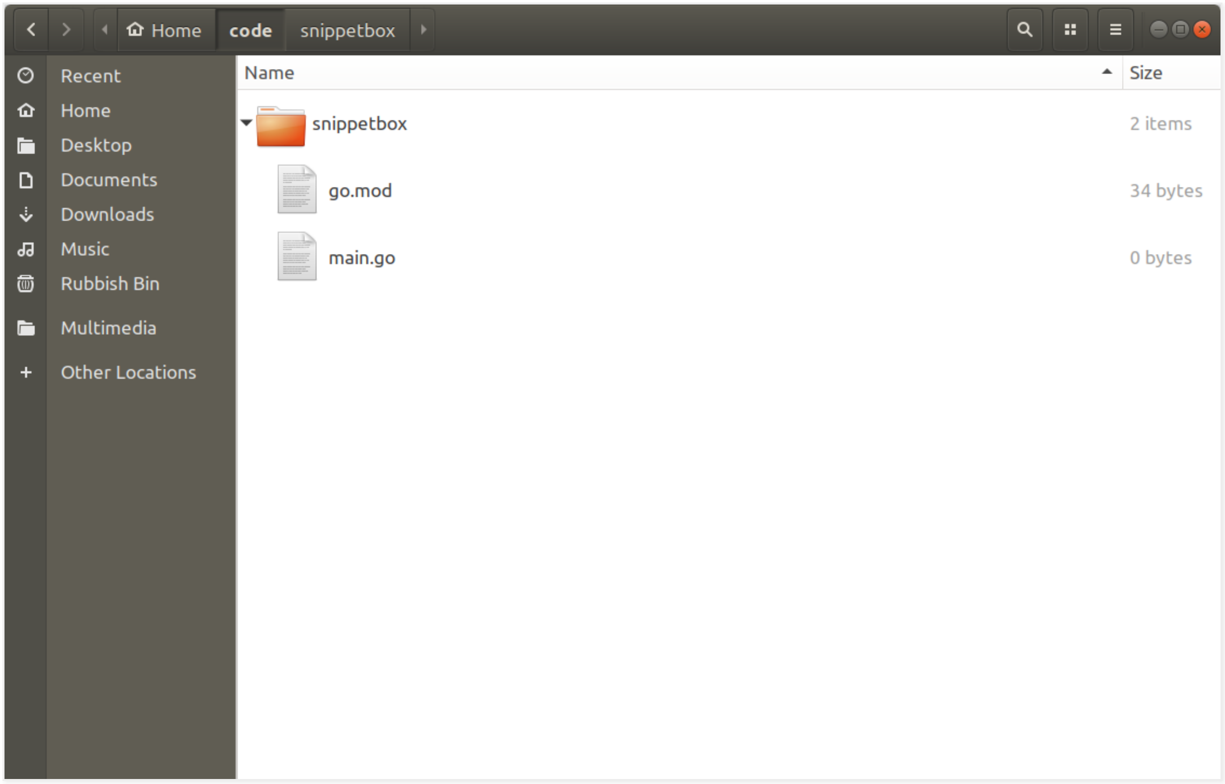
In my case a clear, succinct and *unlikely-to-be-used-by-anything-else* module path for this project would be `alexedwards.net/snippetbox`, and we'll use this throughout the rest of the book. If possible, you should swap this for something that's unique to you instead.

Now that we've decided a unique module path let's enable modules for the project.

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

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

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: mod.go
```

```
module alexedwards.net/snippetbox
```

Later in our build we'll see how this file is used to define the third-party packages (and their versions) which are *required* by our project.

Additional Information

Module Paths for Downloadable Packages

If you’re creating a package or application 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`.

Chapter 2.3.

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're coming from an MVC-background, 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 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 or Apache.

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)

    // Use the http.ListenAndServe() function to start a new web server. We pas
    // two parameters: the TCP network address to listen on (in this case ":400
    // and the servemux we just created. If http.ListenAndServe() returns an er
    // we use the log.Fatal() function to log the error message and exit.
    log.Println("Starting server on :4000")
    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

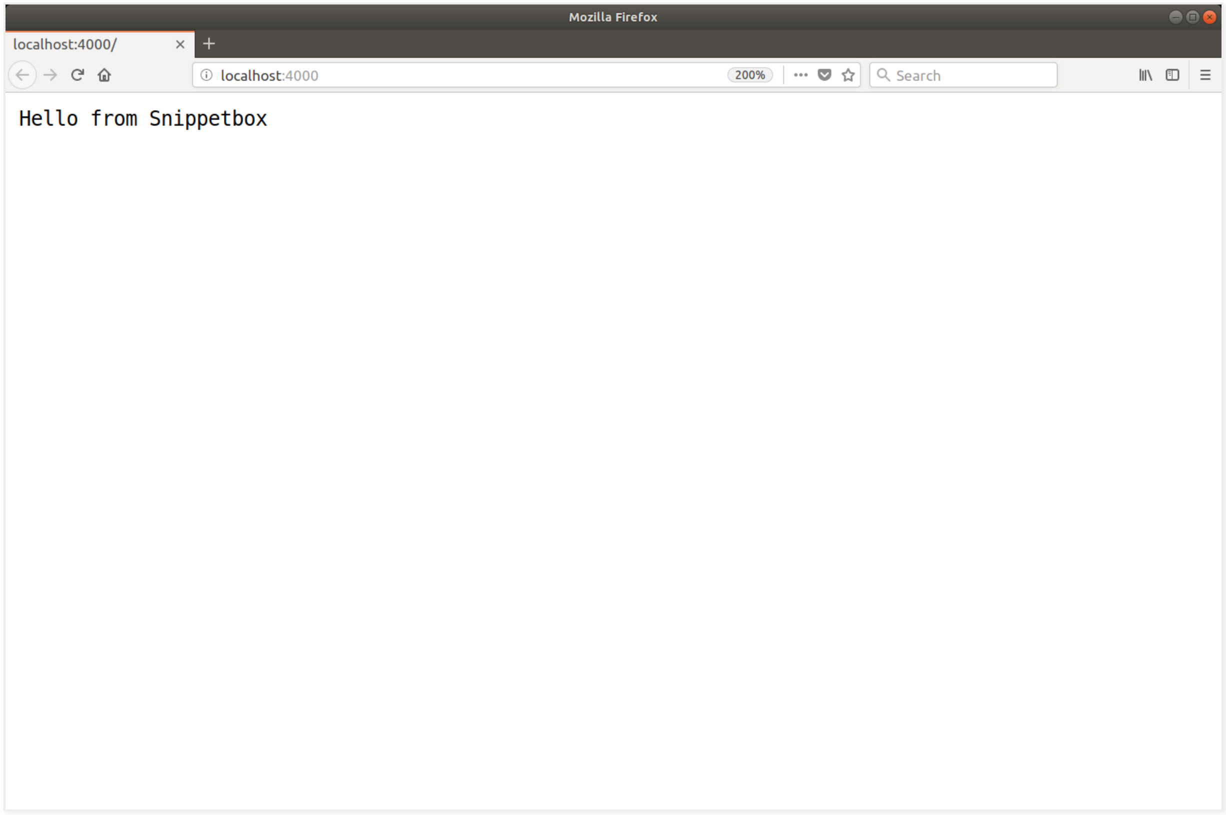
Note: Our `home` handler function is just a regular Go function which takes two parameters: the `http.ResponseWriter` provides methods for assembling a HTTP response and sending it to the user, and `http.Request` is a struct which holds information about the current request (such as the HTTP method and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

When you run this code, it should 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 main.go
2018/08/02 10:08:07 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:



```
Hello from Snippetbox
```

Important: Before we continue, I should explain that Go's servemux treats the URL pattern `"/"` like a catch-all. So at the moment *all HTTP requests* 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`, and you'll receive exactly the same response. We'll talk more about this in the next chapter.

If you head back to your terminal window, you can stop the web 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 Go will attempt to look up the relevant port number from your `/etc/services` file when starting the server, or will return an error if a match can't be found.

Logging

In the code above we're using Go's `log.Println()` and `log.Fatal()` functions to output log messages. Both these functions output messages via Go's 'standard' logger, which — by default — prefixes messages with the local date and time and writes them to the standard error stream (which should display in your terminal window).

The `log.Fatal()` function will also call `os.Exit(1)` after writing the message, causing the application to immediately exit.

Chapter 2.4.

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:

URL Pattern	Handler	Action
/	home	Display the home page
/snippet	showSnippet	Display a specific snippet
/snippet/create	createSnippet	Create a new snippet

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

File: main.go

```go
package main

import (
    "log"
    "net/http"
)

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

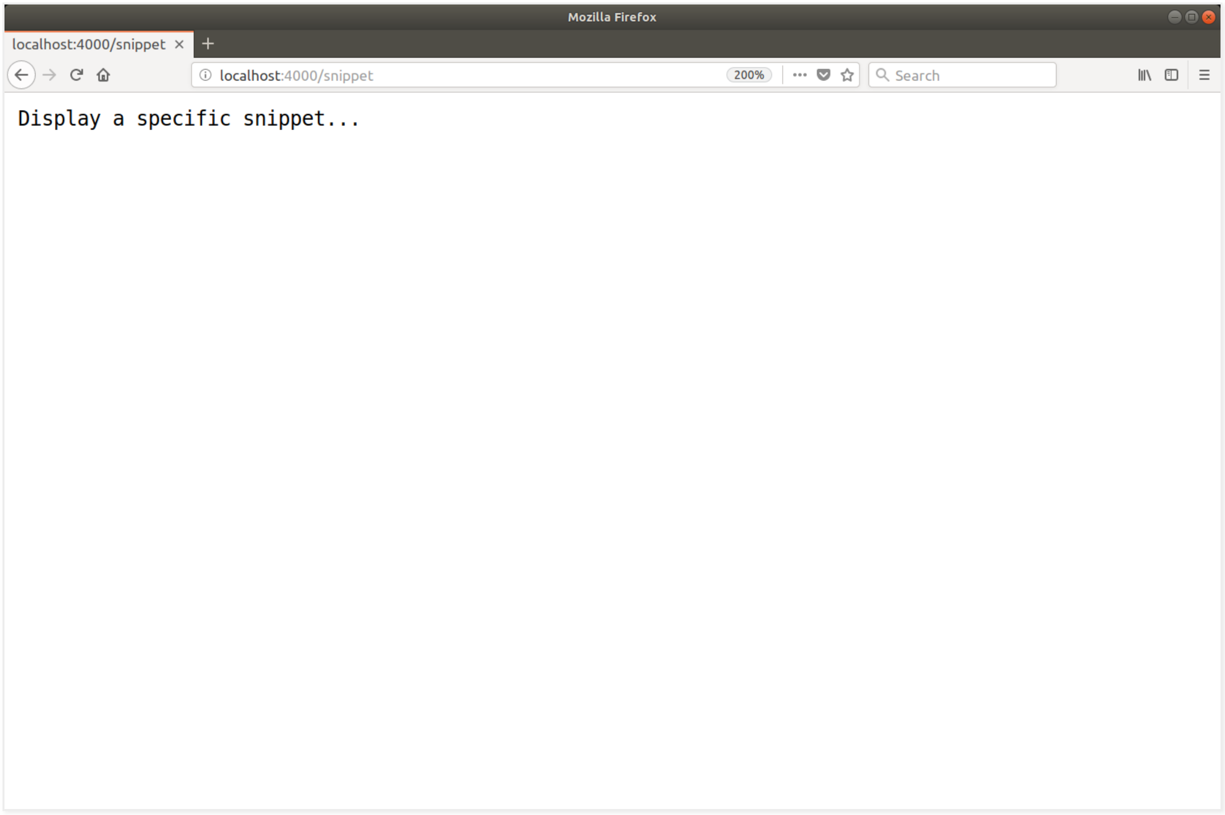
```
// Add a showSnippet handler function.
func showSnippet(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Display a specific snippet..."))
}

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

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

    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 main.go
2018/08/02 11:36:25 Starting server on :4000
```

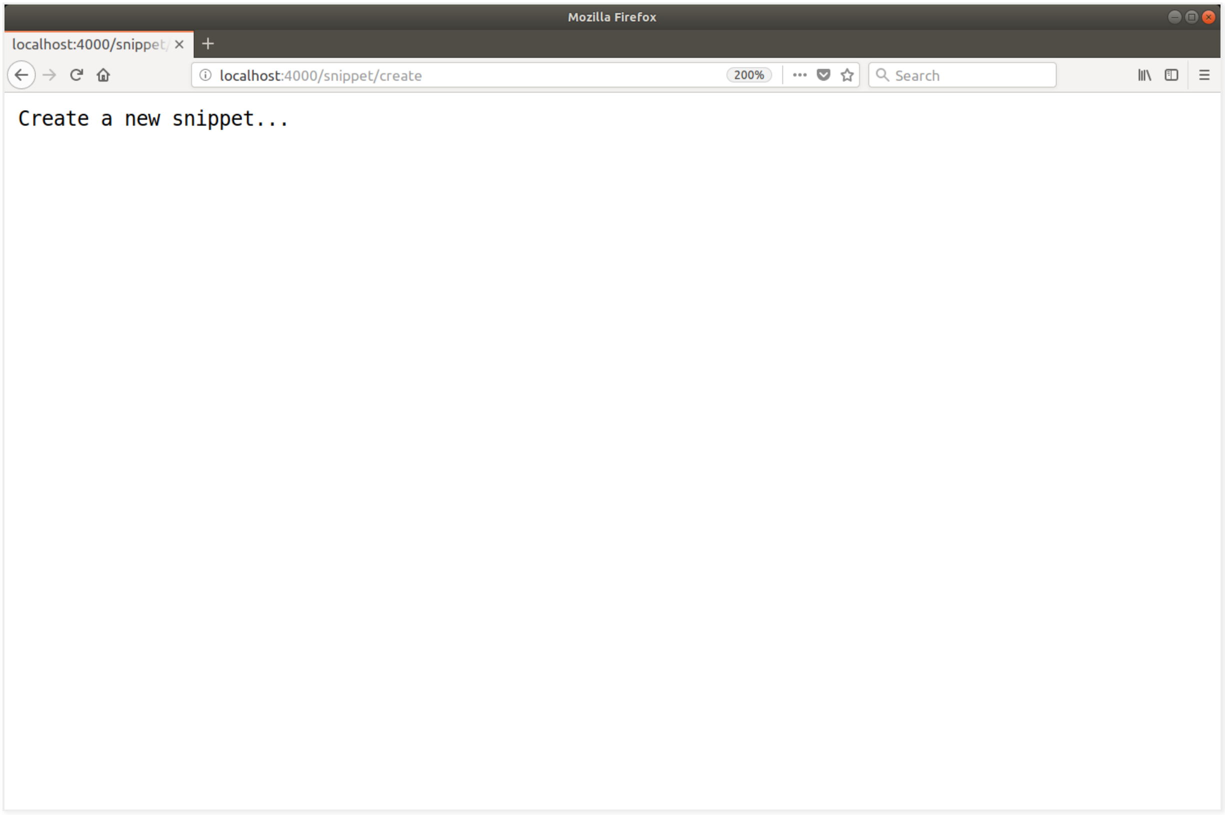
If you visit the following links in your web browser you should now get the appropriate response for each route:

- `http://localhost:4000/snippet`



```
Display a specific snippet...
```

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



```
Create a new snippet...
```

Fixed Path and Subtree Patterns

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

Go's servemux supports two different types of URL patterns: *fixed paths* and *subtree paths*. Fixed paths *don't* end with a trailing slash, whereas subtree paths *do* end with a trailing slash.

Our two new patterns — `"/snippet"` and `"/snippet/create"` — are both examples of fixed paths. In Go's servemux, fixed path patterns like these are only matched (and the corresponding handler called) when the request URL path *exactly* matches the fixed path.

In contrast, our pattern `"/"` is an example of a subtree path (because it ends in a trailing slash). Another example would be something like `"/static/"`. 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 `"/"` pattern is acting like a catch-all. The pattern essentially means *match a single slash, followed by anything (or nothing at all)*.

Restricting the Root URL Pattern

So what if you don't want the `"/"` pattern to act like a catch-all?

For instance, in the application we're building we want the home page to be displayed if — and only if — the request URL path exactly matches `"/"`. Otherwise, we want the user to receive a `404 page not found` response.

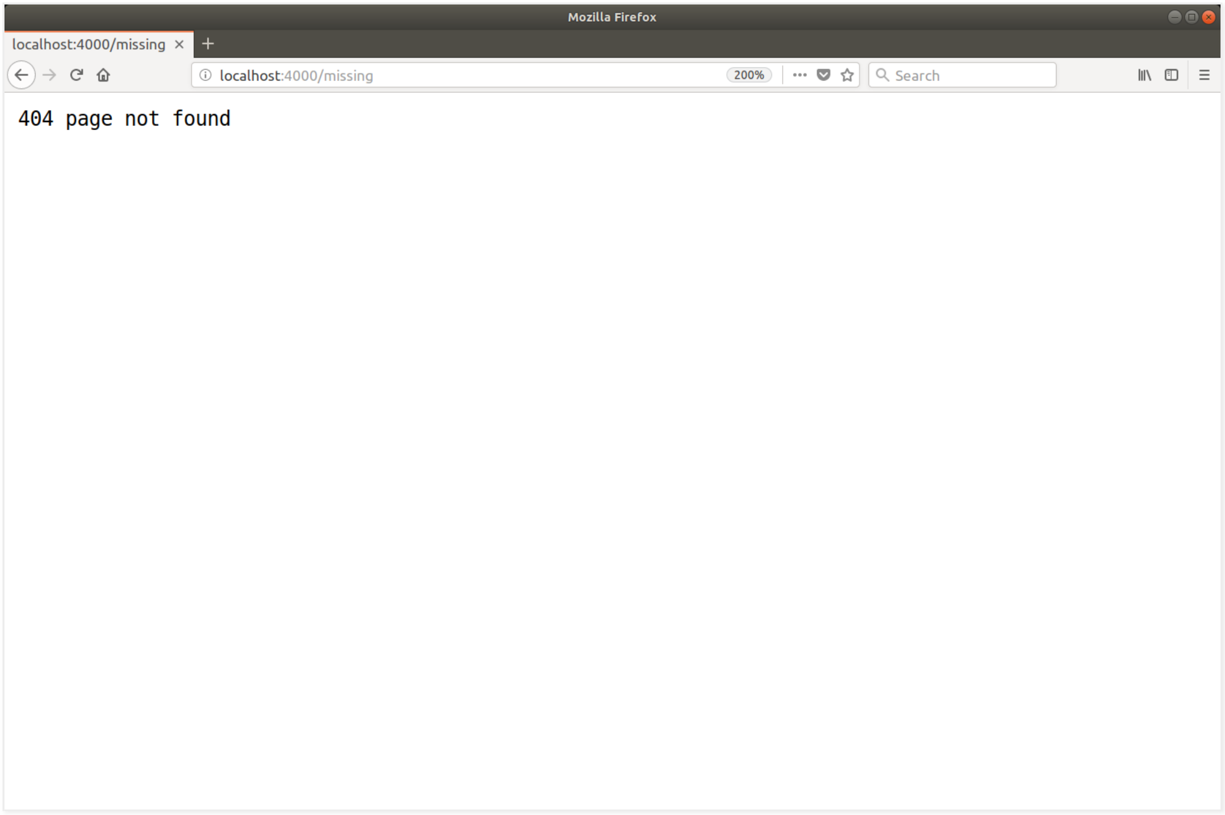
It's not possible to change the behavior of Go's servemux to do this, but you *can* include a simple check in the `home` hander which ultimately has the same effect:

```
File: main.go
```

```go
package main

...
```

```
func home(w http.ResponseWriter, r *http.Request) {
    // Check if the current request URL path exactly matches "/". If it doesn't
    // the http.NotFound() function to send a 404 response to the client.
    // Importantly, we then return from the handler. If we don't return the han
    // would keep executing and also write the "Hello from SnippetBox" message.
    if r.URL.Path != "/" {
        http.NotFound(w, r)
        return
    }

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

...
```

Go ahead and make that change, then restart the server and make a request for an unregistered URL path like `http://localhost:4000/missing`. You should get a `404` response which looks a bit like this:


Mozilla Firefox

localhost:4000/missing

```
404 page not found
```

The DefaultServeMux

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* declaring a servemux, like this:

```go
func main() {
    http.HandleFunc("/", home)
    http.HandleFunc("/snippet", showSnippet)
    http.HandleFunc("/snippet/create", createSnippet)

    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 *DefaultServeMux*. There's nothing special about this — it's just regular servemux like we've already been using, but which is initialized by default and stored in a `net/http` global variable. Here's the relevant line from the Go source code:

```
var DefaultServeMux = NewServeMux()
```

Although this approach can make your code slightly shorter, I don't recommend it for production applications.

Because `DefaultServeMux` is a global variable, any package can access it and register a route — including any third-party packages that your application imports. If one of those third-party packages is compromised, they could use `DefaultServeMux` to expose a malicious handler to the web.

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

Additional Information

Servemux Features and Quirks

- In Go's servemux, longer URL patterns always take precedence over shorter ones. So, if a servemux contains multiple patterns which match a request, it will always dispatch the request to the handler corresponding to the longest pattern. This has the nice side-effect that you can register patterns in any order *and it won't change how the servemux behaves*.

- Request URL paths are automatically sanitized. If the request path contains any `.` or `..` elements or repeated slashes, it will automatically redirect the user 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 URL 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:

```go
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.

What About RESTful Routing?

It's important to acknowledge that the routing functionality provided by Go's servemux is pretty lightweight. It doesn't support routing based on the request method, it doesn't support semantic URLs with variables in them, and it doesn't support regexp-based patterns. If you have a background in using frameworks like Rails, Django or Laravel you might find this a bit restrictive… and surprising!

But don't let that put you off. The reality is that Go's servemux can still get you quite far, and for many applications is perfectly sufficient. For the times that you need more, there's a huge choice of third-party routers that you can use instead of Go's servemux. We'll look at some of the popular options later in the book.

Chapter 2.5.

Customizing HTTP Headers

Let's now update our application so that the `/snippet/create` route only responds to HTTP requests which use the `POST` method, like so:

Method	Pattern	Handler	Action
ANY	/	home	Display the home page
ANY	/snippet	showSnippet	Display a specific snippet
POST	/snippet/create	createSnippet	Create a new snippet

Making this change is important because — later in our application build — requests to the `/snippet/create` route will result in a new snippet being created in a database. Creating a new snippet in a database is a non-idempotent action that changes the state of our server, so we should follow HTTP good practice and restrict this route to act on `POST` requests only.

But the main reason I want to cover this now is because it's a good excuse to talk about HTTP response headers and explain how to customize them.

HTTP Status Codes

Let's begin by updating our `createSnippet()` handler function so that it sends a `405` (method not allowed) HTTP status code *unless* the request method is `POST`. To do this we'll need to use the `w.WriteHeader()` method like so:

```
File: main.go
```

```go
package main

...

func createSnippet(w http.ResponseWriter, r *http.Request) {
    // Use r.Method to check whether the request is using POST or not.
    // If it's not, use the w.WriteHeader() method to send a 405 status code an
    // the w.Write() method to write a "Method Not Allowed" response body. We
    // then return from the function so that the subsequent code is not execute
    if r.Method != "POST" {
        w.WriteHeader(405)
        w.Write([]byte("Method Not Allowed"))
        return
    }

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

...
```

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 OK` 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()`.

Let’s take a look at this in action.

Restart the server, then open a second terminal window and use curl to make a `POST` request to `http://localhost:4000/snippet/create`. You should get a HTTP response with a `200 OK` status code similar to this:

```
$ curl -i -X POST http://localhost:4000/snippet/create
HTTP/1.1 200 OK
Date: Thu, 02 Aug 2018 12:58:54 GMT
Content-Length: 23
Content-Type: text/plain; charset=utf-8

Create a new snippet...
```

But if you use a different request method — like `GET`, `PUT` or `DELETE` — you should now get response with a `405 Method Not Allowed` status code. For example:

```
$ curl -i -X PUT http://localhost:4000/snippet/create
HTTP/1.1 405 Method Not Allowed
Date: Thu, 02 Aug 2018 12:59:16 GMT
Content-Length: 18
Content-Type: text/plain; charset=utf-8

Method Not Allowed
```

Customizing Headers

Another improvement we can make is to include an `Allow: POST` header with every `405 Method Not Allowed` response to let the user know which request methods *are* supported for that particular URL.

We can do this by using the `w.Header().Set()` method to add a new header to the *response header map*, like so:

```
File: main.go
```

```go
package main

...

func createSnippet(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        // Use the Header().Set() method to add an 'Allow: POST' header to the
        // response header map. The first parameter is the header name, and
        // the second parameter is the header value.
        w.Header().Set("Allow", "POST")
        w.WriteHeader(405)
        w.Write([]byte("Method Not Allowed"))
        return
    }


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

...
```

It's important to point out here that changing the header map after a call to `w.WriteHeader()` or `w.Write()` will have no effect on the response headers that the user receives. You need to make sure that your header map contains all the headers you want *before* you call these methods.

Let's take a look at this in action again by sending a non-`POST` request to our `/snippet/create` URL, like so:

```
$ curl -i -X PUT http://localhost:4000/snippet/create
HTTP/1.1 405 Method Not Allowed
Allow: POST
Date: Thu, 02 Aug 2018 13:01:16 GMT
Content-Length: 18
Content-Type: text/plain; charset=utf-8

Method Not Allowed
```

Notice how the response now includes our `Allow: POST` header?

The http.Error Shortcut

If you want to send a non-`200` status code and a plain-text response body (like we are in the code above) then it's a good opportunity to use the `http.Error()` shortcut. This is a lightweight helper function which takes a given message and status code, then calls the `w.WriteHeader()` and `w.Write()` methods behind-the-scenes for us.

Let's update the code to use this instead.

```
File: main.go
```

```
package main

...

func createSnippet(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.Header().Set("Allow", "POST")
        // Use the http.Error() function to send a 405 status code and "Method
        // Allowed" string as the response body.
        http.Error(w, "Method Not Allowed", 405)
        return
    }

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

...
```

In terms of functionality this is almost exactly the same. The biggest difference is that we're now passing our `http.ResponseWriter` to another function, which sends a response to the user for us.

The pattern of passing `http.ResponseWriter` to other functions is super-common in Go, and something we'll do a lot throughout this book. In practice, it's quite rare to use the `w.Write()` and `w.WriteHeader()` methods directly like we have been doing so far. But I wanted to introduce them upfront because they underpin the more advanced (and interesting!) ways to send responses.

Additional Information

Manipulating the Header Map

In the code above we used `w.Header().Set()` to add a new header to the response header map. But there's also `Add()`, `Del()` and `Get()` methods that you can use to read and manipulate 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")
```

System-Generated Headers and Content Sniffing

When sending a response Go will automatically set three *system-generated* headers for you: `Date` and `Content-Length` and `Content-Type`.

The `Content-Type` header is particularly interesting. Go will attempt to set the correct one for you by *content sniffing* 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 new to Go is that it can't distinguish JSON from plain text. And, 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 like so:

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

Header Canonicalization

When you're using the `Add()`, `Get()`, `Set()` and `Del()` 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`). For example:

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

Note: When headers are written to a HTTP/2 connection the header names and values will *always* be converted to lowercase, as per the specifications.

Suppressing System-Generated Headers

The `Del()` method doesn’t remove system-generated headers. To suppress these, you need to access the underlying header map directly and set the value to `nil`. If you want to suppress the `Date` header, for example, you need to write:

```
w.Header()["Date"] = nil
```

Chapter 2.6.

URL Query Strings

While we're on the subject of routing, let's update the `showSnippet` handler so that it accepts an `id` query string parameter from the user like so:

Method	Pattern	Handler	Action
ANY	/	home	Display the home page
ANY	/snippet?id=1	showSnippet	Display a specific snippet
POST	/snippet/create	createSnippet	Create a new snippet

Later we'll use this `id` parameter to select a specific snippet from a database and show it to the user. But for now, we'll just read the value of the `id` parameter and interpolate it with a placeholder response.

To make this work we'll need to update the `showSnippet` handler function to do two things:

1. It needs to retrieve the value of the `id` parameter from the URL query string, which we can do using 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.

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

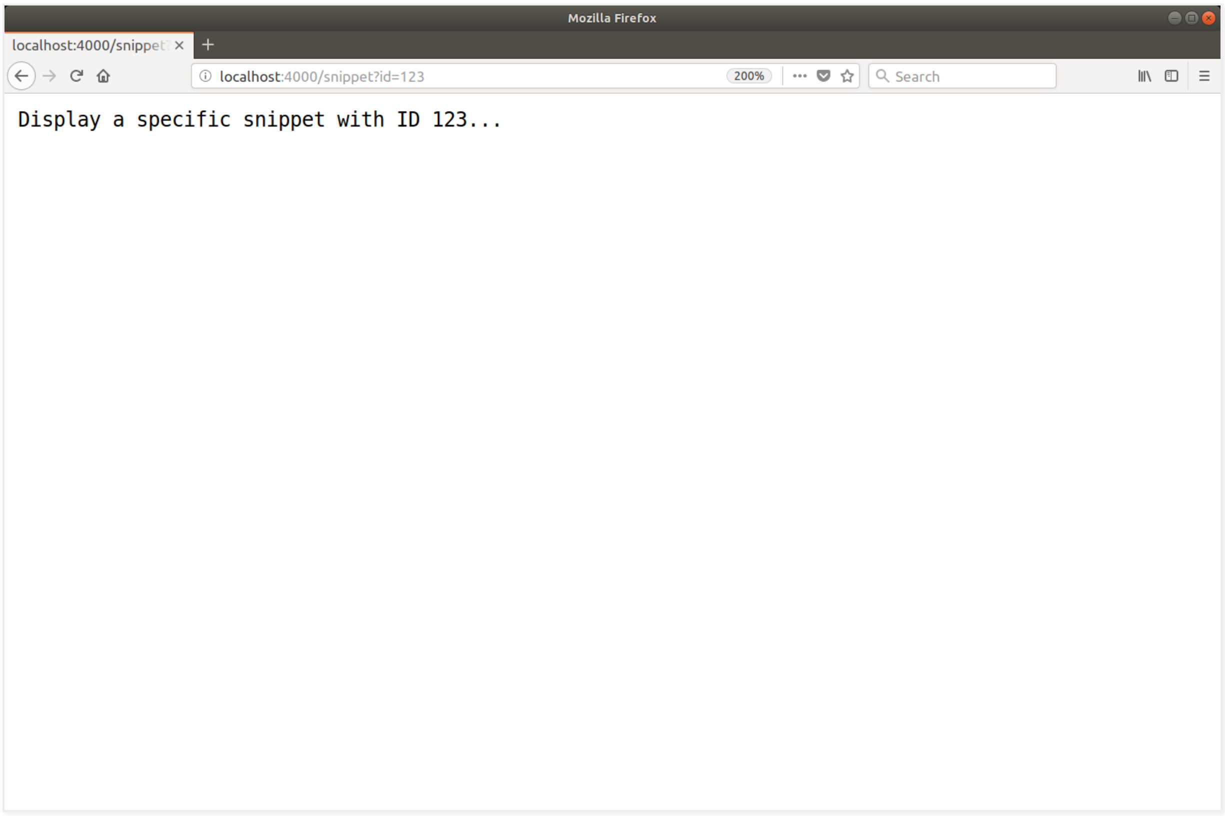
Here's how:

```
File: main.go
```

```go
package main

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

...

func showSnippet(w http.ResponseWriter, r *http.Request) {
    // Extract the value of the id parameter from the query string 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
    // not found response.
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    // Use the fmt.Fprintf() function to interpolate the id value with our resp
    // and write it to the http.ResponseWriter.
    fmt.Fprintf(w, "Display a specific snippet with ID %d...", id)
}

...
```

Let’s try this out.

Restart the application, and try visiting a URL like `http://localhost:4000/snippet?id=123`. You should see a response which looks like this:



```
Display a specific snippet with ID 123...
```

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

- `http://localhost:4000/snippet`

- `http://localhost:4000/snippet?id=-1`
- `http://localhost:4000/snippet?id=foo`

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

The io.Writer Interface

The code above introduced another new thing behind-the-scenes. If you take a look at the documentation for the `fmt.Fprintf()` function you'll notice that it takes an `io.Writer` as the first parameter…

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

…but we passed it our `http.ResponseWriter` object instead — and it worked fine.

We're able to do this because the `io.Writer` type is an *interface*, and the `http.ResponseWriter` object *satisfies the interface* because it has a `w.Write()` method.

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. It's enough to know that — in practice — anywhere you see an `io.Writer` parameter it's OK to pass in your `http.RespsonseWriter` object. Whatever is being written will subsequently be sent as the body of the HTTP response.

Chapter 2.7.

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.

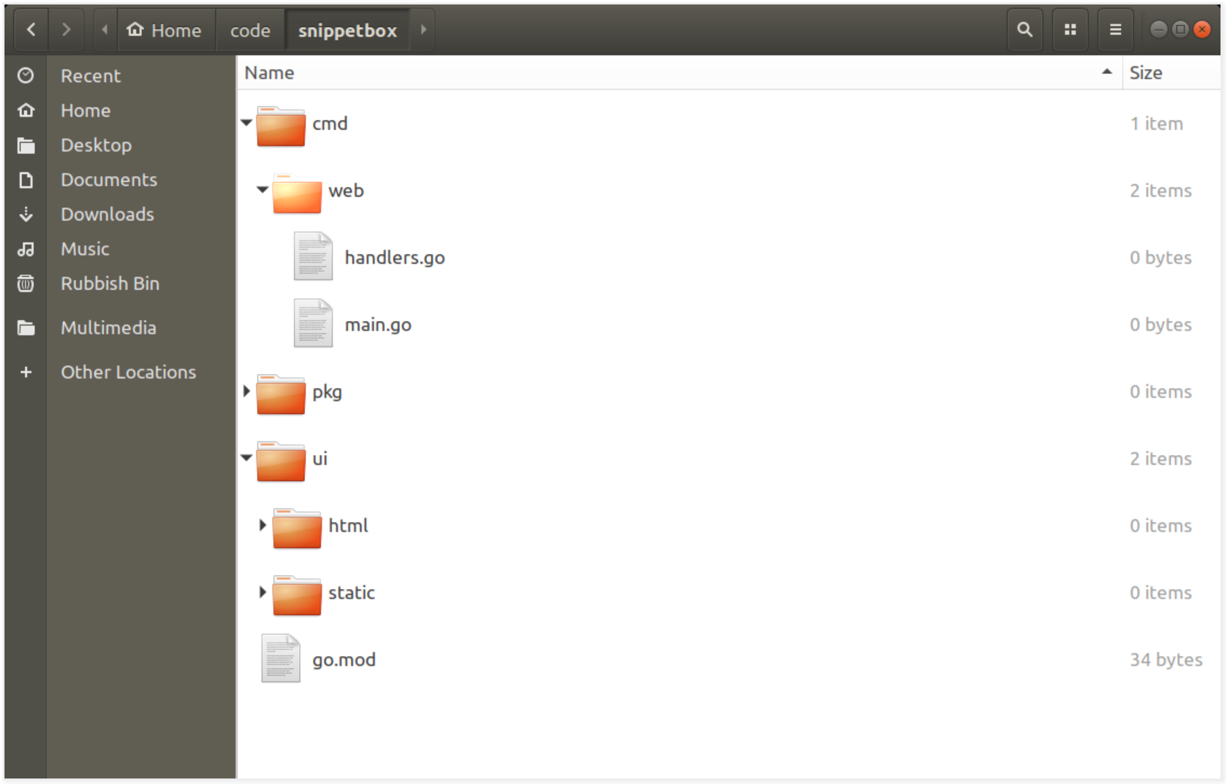
As you gain experience with Go, you'll get a feel for which patterns work well for you in different situations. But as a starting point, the best advice I can give you is *don't over-complicate things*. Try hard to add structure and complexity only when it's demonstrably needed.

For this project we'll implement an outline structure which 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 pkg 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:



Name	Size
cmd	1 item
web	2 items
handlers.go	0 bytes
main.go	0 bytes
pkg	0 items
ui	2 items
html	0 items
static	0 items
go.mod	34 bytes

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 we'll have just one executable application — the web application — which will live under the `cmd/web` directory.

- The `pkg` 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 `pkg` directories, leaving the project root free to hold non-Go assets like UI files, makefiles and module definitions (including our `go.mod` file). This can make things easier to manage when it comes to building and deploying your application in the future.

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 `pkg` directory.

Refactoring Your Existing Code

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

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

```go
package main

import (
    "log"
    "net/http"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", home)
    mux.HandleFunc("/snippet", showSnippet)
    mux.HandleFunc("/snippet/create", createSnippet)

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

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

```go
package main

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

func home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        http.NotFound(w, r)
        return
    }

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

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

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

func createSnippet(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.Header().Set("Allow", "POST")
        http.Error(w, "Method Not Allowed", 405)
        return
    }

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

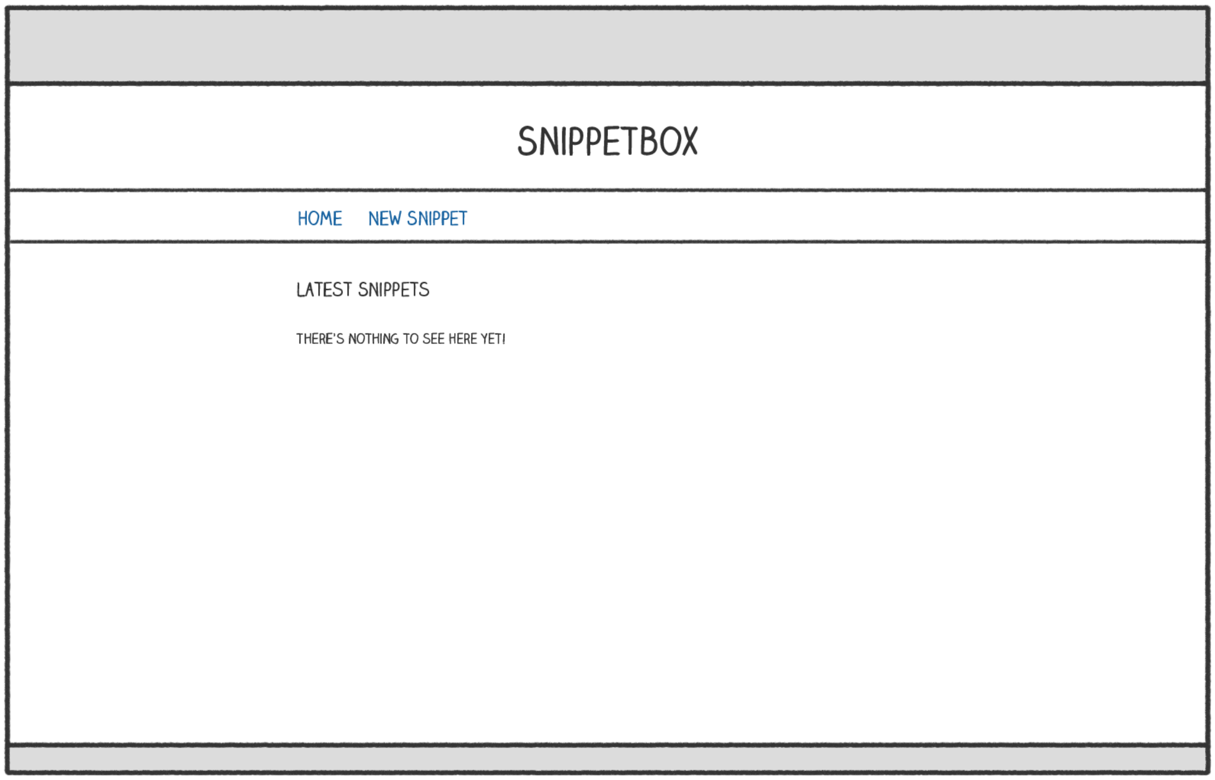
To start the web application you'll now need to execute `go run` using a wildcard pattern, so that it includes both the `main.go` and `handler.go` files:

```
$ go run cmd/web/*
2018/08/02 17:24:28 Starting server on :4000
```

Chapter 2.8.

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:



To do this, let's first create a new *template file* in the `ui/html` directory…

```
$ cd $HOME/code/snippetbox
$ touch ui/html/home.page.tmpl
```

…and then add the following HTML markup for the home page to it:

```
File: ui/html/home.page.tmpl
```

```html
<!doctype html>
<html lang='en'>
    <head>
        <meta charset='utf-8'>
        <title>Home - Snippetbox</title>
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <a href="/">Home</a>
        </nav>
        <section>
            <h2>Latest Snippets</h2>
            <p>There's nothing to see here yet!</p>
        </section>
    </body>
</html>
```

A quick note on naming conventions: Throughout this book we'll use the convention `<name>.<role>.tmpl` for naming template files, where `<role>` is either `page`, `partial` or `layout`. Being able to distinguish the role of the template will help us when it comes to creating a cache of templates later in the book — but actually it doesn't matter so much what naming convention or file extension you use. Go is flexible about this.

So now that we've created a template file with 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 import 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
```

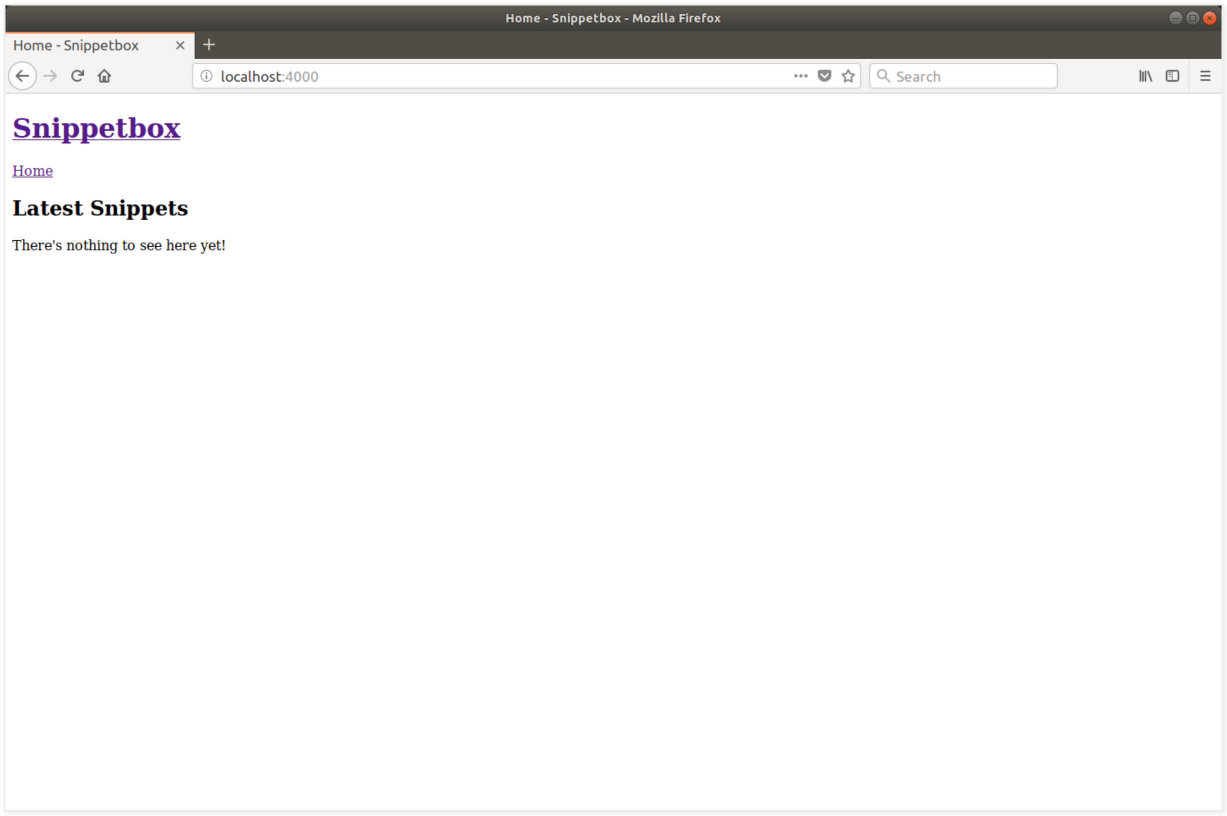
```go
package main

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

func home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        http.NotFound(w, r)
        return
    }

    // 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 and
    // the http.Error() function to send a generic 500 Internal Server Error
    // response to the user.
    ts, err := template.ParseFiles("./ui/html/home.page.tmpl")
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }

    // We then use the Execute() method on the template set to write the templa
    // content as the response body. The last parameter to Execute() represents
    // 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", 500)
    }
}

...
```

It's important to point out that 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.

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/*
2018/08/02 17:49:24 Starting server on :4000
```

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



Snippetbox

Home

Latest Snippets

There's nothing to see here yet!

Template Composition

As we add more pages to this 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 keep our HTML markup DRY and save us typing, it's a good idea to create a *layout* (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.layout.tmpl` file…

```
$ touch ui/html/base.layout.tmpl
```

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

```
File: ui/html/base.layout.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>
        <nav>
            <a href='/'>Home</a>
        </nav>
        <section>
            {{template "body" .}}
        </section>
    </body>
</html>
{{end}}
```

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

Here we're using the `{{define "base"}}...{{end}}` action 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 "body" .}}` actions to denote that we want to *invoke* other named templates (called `title` and `body`) at a particular point 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/home.page.tmpl` and update it to define `title` and `body` named templates containing the specific content for the home page:

```
File: ui/html/home.page.tmpl
```

```
{{template "base" .}}

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

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

Right at the top of this file is arguably the most important part — the `{{template "base" .}}` action. This informs Go that when the `home.page.tmpl` file is executed, that we want to *invoke* the named template `base`.

In turn, the `base` template contains instructions to invoke the `title` and `body` named templates. I know this might feel a bit circular to start with, but stick with me… in practice this pattern works really quite well.

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
```

```go
package main

...

func home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        http.NotFound(w, r)
        return
    }

    // Initialize a slice containing the paths to the two files. Note that the
    // home.page.tmpl file must be the *first* file in the slice.
    files := []string{
        "./ui/html/home.page.tmpl",
        "./ui/html/base.layout.tmpl",
    }

    // Use the template.ParseFiles() function to read the files and store the
    // templates in a template set. Notice that we can pass the slice of file p
    // as a variadic parameter?
    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }
    err = ts.Execute(w, nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
}

...
```

So now, instead of containing HTML directly, our template set contains 3 named templates (`base`, `title` and `body`) and an instruction to invoke the `base` template (which in turn embeds the other two 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).

The big benefit of using this pattern to compose templates is that you're able to cleanly define the page-specific content in individual files on disk, and within those files also control which *layout* template the page uses. This is particularly helpful for larger applications, where different pages of your application may need to use different layouts.

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 some footer content for our web application.

Create a new `ui/html/footer.partial.tmpl` file and add a named template called `footer` like so:

```
$ touch ui/html/footer.partial.tmpl
```

```
File: ui/html/footer.partial.tmpl
```

```
{{define "footer"}}
<footer>Powered by <a href='https://golang.org/'>Go</a></footer>
{{end}}
```

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

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

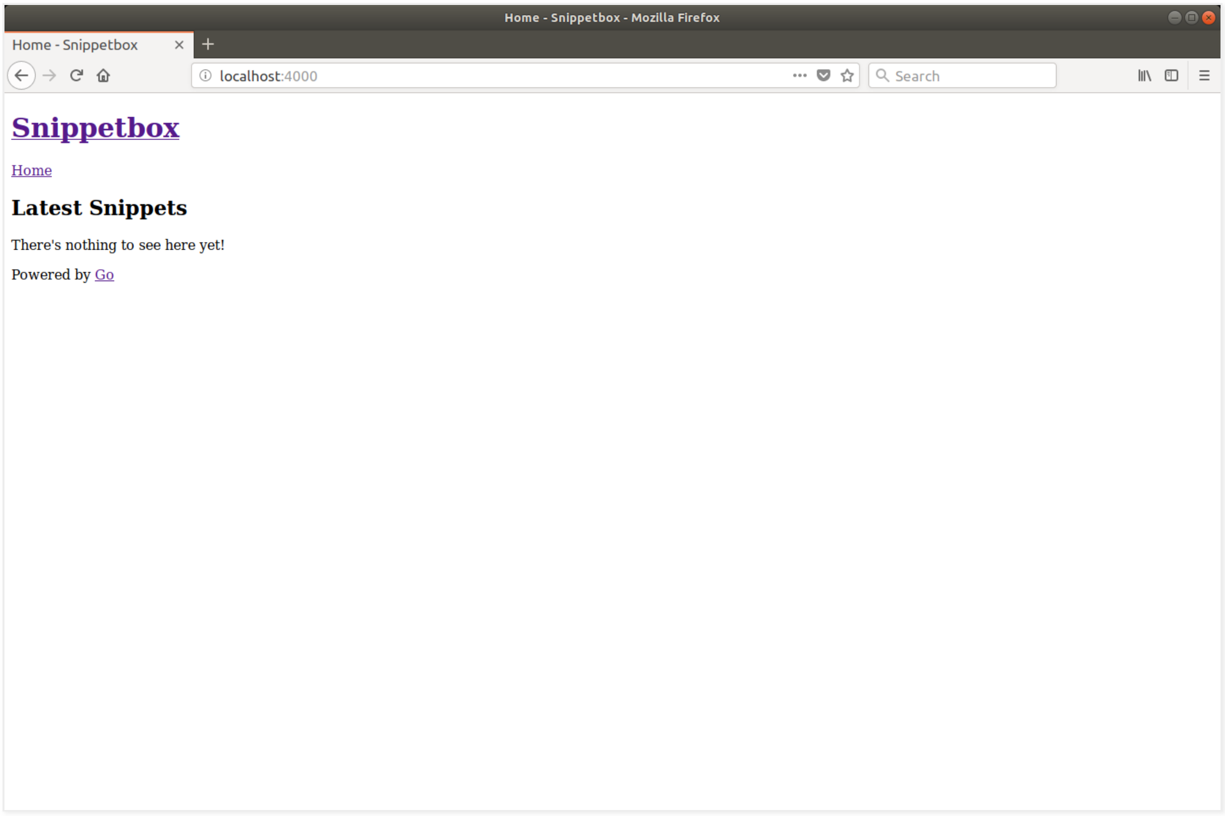
```
{{define "base"}}
<!doctype html>
<html lang='en'>
    <head>
        <meta charset='utf-8'>
        <title>{{template "title" .}} - Snippetbox</title>
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <a href='/'>Home</a>
        </nav>
        <section>
            {{template "body" .}}
        </section>
        <!-- Invoke the footer template -->
        {{template "footer" .}}
    </body>
</html>
{{end}}
```

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

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

```go
package main

...

func home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        http.NotFound(w, r)
        return
    }

    // Include the footer partial in the template files.
    files := []string{
        "./ui/html/home.page.tmpl",
        "./ui/html/base.layout.tmpl",
        "./ui/html/footer.partial.tmpl",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }
    err = ts.Execute(w, nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
}

...
```

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



Snippetbox

Home

Latest Snippets

There's nothing to see here yet!

Powered by Go

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.

Chapter 2.9.

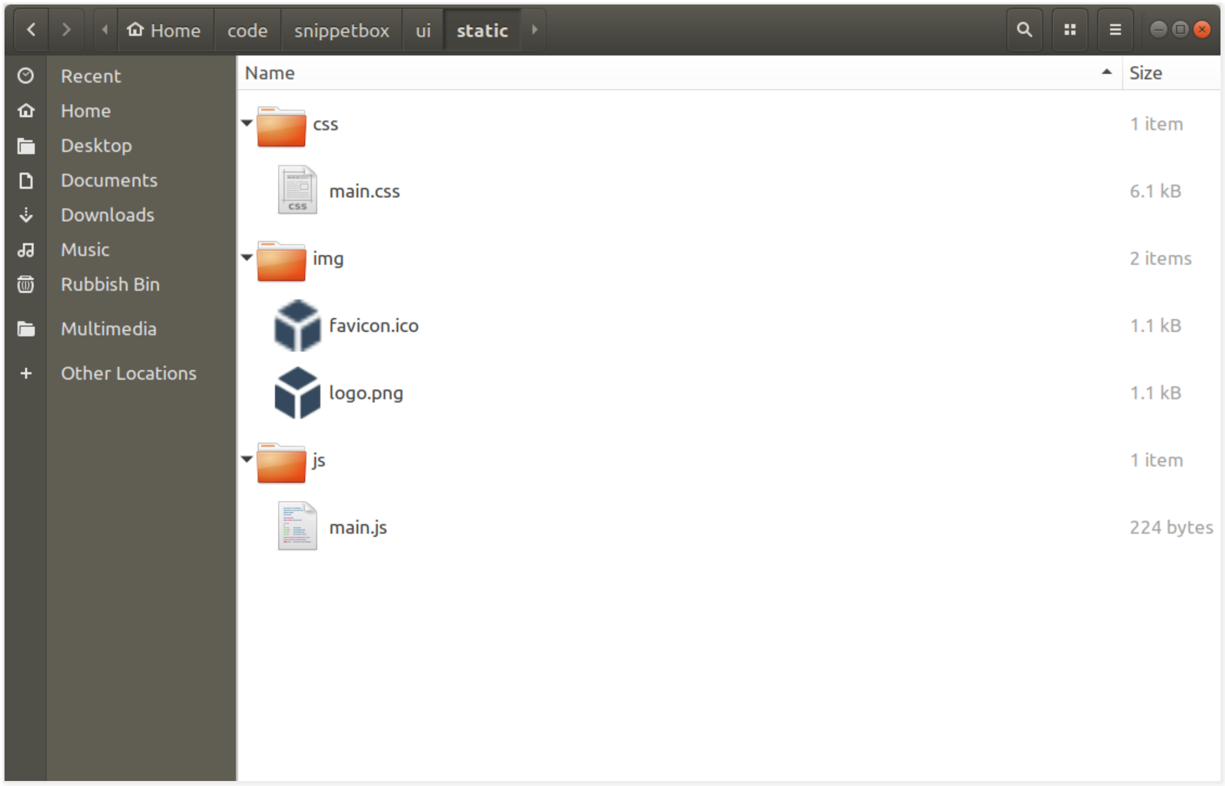
Serving Static Files

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.alexedwards.net/static/sb.v120.tar.gz | tar -xvz -C ./ui/stat
```

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



Name	Size
css	1 item
main.css	6.1 kB
img	2 items
favicon.ico	1.1 kB
logo.png	1.1 kB
js	1 item
main.js	224 bytes

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 requests which begin with `"/static/"` are handled using this, like so:

Method	Pattern	Handler	Action
ANY	/	home	Display the home page
ANY	/snippet?id=1	showSnippet	Display a specific snippet
POST	/snippet/create	createSnippet	Create a new snippet
ANY	/static/	http.FileServer	Serve a specific static file

Remember: The pattern `"/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, it will remove the leading slash from the URL path and then search the `./ui/static` directory for the corresponding file to send to the user.

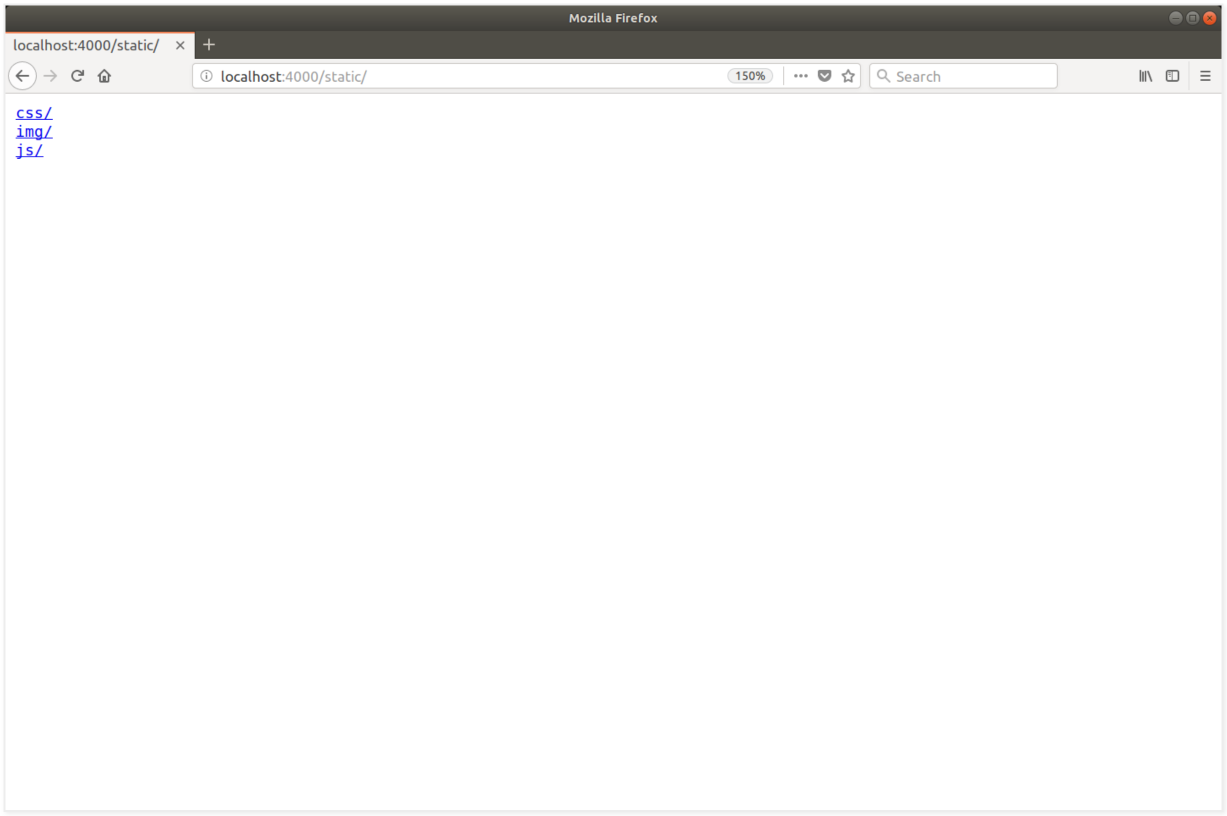
So, for this to work correctly, we must strip the leading `"/static"` from the URL path *before* passing it to `http.FileServer`. Otherwise it will be looking for a file which doesn't exist and the user will receive a `404 page not found` response. Fortunately Go includes a `http.StripPrefix()` helper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

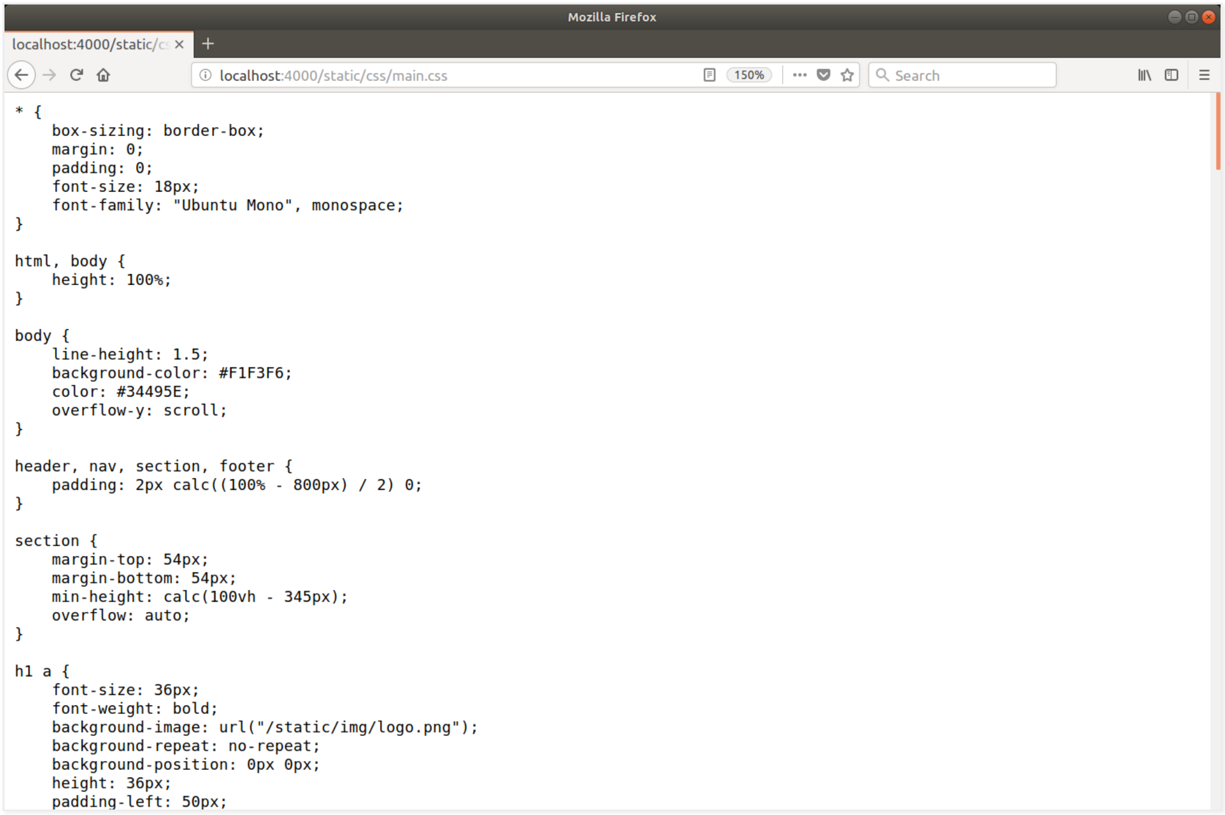
```
package main

import (
    "log"
    "net/http"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", home)
    mux.HandleFunc("/snippet", showSnippet)
    mux.HandleFunc("/snippet/create", createSnippet)

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

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

    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:



```
css/
img/
js/
```

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:



```
* {
    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, section, footer {
    padding: 2px calc((100% - 800px) / 2) 0;
}

section {
    margin-top: 54px;
    margin-bottom: 54px;
    min-height: calc(100vh - 345px);
    overflow: auto;
}

h1 a {
    font-size: 36px;
    font-weight: bold;
    background-image: url("/static/img/logo.png");
    background-repeat: no-repeat;
    background-position: 0px 0px;
    height: 36px;
    padding-left: 50px;
```

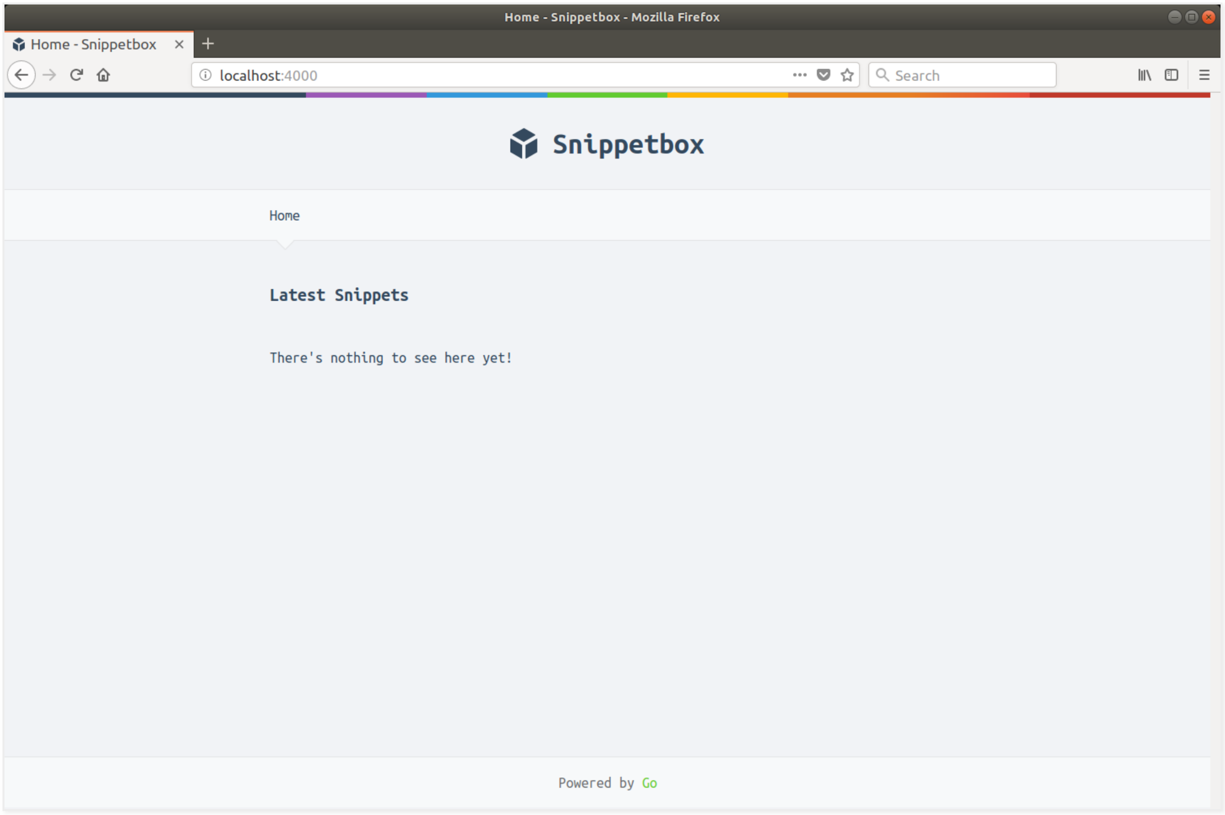
Using the Static Files

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

File: ui/html/base.layout.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-
        <!-- Also link to some fonts hosted by Google -->
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ub
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <a href='/'>Home</a>
        </nav>
        <section>
            {{template "body" .}}
        </section>
        {{template "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 visit `http://localhost:4000`. Your home page should now look like this:



Latest Snippets

There's nothing to see here yet!

Powered by Go

Additional Information

Features and Functions

Go's file server 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/
  HTTP/1.1 206 Partial Content
  Accept-Ranges: bytes
  Content-Length: 100
  Content-Range: bytes 100-199/1075
  Content-Type: image/png
  Last-Modified: Thu, 04 May 2017 13:07:52 GMT
  Date: Wed, 08 Aug 2018 16:21:16 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.

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")
}
```

But be aware: `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.

Chapter 2.10.

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 a `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{})
```

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). 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 need to *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 content 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 code 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 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.

Chapter 3.

Configuration and Error Handling

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

You'll learn how to:

- Set configuration settings for your application at runtime in an easy and idiomatic way using command-line flags.
- Improve your application log messages to include more information, and manage them differently depending on the type (or *level*) of log message.
- 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.

Chapter 3.1.

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, and make 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 from 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
    // and some short help text explaining what the flag controls. The value of
    // 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
    // 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 er
    // encountered during parsing the application will be terminated.
    flag.Parse()

    mux := http.NewServeMux()
    mux.HandleFunc("/", home)
    mux.HandleFunc("/snippet", showSnippet)
    mux.HandleFunc("/snippet/create", createSnippet)

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

    // The value returned from the flag.String() function is a pointer to the f
    // value, not the value itself. So we need to dereference the pointer (i.e.
    // prefix it with the * symbol) before using it.
    log.Printf("Starting server on %s", *addr)
    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"
2018/08/08 19:15:47 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/*
2018/08/08 19:28:11 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 log an error and exit.

Go also has a range of other functions including `flag.Int()`, `flag.Bool()` and `flag.Float64()`. These work in exactly the same way as `flag.String()`, except they automatically convert the command-line flag value to the appropriate type. We'll use some of these different functions later in the book.

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-build786121279/b001/exe/handlers:
    -addr string
        HTTP network address (default ":4000")
exit status 2
```

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 have 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()` 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
2018/08/08 19:45:29 Starting server on :9999
```

Boolean Flags

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

```
$ go run example.go -flag=true
$ go run example.go -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 the `flag.StringVar()`, `flag.IntVar()`, `flag.BoolVar()` and other functions. This can be 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
}

...

cfg := new(Config)
flag.StringVar(&cfg.Addr, "addr", ":4000", "HTTP network address")
flag.StringVar(&cfg.StaticDir, "static-dir", "./ui/static", "Path to static ass
flag.Parse()
```

Chapter 3.2.

Leveled Logging

At the moment in our `main.go` file we're logging messages using the `log.Printf()` and `log.Fatal()` functions like so:

```
log.Printf("Starting server on %s", *addr) // Information message
err := http.ListenAndServe(*addr, mux)
log.Fatal(err) // Error message
```

We can break these log messages down into two distinct types, or *levels*. The first type is *informational* messages (like `"Starting server on :4000"`) and the second type is *error* messages.

Currently both types of message are outputted using Go's *standard logger*, which prefixes the message with the local date and time and then outputs it to your standard error (sterr) stream.

Let's improve our application by adding some *leveled logging* capability, so that information and error messages are managed slightly differently. Specifically:

- We will prefix informational messages with `"INFO"` and output the message to standard out (stdout).

- We will prefix error messages with `"ERROR"` and output them to standard error (sterr), along with the relevant file name and line number that called the logger (to help with debugging).

There are a couple of different ways to do this, but a simple and clean approach is to use the `log.New()` function to create two new *custom loggers*.

Open up your `main.go` file and update it as follows:

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

```go
package main

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

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

    // Use log.New() to create a logger for writing information messages. This
    // three parameters: the destination to write the logs to (os.Stdout), a st
    // prefix for message (INFO followed by a tab), and flags to indicate what
    // additional information to include (local date and time). Note that the f
    // are joined using the bitwise OR operator |.
    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)

    // Create a logger for writing error messages in the same way, but use stde
    // the destination and use the log.Lshortfile flag to include the relevant
    // file name and line number.
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    mux := http.NewServeMux()
    mux.HandleFunc("/", home)
    mux.HandleFunc("/snippet", showSnippet)
    mux.HandleFunc("/snippet/create", createSnippet)

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

    // Write messages using the two new loggers, instead of the standard logger
    infoLog.Printf("Starting server on %s", *addr)
    err := http.ListenAndServe(*addr, mux)
    errorLog.Fatal(err)
}
```

Alright… let's try these 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/*
INFO      2018/08/09 20:10:43 Starting server on :4000
ERROR    2018/08/09 20:10:43 main.go:30: listen tcp :4000: bind: address already
exit status 1
```

Notice how the two messages are prefixed differently — so they can be easily distinguished in the terminal — and our error message also includes the file name and line number (`main.go:30`) that called the logger?

Hint: If you want to include the full file path in your log output, instead of just the file name, you can use the `log.Llongfile` flag instead of `log.Lshortfile` when creating your custom logger. You can also force your logger to use UTC datetimes, instead of local ones, with the `log.LUTC` flag.

A big benefit of logging your messages to the standard streams (stdout and sterr) like we are 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 streams are displayed in the terminal.

In staging or production environments, you can redirect the streams 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 stdout and stderr streams to on-disk files when starting the application like so:

```
$ go run cmd/web/* >>/tmp/info.log 2>>/tmp/error.log
```

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

The http.Server Error Log

There is one more change we need to make to our application. By default, if Go's HTTP server encounters an error it will log it using the standard logger. For consistency it'd be better to use our new `errorLog` logger instead.

To make this happen we need to *initialize a new* `http.Server` *struct* containing the configuration settings for our server, instead of using the `http.ListenAndServe()` shortcut.

It's probably easiest to demonstrate this:

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

```go
package main

...

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

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    mux := http.NewServeMux()
    mux.HandleFunc("/", home)
    mux.HandleFunc("/snippet", showSnippet)
    mux.HandleFunc("/snippet/create", createSnippet)

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

    // Initialize a new http.Server struct. We set the Addr and Handler fields
    // that the server uses the same network address and routes as before, and
    // the ErrorLog field so that the server now uses the custom errorLog logge
    // the event of any problems.
    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  mux,
    }

    infoLog.Printf("Starting server on %s", *addr)
    // Call the ListenAndServe() method on our new http.Server struct.
    err := srv.ListenAndServe()
    errorLog.Fatal(err)
}
```

Additional Information

Additional Logging Methods

In the code so far we've used the `Println()`, `Printf()` and `Fatal()` methods to write log messages, but Go provides a range of other methods that are worth familiarizing yourself with.

As a rule of thumb, you should avoid using the `Panic()` and `Fatal()` variations outside of your `main()` function — it's good practice to return errors instead, and only panic or exit directly from `main()`.

Concurrent Logging

Custom loggers created by `log.New()` are concurrency-safe. You can share a single logger and use it across multiple goroutines and in your handlers without needing to worry about race conditions.

That said, if you have multiple 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.

Logging to a File

As I said above, my general recommendation is to log your output to standard streams and redirect the output to a file at runtime. But if you *don't* want to do this, you can always open a file in Go and use it as your log destination. As a rough example:

```go
f, err := os.OpenFile("/tmp/info.log", os.O_RDWR|os.O_CREATE, 0666)
if err != nil {
    log.Fatal(err)
}
defer f.Close()

infoLog := log.New(f, "INFO\t", log.Ldate|log.Ltime)
```

Chapter 3.3.

Dependency Injection

There's one more problem with our logging that we need to address. 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 `errorLog` logger that we want to be using.

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

    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }

    err = ts.Execute(w, nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
}
```

This raises a good question: *how can we make our new `errorLog` 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.

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

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

```
package main

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

// Define an application struct to hold the application-wide dependencies for t
```

```go
// web application. For now we'll only include fields for the two custom logger
// we'll add more to it as the build progresses.
type application struct {
    errorLog *log.Logger
    infoLog  *log.Logger
}


func main() {
    ...
}
```

And then in the `handlers.go` file update your handler functions so that they become *methods against the* `application` *struct…*

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

```go
package main

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

// Change the signature of the home handler so it is defined as a method agains
// *application.
func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        http.NotFound(w, r)
        return
    }

    files := []string{
        "./ui/html/home.page.tmpl",
        "./ui/html/base.layout.tmpl",
        "./ui/html/footer.partial.tmpl",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        // Because the home handler function is now a method against applicatio
        // it can access its fields, including the error logger. We'll write th
        // message to this instead of the standard logger.
        app.errorLog.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }

    err = ts.Execute(w, nil)
    if err != nil {
        // Also update the code here to use the error logger from the applicati
        // struct.
        app.errorLog.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
}

// Change the signature of the showSnippet handler so it is defined as a method
// against *application.
func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("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 createSnippet handler so it is defined as a meth
// against *application.
func (app *application) createSnippet(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.Header().Set("Allow", "POST")
        http.Error(w, "Method Not Allowed", 405)
        return
    }

    w.Write([]byte("Create 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"
    "net/http"
    "os"
)

type application struct {
    errorLog *log.Logger
    infoLog  *log.Logger
}

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

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    // Initialize a new instance of application containing the dependencies.
    app := &application{
        errorLog: errorLog,
        infoLog:  infoLog,
    }

    // Swap the route declarations to use the application struct's methods as t
    // handler functions.
    mux := http.NewServeMux()
    mux.HandleFunc("/", app.home)
    mux.HandleFunc("/snippet", app.showSnippet)
    mux.HandleFunc("/snippet/create", app.createSnippet)

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

    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  mux,
    }

    infoLog.Printf("Starting server on %s", *addr)
    err := srv.ListenAndServe()
    errorLog.Fatal(err)
}
```

I understand that this approach might feel a bit complicated and convoluted, especially when an alternative is to simply make the `infoLog` and `errorLog` loggers global variables. 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/home.page.tmpl` to `ui/html/home.page.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/home.page.tmpl` no longer exists.

Go ahead and make the change:

```
$ cd $HOME/code/snippetbox
$ mv ui/html/home.page.tmpl ui/html/home.page.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 error message in your terminal similar to this:

```
$ go run cmd/web/*
INFO      2018/08/30 21:28:33 Starting server on :4000
ERROR    2018/08/30 21:28:41 handlers.go:25: open ./ui/html/home.page.tmpl: no
```

Notice how the log message is now prefixed with `ERROR` and originated from line `25` of the `handlers.go` file? This demonstrates nicely that our custom `errorLog` logger is 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 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 `config` package exporting an `Application` struct and have your handler functions close over this to form a *closure*. Very roughly:

```go
func main() {
    app := &config.Application{
        ErrorLog: log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortf
    }

    mux.Handle("/", handlers.Home(app))
}
```

```go
func Home(app *config.Application) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ...
        ts, err := template.ParseFiles(files...)
        if err != nil {
            app.ErrorLog.Println(err.Error())
            http.Error(w, "Internal Server Error", 500)
            return
        }
        ...
    }
}
```

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

Chapter 3.4.

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:

```
$ cd $HOME/code/snippetbox
$ touch cmd/web/helpers.go
```

And add the following code:

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

```go
package main

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

// The serverError helper writes an error message and stack trace to the errorL
// then sends a generic 500 Internal Server Error response to the user.
func (app *application) serverError(w http.ResponseWriter, err error) {
    trace := fmt.Sprintf("%s\n%s", err.Error(), debug.Stack())
    app.errorLog.Println(trace)

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

// The clientError helper sends a specific status code and corresponding descri
// to the user. We'll use this later in the book to send responses like 400 "Ba
// 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)
}

// For consistency, we'll also implement a notFound helper. This is simply a
// convenience wrapper around clientError which sends a 404 Not Found response
// the user.
func (app *application) notFound(w http.ResponseWriter) {
    app.clientError(w, http.StatusNotFound)
}
```

There’s not a huge amount of new code here, but it does introduce some new features which are worth discussing.

- In the `serverError()` helper we use the `debug.Stack()` function to get a *stack trace* for the *current goroutine* and append it to the log message. Being able to see the execution path of the application via the stack trace can be helpful when you’re trying to debug errors.

- In the `clientError()` helper we use the `http.StatusText()` function to automatically generate a human-friendly text representation of a given HTTP status code. For example, `http.StatusText(400)` will return the string `"Bad Request"`.

- We've started using the `net/http` package's named constants for HTTP status codes, instead of integers. In the `serverError()` helper we've used the constant `http.StatusInternalServerError` instead of writing `500`, and in the `notFound()` helper we've used the constant `http.StatusNotFound` instead of writing `404`.

  Using status constants is a nice touch which helps make your code clear and self-documenting — especially when dealing with less-commonly-used status codes. You can find the complete list of status code constants here.

Once that's done, head back to your `handlers.go` file and update it to use the new helpers:

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

```go
package main

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

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        app.notFound(w) // Use the notFound() helper
        return
    }

    files := []string{
        "./ui/html/home.page.tmpl",
        "./ui/html/base.layout.tmpl",
        "./ui/html/footer.partial.tmpl",
    }

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

    err = ts.Execute(w, nil)
    if err != nil {
        app.serverError(w, err) // Use the serverError() helper.
    }
}

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        app.notFound(w) // Use the notFound() helper.
        return
    }

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

func (app *application) createSnippet(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.Header().Set("Allow", "POST")
        app.clientError(w, http.StatusMethodNotAllowed) // Use the clientError(
        return
    }

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

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 error message and stack trace in your terminal:

```
$ go run cmd/web/*
INFO      2018/08/31 12:04:51 Starting server on :4000
ERROR   2018/08/31 12:04:57 helpers.go:14: open ./ui/html/home.page.tmpl: no su
goroutine 5 [running]:
runtime/debug.Stack(0xc000073cb0, 0xc00001e2c0, 0x36)
    /usr/local/go/src/runtime/debug/stack.go:24 +0xa7
main.(*application).serverError(0xc000010c40, 0x809040, 0xc0000f00e0, 0x8061c0,
    /home/alex/code/snippetbox/cmd/web/helpers.go:14 +0x62
main.(*application).home(0xc000010c40, 0x809040, 0xc0000f00e0, 0xc00009e400)
    /home/alex/code/snippetbox/cmd/web/handlers.go:25 +0x1d2
main.(*application).home-fm(0x809040, 0xc0000f00e0, 0xc00009e400)
    /home/alex/code/snippetbox/cmd/web/routes.go:7 +0x48
net/http.HandlerFunc.ServeHTTP(0xc000010c50, 0x809040, 0xc0000f00e0, 0xc00009e40
    /usr/local/go/src/net/http/server.go:1964 +0x44
net/http.(*ServeMux).ServeHTTP(0xc000072fc0, 0x809040, 0xc0000f00e0, 0xc00009e40
    /usr/local/go/src/net/http/server.go:2361 +0x127
net/http.serverHandler.ServeHTTP(0xc000075040, 0x809040, 0xc0000f00e0, 0xc00009e
    /usr/local/go/src/net/http/server.go:2741 +0xab
net/http.(*conn).serve(0xc00007edc0, 0x8092c0, 0xc00005c2c0)
    /usr/local/go/src/net/http/server.go:1847 +0x646
created by net/http.(*Server).Serve
    /usr/local/go/src/net/http/server.go:2851 +0x2f5
```

If you look closely at this you'll notice a small problem: the file name and line number being reported in the log message is now `helpers.go:14` — because this is where the log message is now being written from.

What we want to report is the file name and line number *one step back* in the stack trace, which would give us a clearer idea of where the error actually originated from.

We can do this by changing the `serverError()` helper to use our logger's `Output()` function and setting the *frame depth* to 2. Reopen your `helpers.go` file and update it like so:

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

```
package main

...

func (app *application) serverError(w http.ResponseWriter, err error) {
    trace := fmt.Sprintf("%s\n%s", err.Error(), debug.Stack())
    app.errorLog.Output(2, trace)

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

...
```

And if you try again now, you should find that the appropriate file name and line number (`handlers.go:25`) is being reported:

```
$ go run cmd/web/*
INFO	2018/08/31 12:45:51 Starting server on :4000
ERROR	2018/08/31 12:45:54 handlers.go:25: open ./ui/html/home.page.tmpl: no s
goroutine 19 [running]:
runtime/debug.Stack(0xc000087cb0, 0xc000116240, 0x36)
    /usr/local/go/src/runtime/debug/stack.go:24 +0xa7
```

```
main.(*application).serverError(0xc000082c00, 0x809040, 0xc0001140e0, 0x8061c0,
    /home/alex/code/snippetbox/cmd/web/helpers.go:14 +0x62
main.(*application).home(0xc000082c00, 0x809040, 0xc0001140e0, 0xc0000be400)
    /home/alex/code/snippetbox/cmd/web/handlers.go:25 +0x1d2
main.(*application).home-fm(0x809040, 0xc0001140e0, 0xc0000be400)
    /home/alex/code/snippetbox/cmd/web/routes.go:7 +0x48
net/http.HandlerFunc.ServeHTTP(0xc000082c10, 0x809040, 0xc0001140e0, 0xc0000be40
    /usr/local/go/src/net/http/server.go:1964 +0x44
net/http.(*ServeMux).ServeHTTP(0xc000086fc0, 0x809040, 0xc0001140e0, 0xc0000be40
    /usr/local/go/src/net/http/server.go:2361 +0x127
net/http.serverHandler.ServeHTTP(0xc000089450, 0x809040, 0xc0001140e0, 0xc0000be
    /usr/local/go/src/net/http/server.go:2741 +0xab
net/http.(*conn).serve(0xc00009ed20, 0x8092c0, 0xc000092280)
    /usr/local/go/src/net/http/server.go:1847 +0x646
created by net/http.(*Server).Serve
    /usr/local/go/src/net/http/server.go:2851 +0x2f5
```

Revert the Deliberate Error

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

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

Chapter 3.5.

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:

```
$ cd $HOME/code/snippetbox
$ touch cmd/web/routes.go
```

File: cmd/web/routes.go

```go
package main

import "net/http"

func (app *application) routes() *http.ServeMux {
    mux := http.NewServeMux()
    mux.HandleFunc("/", app.home)
    mux.HandleFunc("/snippet", app.showSnippet)
    mux.HandleFunc("/snippet/create", app.createSnippet)

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

    return mux
}
```

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

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

```go
package main

...

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

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    app := &application{
        errorLog: errorLog,
        infoLog:  infoLog,
    }

    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  app.routes(), // Call the new app.routes() method
    }

    infoLog.Printf("Starting server on %s", *addr)
    err := srv.ListenAndServe()
    errorLog.Fatal(err)
}
```

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.

Chapter 4.

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.

In this section you'll learn how to:

- 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.

Chapter 4.1.

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 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.
    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-
    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\
    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 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     | 2019-09-07 11:46:10 |
|  2 | Over the wintry forest | 2019-09-07 11:46:10 |
|  3 | First autumn morning   | 2018-09-14 11:46:11 |
+----+------------------------+---------------------+

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

Chapter 4.2.

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: finding github.com/go-sql-driver/mysql v1.4.0
go: downloading github.com/go-sql-driver/mysql v1.4.0
```

Notice here that we're postfixing the package path with `@v1` to indicate that we want to download the latest available version of the package *with* the major release number 1.

At the time of writing this is `v1.4.0`, but the version you download might be `v1.4.1`, `v1.5.0` 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
```

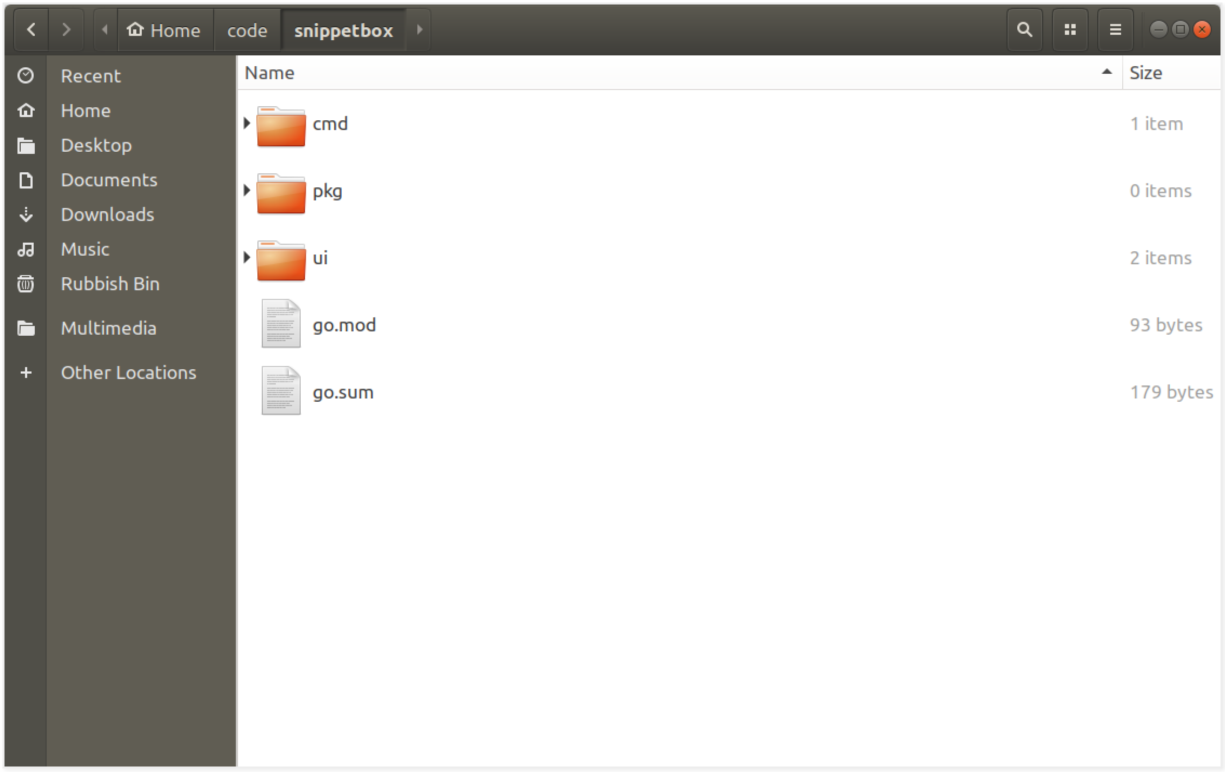
Or if you want to download a specific version of a package, you can use the full version number. For example:

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

Once the driver is installed, take a look at your `go.mod` file (which we created right at the start of the book). You should see a new `require` line containing the package path and exact version that was downloaded:

```
File: go.mod

module alexedwards.net/snippetbox

require github.com/go-sql-driver/mysql v1.4.0 // indirect
```

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

github.com/go-sql-driver/mysql v1.4.0 h1:7LxgVwFb2hIQtMm87NdgAVfXjnt4OePseqT1tK
github.com/go-sql-driver/mysql v1.4.0/go.mod h1:zAC/RDZ24gD3HViQzih4MyKcchzm+sO
```

Unlike the go.mod file, go.sum isn't designed to me 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.

- 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 dependencies they are downloading and the checksums in the file.

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’s 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 in your build 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 could run `go mod tidy`, which will automatically remove any unused packages from your `go.mod` and `go.sum` files.

```
$ go mod tidy -v
```

Chapter 4.3.

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 essentiall
// 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 these connections 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 handler itself — it would be a waste of memory and network resources.

Usage 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"
    "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 dat
    flag.Parse()

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

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

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

    app := &application{
        errorLog: errorLog,
        infoLog:  infoLog,
    }

    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  app.routes(),
    }

    infoLog.Printf("Starting server on %s", *addr)
    err = srv.ListenAndServe()
    errorLog.Fatal(err)
}

// 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
    }
    if err = db.Ping(); err != nil {
        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. 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.

- 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 `errorLog.Fatal()`. In both of those cases, the program exits immediately and deferred functions are never run. But including `db.Close()` is a good habit to get into and it could be beneficial later 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/*
INFO    2018/09/07 13:54:19 Starting server on :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/*
ERROR    2018/09/07 13:55:51 main.go:44: Error 1045: Access denied for user 'web
exit status 1
```

Chapter 4.4.

Designing a Database Model

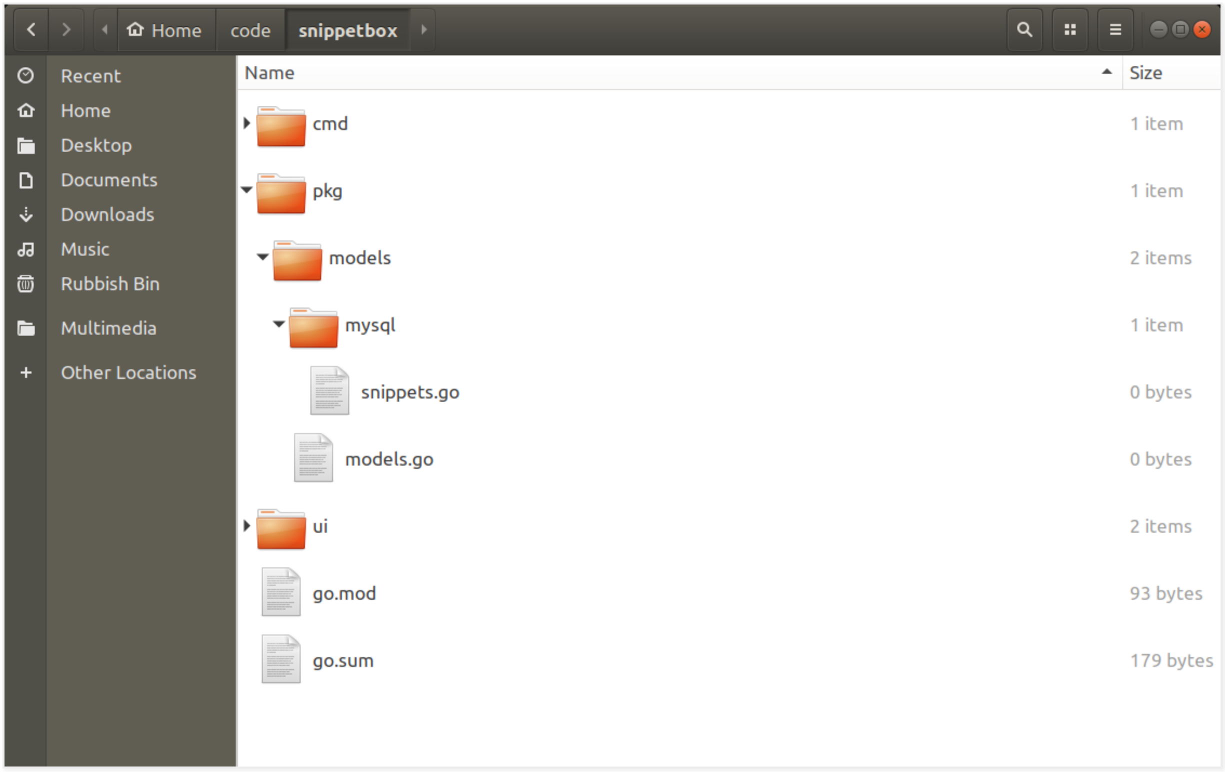
In this chapter we're going to sketch out a database model for our project.

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 couple of new folders and files under the `pkg` directory:

```
$ cd $HOME/code/snippetbox
$ mkdir -p pkg/models/mysql
$ touch pkg/models/models.go
$ touch pkg/models/mysql/snippets.go
```



```
cmd            1 item
pkg            1 item
  models       2 items
    mysql      1 item
      snippets.go   0 bytes
    models.go  0 bytes
ui             2 items
go.mod         93 bytes
go.sum         179 bytes
```

Remember: The pkg 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.

We'll start by using the `pkg/models/models.go` file to define the top-level data types that our database model will use and return. Open it up and add the following code:

File: pkg/models/models.go

```
package models

import (
    "errors"
    "time"
)
```

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

type Snippet struct {
    ID      int
    Title   string
    Content string
    Created time.Time
    Expires time.Time
}
```

Notice how the fields of the `Snippet` struct correspond to the fields in our MySQL `snippets` table?

Now let's move on to the `pkg/models/mysql/snippets.go` file, which will contain the code specifically for working with the snippets in our MySQL database. In this file we're going to define a new `SnippetModel` type and implement some methods on it to access and manipulate the database. Like so:

File: pkg/models/mysql/snippets.go

```
package mysql

import (
    "database/sql"

    "alexedwards.net/snippetbox/pkg/models"
)

// 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, content, expires string) (int, error) {
    return 0, nil
}

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

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

One other important thing to point out here is the import statement for `alexedwards.net/snippetbox/pkg/models`. Notice how the import path for our internal package is prefixed with the module path I chose right at the start of the book?

Using the SnippetModel

To use this model in our handlers we need to establish a new `SnippetModel` struct in `main()` 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"
    "net/http"
    "os"

    "alexedwards.net/snippetbox/pkg/models/mysql" // New import

    _ "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 {
    errorLog *log.Logger
    infoLog  *log.Logger
    snippets *mysql.SnippetModel
}

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

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    db, err := openDB(*dsn)
    if err != nil {
        errorLog.Fatal(err)
    }
    defer db.Close()
    // Initialize a mysql.SnippetModel instance and add it to the application
    // dependencies.
    app := &application{
        errorLog: errorLog,
        infoLog:  infoLog,
        snippets: &mysql.SnippetModel{DB: db},
    }

    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  app.routes(),
    }

    infoLog.Printf("Starting server on %s", *addr)
    err = srv.ListenAndServe()
    errorLog.Fatal(err)
}

...
```

Additional Information

Benefits of This Structure

Setting your models up in this way might seem a bit complex and convoluted, especially if you're new to Go, but as our application continues to grow it should start to become clearer why we're structuring things the way we are.

If you take a step back, you might be able to see a few benefits emerging:

- There’s a clean separation of concerns. Our database logic isn’t 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.

- We have total control over which database is used at runtime, just by using the command-line flag.

- *And finally*, the directory structure scales nicely if your project has multiple back ends. For example, if some of your data is held in Redis you could put all the models for it in a `pkg/models/redis` package.

Chapter 4.5.

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 `pkg/models/mysql/snippets.go` file and update it like so:

```
File: pkg/models/mysql/snippets.go

package mysql

...

type SnippetModel struct {
    DB *sql.DB
}

func (m *SnippetModel) Insert(title, content, expires string) (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
    // title, content and expiry values for the placeholder parameters. This
    // method returns a sql.Result object, 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 object 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` interface 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.

It's important to note that not all drivers and databases support these two 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 `createSnippet` handler like so:

File: cmd/web/handlers.go

```
package main

...

func (app *application) createSnippet(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.Header().Set("Allow", "POST")
        app.clientError(w, http.StatusMethodNotAllowed)
        return
    }

    // 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
    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, err)
        return
    }

    // Redirect the user to the relevant page for the snippet.
    http.Redirect(w, r, fmt.Sprintf("/snippet?id=%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 -X POST http://localhost:4000/snippet/create
HTTP/1.1 303 See Other
Location: /snippet?id=4
Date: Fri, 07 Sep 2018 15:04:57 GMT
Content-Length: 0

HTTP/1.1 200 OK
Date: Fri, 07 Sep 2018 15:04:57 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 `createSnippet` 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 as a query string parameter.

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     | 2019-09-07 11:46:10 |
|  2 | Over the wintry forest | 2019-09-07 11:46:10 |
|  3 | First autumn morning   | 2018-09-14 11:46:11 |
|  4 | O snail                | 2018-09-14 15:04:57 |
+----+------------------------+---------------------+
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, `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 an 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)", ...)
```

Chapter 4.6.

Single-record SQL Queries

The pattern for `SELECT`ing 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 `pkg/models/mysql/snippets.go` file and add the following code:

```
File: pkg/models/mysql/snippets.go
```

```go
package mysql

import (
    "database/sql"

    "alexedwards.net/snippetbox/pkg/models"
)

type SnippetModel struct {
    DB *sql.DB
}

...

func (m *SnippetModel) Get(id int) (*models.Snippet, error) {
    // Write the SQL statement we want to execute. Again, I've split it over tw
    // 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 pointer to a new zeroed Snippet struct.
    s := &models.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. If the query returns no rows, then
    // row.Scan() will return a sql.ErrNoRows error. We check for that and retu
    // our own models.ErrNoRecord error instead of a Snippet object.
    err := row.Scan(&s.ID, &s.Title, &s.Content, &s.Created, &s.Expires)
    if err == sql.ErrNoRows {
        return nil, models.ErrNoRecord
    } else if err != nil {
        return nil, err
    }

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

...
```

Aside: You might be wondering why we're returning the `models.ErrNoRecord` error instead of `sql.ErrNoRows` directly. The reason is to help encapsulate the model completely, so that our application isn't concerned with the underlying datastore or reliant on datastore-specific errors for its behavior.

Type Conversions

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`.

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.

Using the Model in Our Handlers

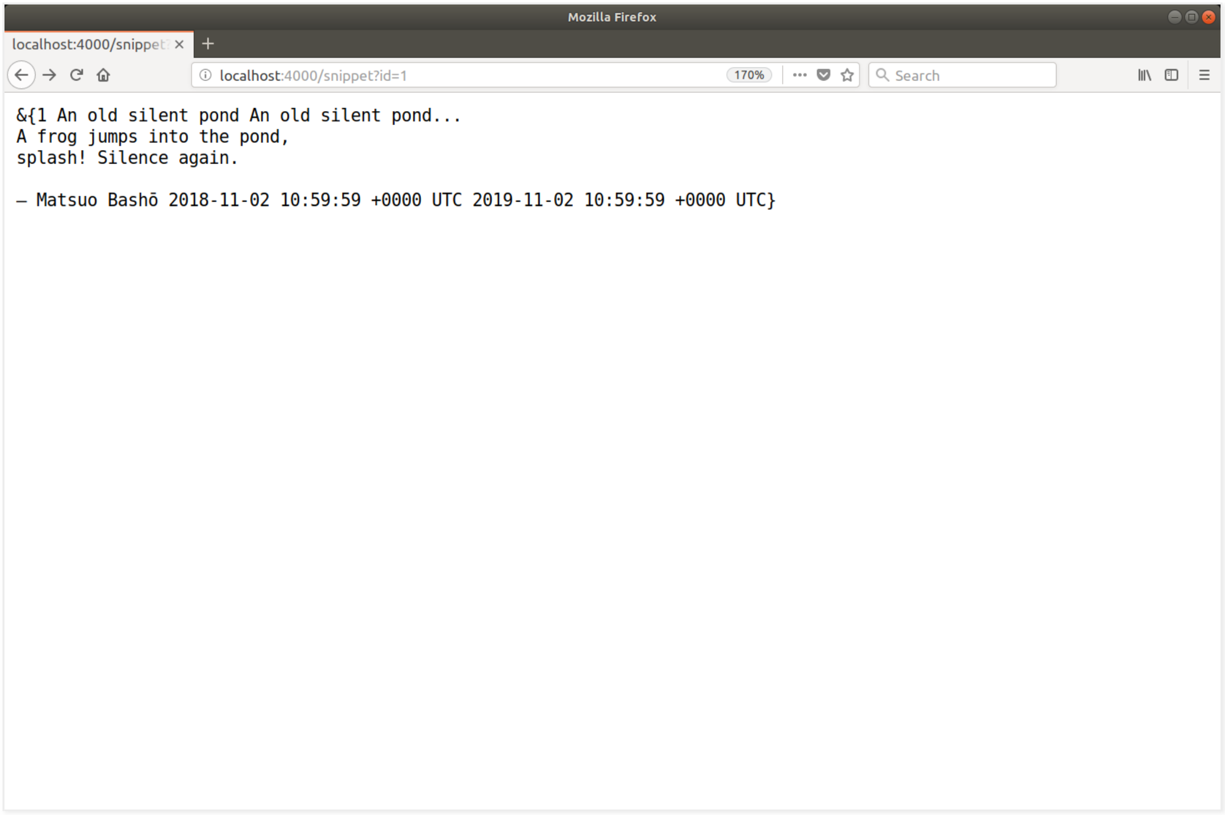
Alright, let's put the `SnippetModel.Get()` method into action.

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

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

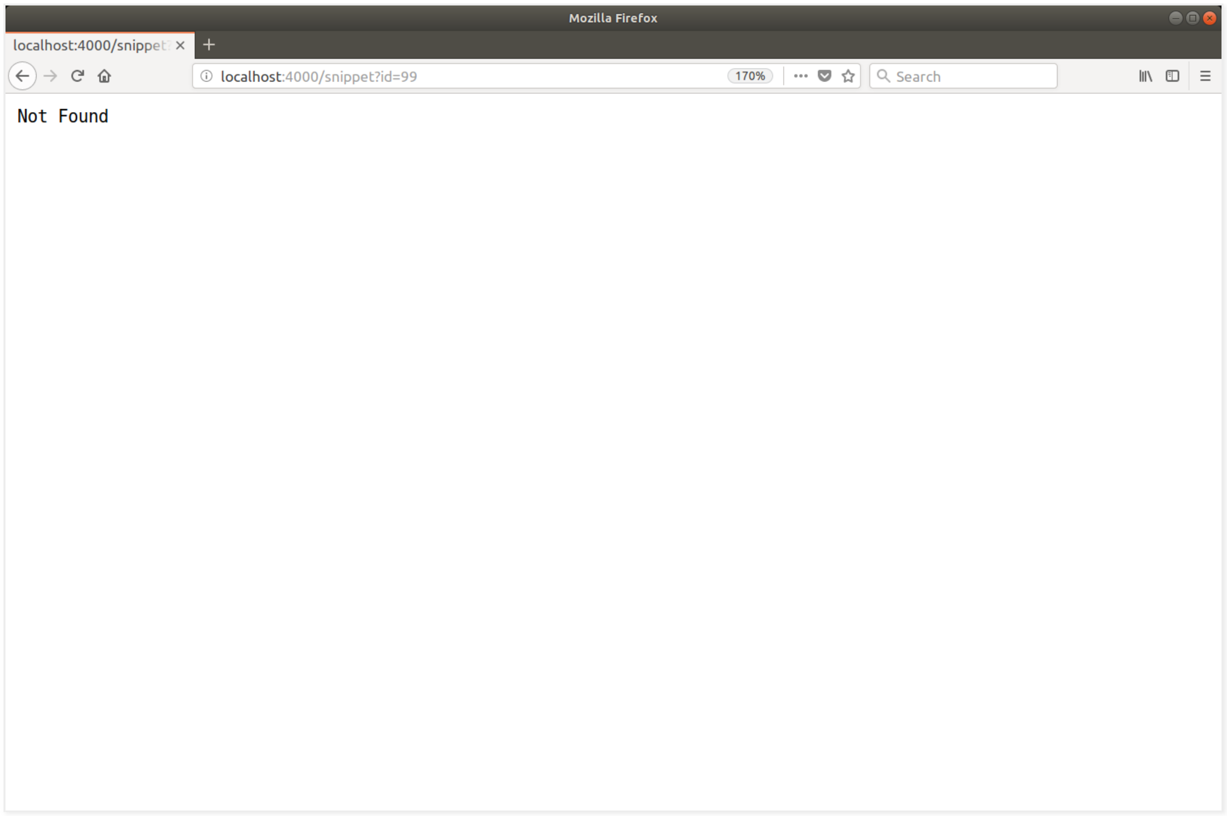
```go
package main

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

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

...

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    // Use the SnippetModel object'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.
    s, err := app.snippets.Get(id)
    if err == models.ErrNoRecord {
        app.notFound(w)
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }

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

...
```

Let’s give this a try. Go to your web browser and visit `http://localhost:4000/snippet?id=1`. You should see a HTTP response which looks similar to this:



```
&{1 An old silent pond An old silent pond...
A frog jumps into the pond,
splash! Silence again.

– Matsuo Bashō 2018-11-02 10:59:59 +0000 UTC 2019-11-02 10:59:59 +0000 UTC}
```

You might also want to try making some requests for other snippets which are expired or don't yet exist (like `id=99`) to verify that they return a `404 Not Found` response:



```
Not Found
```

Additional Information

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 your code.

In practice, you can shorten the code (or at least, the number of lines!) 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) (*models.Snippet, error) {
    s := &models.Snippet{}
    err := m.DB.QueryRow("SELECT ...", id).Scan(&s.ID, &s.Title, &s.Content, &s
    if err == sql.ErrNoRows {
        return nil, models.ErrNoRecord
    } else if err != nil {
        return nil, err
    }

    return s, nil
}
```

Chapter 4.7.

Multiple-record SQL Queries

Finally let's look at the pattern for executing SQL statements which return multiple rows. I'll demonstrate by updating the `SnippetModel.Latest()` method to return the *most recently created ten 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 created DESC LIMIT 10
```

Open up the `pkg/models/mysql/snippets.go` file and add the following code:

```
File: pkg/models/mysql/snippets.go
```

```
package mysql

import (
    "database/sql"

    "alexedwards.net/snippetbox/pkg/models"
)

type SnippetModel struct {
    DB *sql.DB
}
```

```
...

func (m *SnippetModel) Latest() ([]*models.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 created 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 o
    // 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 models.Snippets objects.
    snippets := []*models.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.
        s := &models.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 th
        // 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 an
    // 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 here. 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.

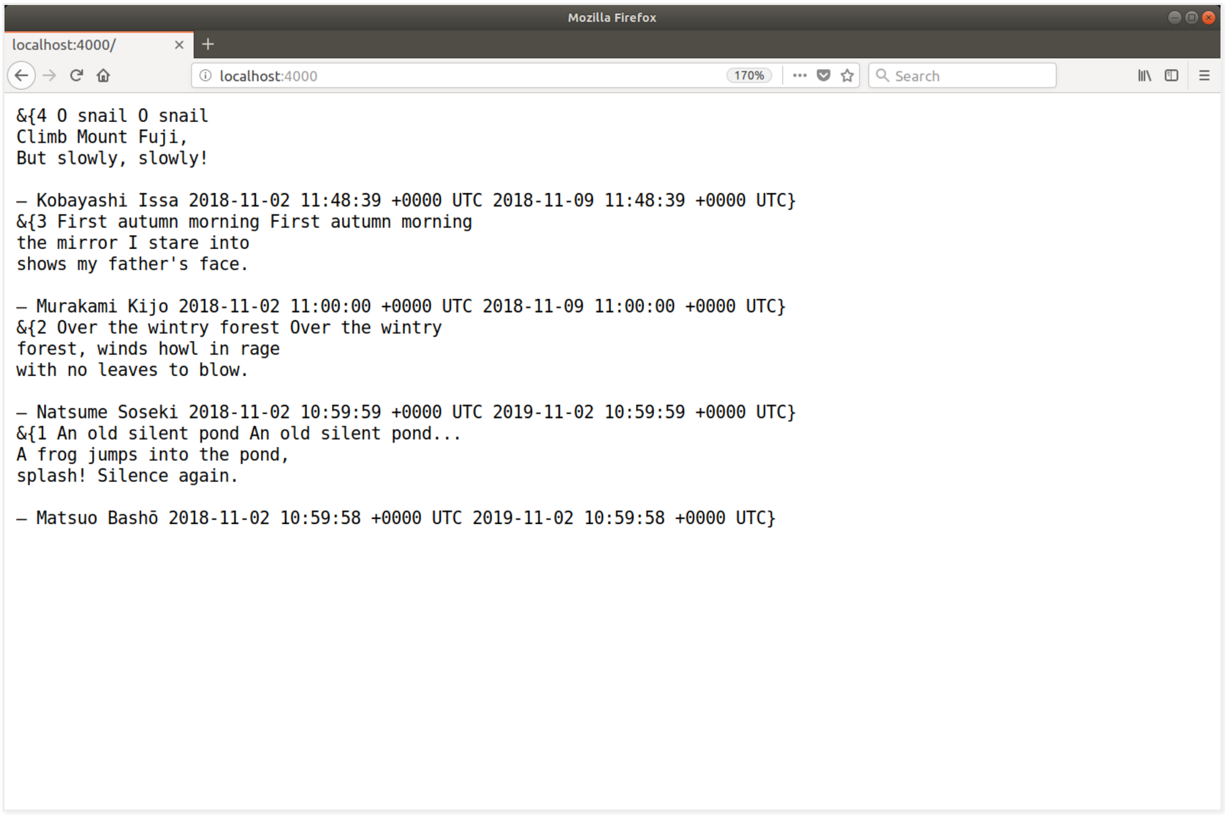
Using the Model in Our Handlers

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

```go
package main

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

    "alexedwards.net/snippetbox/pkg/models"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    for _, snippet := range s {
        fmt.Fprintf(w, "%v\n", snippet)
    }

    // files := []string{
    //     "./ui/html/home.page.tmpl",
    //     "./ui/html/base.layout.tmpl",
    //     "./ui/html/footer.partial.tmpl",
    // }

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


    // err = ts.Execute(w, nil)
    // if err != nil {
    //     app.serverError(w, err)
    // }
}

...
```

If you run the application now and visit `http://localhost:4000` in your browser you should get a response similar to this:



```
&{4 0 snail 0 snail
Climb Mount Fuji,
But slowly, slowly!

– Kobayashi Issa 2018-11-02 11:48:39 +0000 UTC 2018-11-09 11:48:39 +0000 UTC}
&{3 First autumn morning First autumn morning
the mirror I stare into
shows my father's face.

– Murakami Kijo 2018-11-02 11:00:00 +0000 UTC 2018-11-09 11:00:00 +0000 UTC}
&{2 Over the wintry forest Over the wintry
forest, winds howl in rage
with no leaves to blow.

– Natsume Soseki 2018-11-02 10:59:59 +0000 UTC 2019-11-02 10:59:59 +0000 UTC}
&{1 An old silent pond An old silent pond...
A frog jumps into the pond,
splash! Silence again.

– Matsuo Bashō 2018-11-02 10:59:58 +0000 UTC 2019-11-02 10:59:58 +0000 UTC}
```

Chapter 4.8.

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.

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.

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 an 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 are 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
    }

    // 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 {
        // If there is any error, we call the tx.Rollback() method on the
        // transaction. This will abort the transaction and no changes will be
        // made to the database.
        tx.Rollback()
        return err
    }

    // Carry out another transaction in exactly the same way.
    _, err = tx.Exec("UPDATE ...")
    if err != nil {
        tx.Rollback()
        return err
    }

    // If there are no errors, the statements in the transaction can be committ
    // to the database with the tx.Commit() method. It's really important to AL
    // call either Rollback() or Commit() before your function returns. If you
    // don't the connection will stay open and not be returned to the connectio
    // pool. This can lead to hitting your maximum connection limit/running out
    // resources.
    err = tx.Commit()
    return err
}
```

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.

Managing Connections

The sql.DB connection pool is made up of connections which are either *idle* or *open* (in use). By default, there is no limit on the maximum number of open connections at one time, but the default maximum number of idle connections in the pool is 2. You can change these defaults with the SetMaxOpenConns() and SetMaxIdleConns() methods. For example:

```
db, err := sql.Open("mysql", *dsn)
if err != nil {
    log.Fatal(err)
}

// Set the maximum number of concurrently open connections. Setting this to
// less than or equal to 0 will mean there is no maximum limit. If the maximum
// number of open connections is reached and a new connection is needed, Go wil
// wait until one of the connections is freed and becomes idle. From a
// user perspective, this means their HTTP request will hang until a connection
// is freed.
db.SetMaxOpenConns(95)

// Set the maximum number of idle connections in the pool. Setting this
// to less than or equal to 0 will mean that no idle connections are retained.
db.SetMaxIdleConns(5)
```

Using these methods comes with a caveat: *your database itself probably has a hard limit on the maximum number of connections*.

For example, the default limit for MySQL is 151. So leaving `SetMaxOpenConns()` totally unlimited or setting the *total* maximum of open and idle connections to greater than 151 may result in your database returning a `"too many connections"` error under high load. To prevent this error from happening, you'd need to set the *total* maximum of open and idle connections to comfortably below 151.

But in turn that creates another problem. When the `SetMaxOpenConns()` limit is reached, any new database tasks that your application needs to execute will be forced to wait until a connection becomes free.

For some applications that behavior might be fine, but in a web application it's arguably better to immediately log the `"too many connections"` error message and send a `500 Internal Server Error` to the user, rather than having their HTTP request hang and potentially timeout while waiting for a free connection.

That's why I haven't used the `SetMaxOpenConns()` and `SetMaxIdleConns()` methods in our application, and left the behavior of `sql.DB` as the default settings.

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:

```go
// We need somewhere to store the prepared statement for the lifetime of our
// web application. A neat way is to embed it 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 object, alongside the connection pool.
    return &ExampleModel{db, insertStmt}, nil


// Any methods implemented against the ExampleModel object will have access to
// the prepared statement.
func (m *ExampleModel) Insert(args...) error {
    // Notice how we 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 {
        errorLog.Fatal(err)
    }
    defer db.Close()

    // Create a new ExampleModel object, which includes the prepared statement.
    exampleModel, err := NewExampleModel(db)
    if err != nil {
       errorLog.Fatal(err)
    }

    // 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 amount 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 too is 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.

Chapter 5.

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:

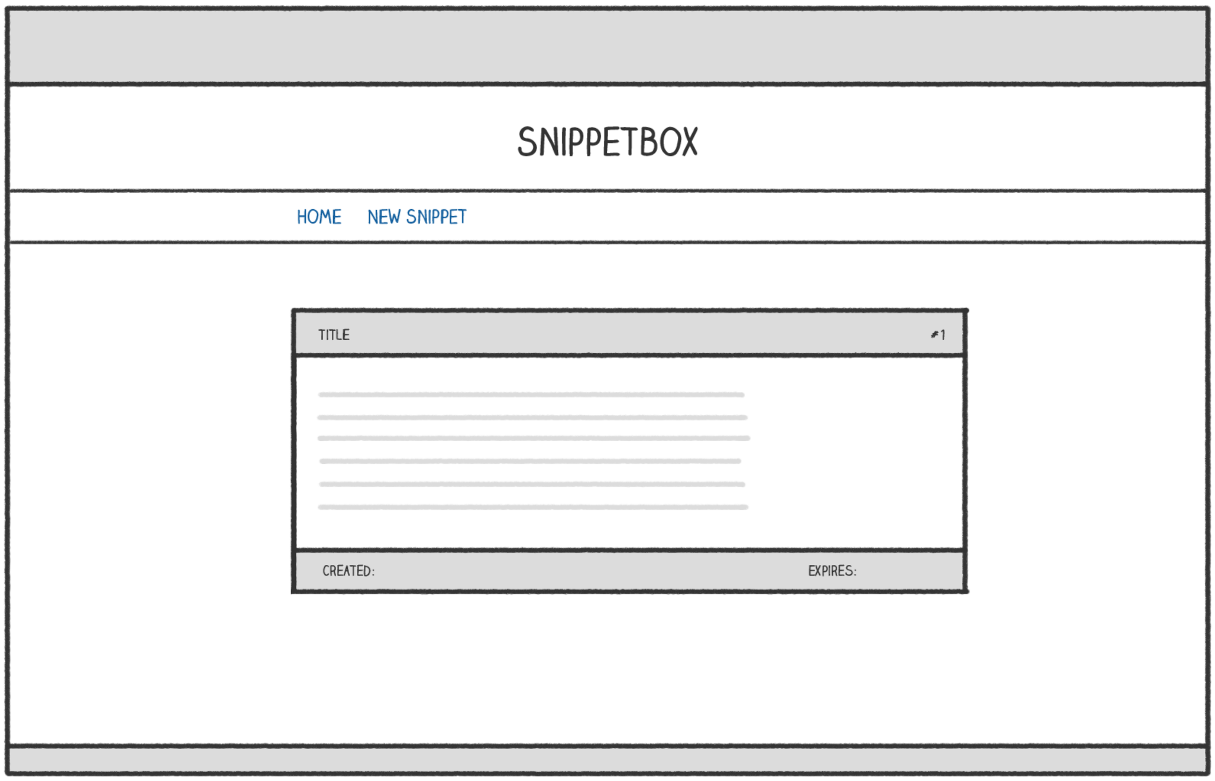
- Pass dynamic data to your HTML templates in a simple, scalable and type-safe way.
- Use the various actions and functions in Go's `html/template` package to control the display of dynamic data.
- Create a template cache so that your templates aren't being read from disk for each HTTP request.
- Gracefully 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.

Chapter 5.1.

Displaying Dynamic Data

Currently our `showSnippet` hander function fetches a `models.Snippet` object from the database and then dumps the contents out in a plain-text HTTP response.

In this section we'll update this so that the data is displayed in a proper HTML webpage which looks a bit like this:



Let's start in the `showSnippet` handler and add some code to render a new `show.page.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
```

```go
package main

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

    "alexedwards.net/snippetbox/pkg/models"
)

...

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err == models.ErrNoRecord {
        app.notFound(w)
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }

    // Initialize a slice containing the paths to the show.page.tmpl file,
    // plus the base layout and footer partial that we made earlier.
    files := []string{
        "./ui/html/show.page.tmpl",
        "./ui/html/base.layout.tmpl",
        "./ui/html/footer.partial.tmpl",
    }

    // Parse the template files...
    ts, err := template.ParseFiles(files...)
    if err != nil {
        app.serverError(w, 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.Execute(w, s)
    if err != nil {
        app.serverError(w, err)
    }
}

...
```

Next up we need to create the `show.page.tmpl` file containing the HTML markup for the page. But before we do, there's a little theory that I need to explain…

Within your HTML templates, any dynamic data that you pass in is represented 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 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.

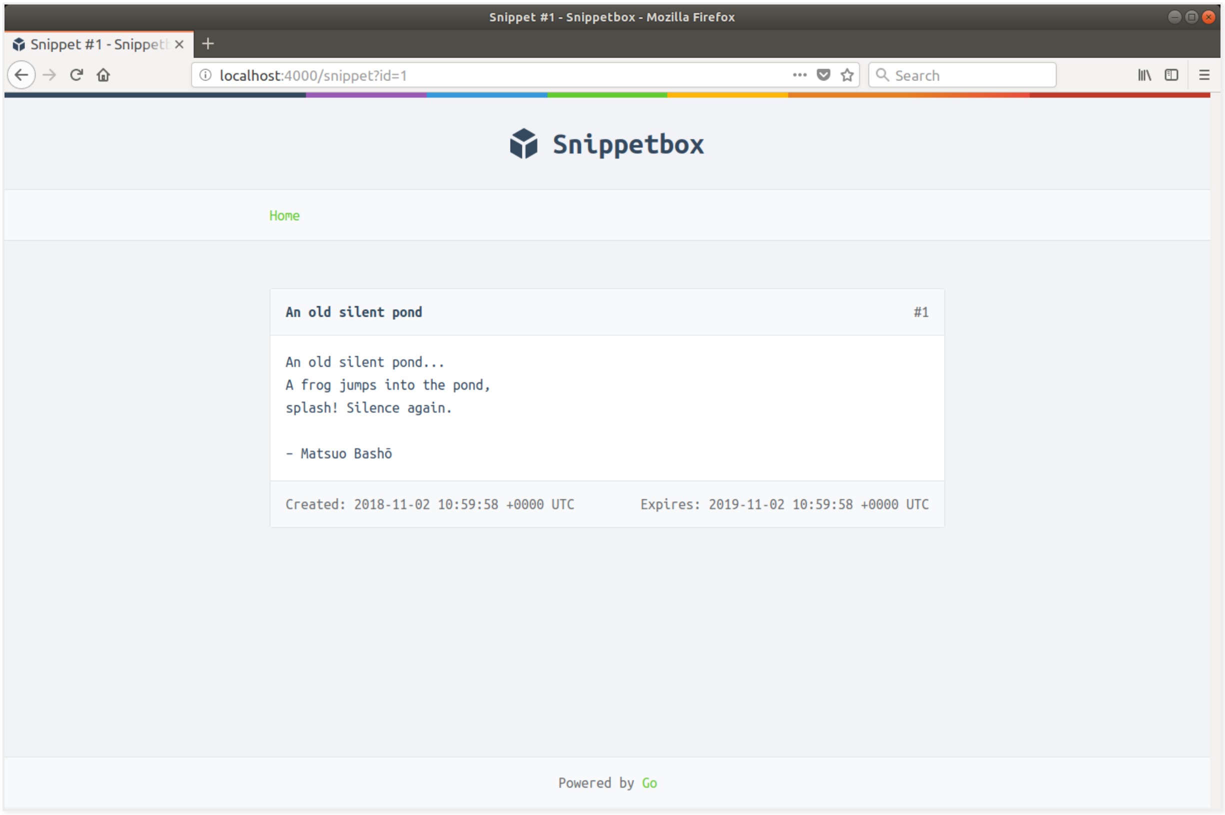
I'll demonstrate. Create a new file at `ui/html/show.page.tmpl` and add the following markup:

```
$ touch ui/html/show.page.tmpl
```

```
File: ui/html/show.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Snippet #{{.ID}}{{end}}

{{define "body"}}
<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?id=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.


Snippet #1 - Snippetbox - Mozilla Firefox
Snippet #1 - Snippetb
localhost:4000/snippet?id=1
Search
Snippetbox
Home

```
An old silent pond                                              #1

An old silent pond...
A frog jumps into the pond,
splash! Silence again.

- Matsuo Bashō

Created: 2018-11-02 10:59:58 +0000 UTC      Expires: 2019-11-02 10:59:58 +0000 UTC
```

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 acheive 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
```

```go
package main

import "alexedwards.net/snippetbox/pkg/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 `showSnippet` handler to use this new struct when executing our templates:

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

```go
package main

...

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err == models.ErrNoRecord {
```

```
    if err == models.ErrNoRecord {
        app.notFound(w)
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }

    // Create an instance of a templateData struct holding the snippet data.
    data := &templateData{Snippet: s}

    files := []string{
        "./ui/html/show.page.tmpl",
        "./ui/html/base.layout.tmpl",
        "./ui/html/footer.partial.tmpl",
    }

    ts, err := template.ParseFiles(files...)

    if err != nil {
        app.serverError(w, err)
        return
    }

    // Pass in the templateData struct when executing the template.
    err = ts.Execute(w, data)
    if err != nil {
        app.serverError(w, 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/show.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "body"}}
<div class='snippet'>
    <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?id=1` again. You should see the same page rendered in your browser as before.

Additional Information

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 "base" .}}
{{template "body" .}}
{{template "footer" .}}
{{block "sidebar" .}}{{end}}
```

As a 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 object that you're yielding has methods defined against it, you can call them (so long as they are exported and they return only a single value — or a single value and an error).

For example, if `.Snippet.Created` has the underlying type `time.Time` (which it does) 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.

Chapter 5.2.

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's three more which you can use to control the display of dynamic data — `{{if}}`, `{{with}}` and `{{range}}`.

Action	Description
`{{if .Foo}} C1 {{else}} C2 {{end}}`	If `.Foo` is not empty then render the content C1, otherwise render the content C2.
`{{with .Foo}} C1 {{else}} C2 {{end}}`	If `.Foo` is not empty, then set dot to the value of `.Foo` and render the content C1, otherwise render the content C2.
`{{range .Foo}} C1 {{else}} C2 {{end}}`	If the length of `.Foo` is greater than zero then loop over each element, setting dot to the value of each element and rendering the content C1. If the length of `.Foo` is zero then render the content C2. The underlying type of `.Foo` 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
`{{eq .Foo .Bar}}`	Yields true if `.Foo` is equal to `.Bar`
`{{ne .Foo .Bar}}`	Yields true if `.Foo` is not equal to `.Bar`
`{{not .Foo}}`	Yields the boolean negation of `.Foo`
`{{or .Foo .Bar}}`	Yields `.Foo` if `.Foo` is not empty; otherwise yields `.Bar`
`{{index .Foo i}}`	Yields the value of `.Foo` at index `i`. The underlying type of `.Foo` must be a map, slice or array.

Function	Description
`{{printf "%s-%s" .Foo .Bar}}`	Yields a formatted string containing the `.Foo` and `.Bar` values. Works in the same way as fmt.Sprintf().
`{{len .Foo}}`	Yields the length of `.Foo` as an integer.
`{{$bar := len .Foo}}`	Assign the length of `.Foo` to the template variable `$bar`

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 `show.page.tmpl` file that we created in the previous chapter. Go ahead and update it like so:

```
File: ui/html/show.page.tmpl
```

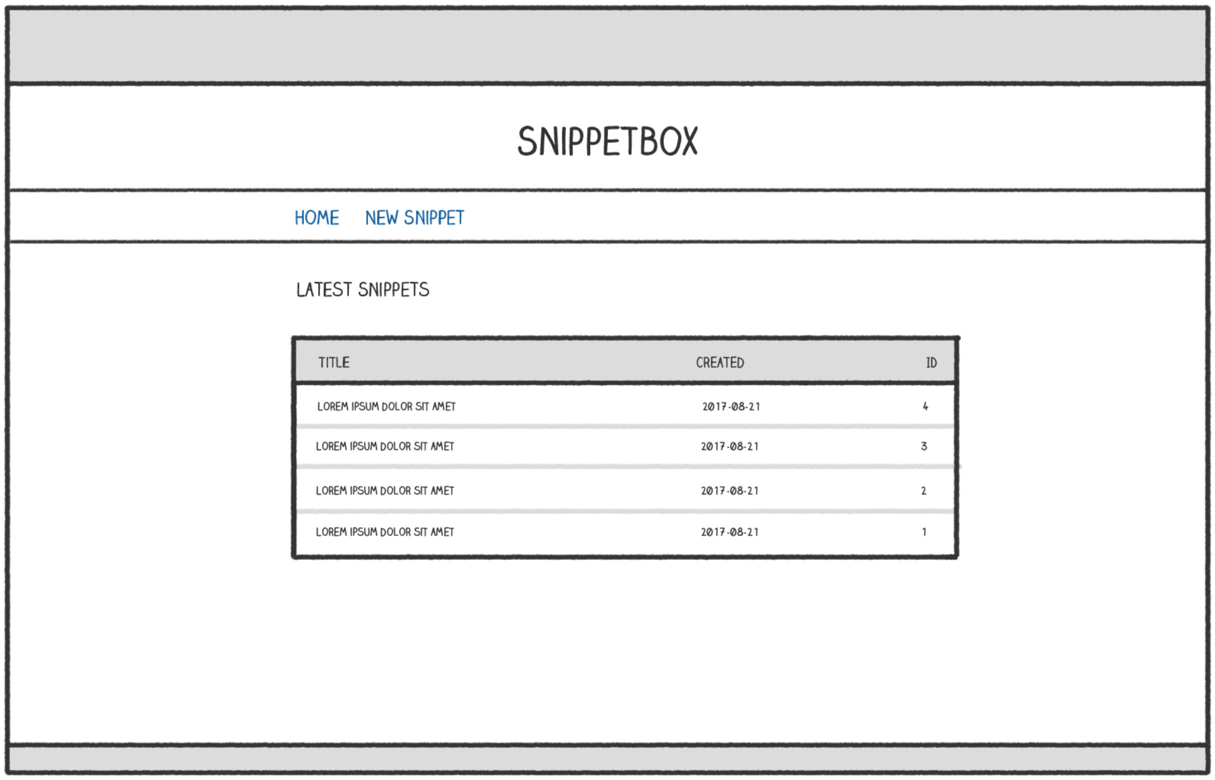
```
{{template "base" .}}

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "body"}}
    {{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:



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
```

```go
package main

import "alexedwards.net/snippetbox/pkg/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.page.tmpl` template:

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

```
package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    // Create an instance of a templateData struct holding the slice of
    // snippets.
    data := &templateData{Snippets: s}

    files := []string{
        "./ui/html/home.page.tmpl",
        "./ui/html/base.layout.tmpl",
        "./ui/html/footer.partial.tmpl",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        app.serverError(w, err)
        return
    }
    // Pass in the templateData struct when executing the template.
    err = ts.Execute(w, data)
    if err != nil {
        app.serverError(w, err)
    }
}

...
```

Now let’s head over to the `ui/html/home.page.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.

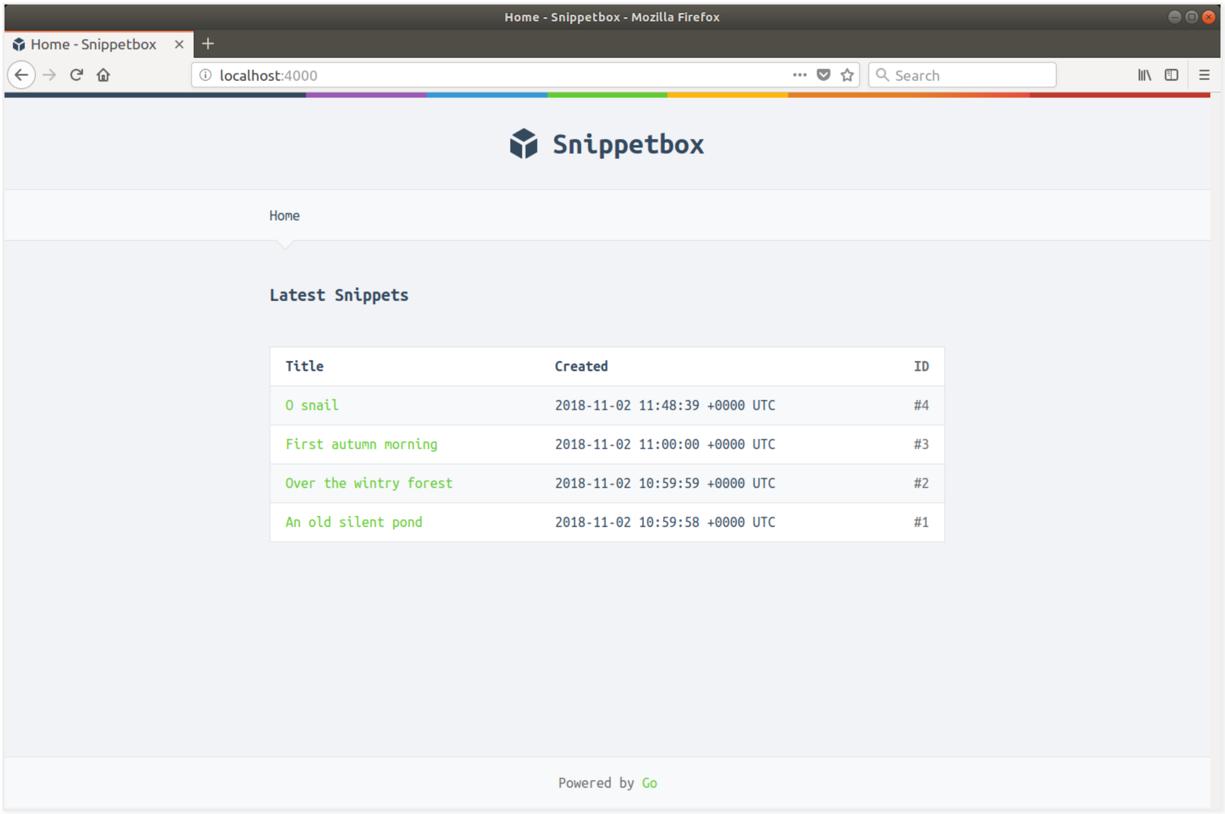
Here’s the markup:

```
File: ui/html/home.page.tmpl
```

```
{{template "base" .}}

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

{{define "body"}}
    <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?id={{.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:



Latest Snippets

Title	Created	ID
O snail	2018-11-02 11:48:39 +0000 UTC	#4
First autumn morning	2018-11-02 11:00:00 +0000 UTC	#3
Over the wintry forest	2018-11-02 10:59:59 +0000 UTC	#2
An old silent pond	2018-11-02 10:59:58 +0000 UTC	#1

Powered by Go

Chapter 5.3.

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're two main issues at the moment:

1. Each and every time we render a web page, our application must read the template files from disk. This could be speeded up by *caching* the templates in memory.

2. There's duplicated code in the `home` and `showSnippet` handlers, which we could reduce 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 templates. Open your `cmd/web/templates.go` file and add the following code:

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

```go
package main

import (
    "html/template" // New import
    "path/filepath" // New import

    "alexedwards.net/snippetbox/pkg/models"
)
```

```
...

func newTemplateCache(dir string) (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 with
    // the extension '.page.tmpl'. This essentially gives us a slice of all the
    // 'page' templates for the application.
    pages, err := filepath.Glob(filepath.Join(dir, "*.page.tmpl"))
    if err != nil {
        return nil, err
    }

    // Loop through the pages one-by-one.
    for _, page := range pages {
        // Extract the file name (like 'home.page.tmpl') from the full file pat
        // and assign it to the name variable.
        name := filepath.Base(page)

        // Parse the page template file in to a template set.
        ts, err := template.ParseFiles(page)
        if err != nil {
            return nil, err
        }

        // Use the ParseGlob method to add any 'layout' templates to the
        // template set (in our case, it's just the 'base' layout at the
        // moment).
        ts, err = ts.ParseGlob(filepath.Join(dir, "*.layout.tmpl"))
        if err != nil {
            return nil, err
        }

        // Use the ParseGlob method to add any 'partial' templates to the
        // template set (in our case, it's just the 'footer' partial at the
        // moment).
        ts, err = ts.ParseGlob(filepath.Join(dir, "*.partial.tmpl"))
        if err != nil {
            return nil, err
        }

        // Add the template set to the cache, using the name of the page
        // (like 'home.page.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"
    "net/http"
    "os"

    "alexedwards.net/snippetbox/pkg/models/mysql"

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

...
```

```
// Add a templateCache field to the application struct.
type application struct {
    errorLog      *log.Logger
    infoLog       *log.Logger
    snippets      *mysql.SnippetModel
    templateCache map[string]*template.Template
}

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

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    db, err := openDB(*dsn)
    if err != nil {
        errorLog.Fatal(err)
    }
    defer db.Close()

    // Initialize a new template cache...
    templateCache, err := newTemplateCache("./ui/html/")
    if err != nil {
        errorLog.Fatal(err)
    }

    // And add it to the application dependencies.
    app := &application{
        errorLog:      errorLog,
        infoLog:       infoLog,
        snippets:      &mysql.SnippetModel{DB: db},
        templateCache: templateCache,
    }

    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  app.routes(),
    }

    infoLog.Printf("Starting server on %s", *addr)
    err = srv.ListenAndServe()
    errorLog.Fatal(err)
}

...
```

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 method:

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

package main

...

func (app *application) render(w http.ResponseWriter, r *http.Request, name str
    // Retrieve the appropriate template set from the cache based on the page n
    // (like 'home.page.tmpl'). If no entry exists in the cache with the
    // provided name, call the serverError helper method that we made earlier.
    ts, ok := app.templateCache[name]
    if !ok {
        app.serverError(w, fmt.Errorf("The template %s does not exist", name))
        return
    }
    // Execute the template set, passing in any dynamic data.
    err := ts.Execute(w, td)
    if err != nil {
        app.serverError(w, err)
    }
}
```

At this point you might be wondering why the signature for the `render()` method includes a `*http.Request` parameter… even though it's not used anywhere. This is simply to future-proof the method signature, because later in the book we *will* need access to this.

With that complete, we now get to see the pay-off from these changes and can dramatically simplify the code in our handlers:

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

```
package main

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

    "alexedwards.net/snippetbox/pkg/models"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    // Use the new render helper.
    app.render(w, r, "home.page.tmpl", &templateData{
        Snippets: s,
    })
}

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err == models.ErrNoRecord {
        app.notFound(w)
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }

    // Use the new render helper.
    app.render(w, r, "show.page.tmpl", &templateData{
        Snippet: s,
    })
}

...
```

Chapter 5.4.

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 `show.page.tmpl` template and see what happens:

```
File: ui/html/show.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "body"}}
    {{with .Snippet}}
    <div class='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>
    </div>
    {{end}}
{{end}}
```

Note: In the 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/*
INFO    2018/09/09 11:50:25 Starting server on :4000
```

But if you use curl to make a request to `http://localhost:4000/snippet?id=1` you'll get a response which looks a bit like this.

```
$ curl -i http://localhost:4000/snippet?id=1
HTTP/1.1 200 OK
Date: Sun, 09 Sep 2018 09:50:27 GMT
Content-Length: 615
Content-Type: text/html; charset=utf-8

&lt;!doctype html&gt;
&lt;html lang='en'&gt;
    &lt;head&gt;
        &lt;meta charset='utf-8'&gt;
        &lt;title&gt;Snippet #1 - Snippetbox&lt;/title&gt;

        &lt;link rel='stylesheet' href='/static/css/main.css'&gt;
        &lt;link rel='shortcut icon' href='/static/img/favicon.ico' type='image
        &lt;link rel='stylesheet' href='https://fonts.googleapis.com/css?family
    &lt;/head&gt;
    &lt;body&gt;
```

```
        &lt;header&gt;
            &lt;h1&gt;&lt;a href='/'&gt;Snippetbox&lt;/a&gt;&lt;/h1&gt;
        &lt;/header&gt;
        &lt;nav&gt;
            &lt;a href='/'&gt;Home&lt;/a&gt;
        &lt;/nav&gt;
        &lt;section&gt;


    &lt;div class='snippet'&gt;
        &lt;div class='metadata'&gt;
            &lt;strong&gt;An old silent pond&lt;/strong&gt;
            &lt;span&gt;#1&lt;/span&gt;
        &lt;/div&gt;
        Internal Server Error
```

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
```

```go
package main

import (
    "bytes" // New import
    "fmt"
    "net/http"
    "runtime/debug"
)

...

func (app *application) render(w http.ResponseWriter, r *http.Request, name str
    ts, ok := app.templateCache[name]
    if !ok {
        app.serverError(w, fmt.Errorf("The template %s does not exist", name))
        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 an
    // return.
    err := ts.Execute(buf, td)
    if err != nil {
        app.serverError(w, err)
        return
    }

    // Write the contents of the buffer to the http.ResponseWriter. Again, 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?id=1
HTTP/1.1 500 Internal Server Error
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Sun, 09 Sep 2018 09:55:13 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 `show.page.tmpl` file and remove the deliberate error:

```
File: ui/html/show.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "body"}}
    {{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}}
```

Chapter 5.5.

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
```

```go
package main

...

// Add a CurrentYear field to the templateData struct.
type templateData struct {
    CurrentYear int
    Snippet     *models.Snippet
    Snippets    []*models.Snippet
}

...
```

Then the next step is to create a new `addDefaultData()` helper method to our application, which will inject the current year into an instance of a `templateData` struct.

We can then call this from our `render()` helper to add the default data automatically for each page.

I'll demonstrate:

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

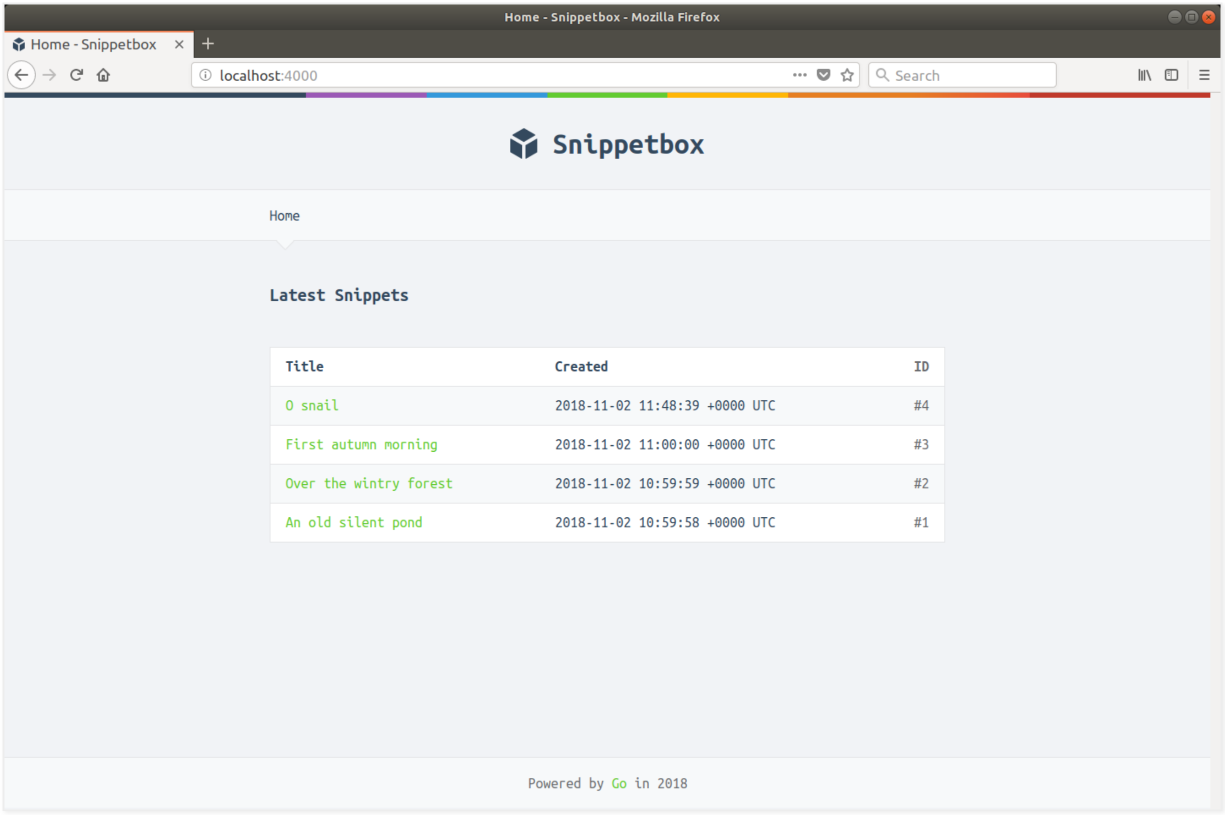
```go
package main

import (
    "bytes"
    "fmt"
    "net/http"
    "runtime/debug"
    "time" // New import
)

...

// Create an addDefaultData helper. This takes a pointer to a templateData
// struct, adds the current year to the CurrentYear field, and then returns
// the pointer. Again, we're not using the *http.Request parameter at the
// moment, but we will do later in the book.
func (app *application) addDefaultData(td *templateData, r *http.Request) *temp
    if td == nil {
        td = &templateData{}
    }
    td.CurrentYear = time.Now().Year()
    return td
}

func (app *application) render(w http.ResponseWriter, r *http.Request, name str
    ts, ok := app.templateCache[name]
    if !ok {
        app.serverError(w, fmt.Errorf("The template %s does not exist", name))
        return
    }

    buf := new(bytes.Buffer)

    // Execute the template set, passing the dynamic data with the current
    // year injected.
    err := ts.Execute(buf, app.addDefaultData(td, r))
    if err != nil {
        app.serverError(w, err)
        return
    }

    buf.WriteTo(w)
}
```

Now we just need to update the `ui/html/footer.partial.tmpl` file to display the year, like so:

```
File: ui/html/footer.partial.tmpl
```

```
{{define "footer"}}
<footer>Powered by <a href='https://golang.org/'>Go</a>  in {{.CurrentYear}}</f
{{end}}
```

Restart the application and visit the home page at `http://localhost:4000`. You should see the current year in the footer like this:



Snippetbox

Home

Latest Snippets

Title	Created	ID
O snail	2018-11-02 11:48:39 +0000 UTC	#4
First autumn morning	2018-11-02 11:00:00 +0000 UTC	#3
Over the wintry forest	2018-11-02 10:59:59 +0000 UTC	#2
An old silent pond	2018-11-02 10:59:58 +0000 UTC	#1

Powered by Go in 2018

Chapter 5.6.

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 `02 Jan 2019 at 15:04`, instead of outputting dates in the default format of `2019-01-02 15:04:00 +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
```

```go
package main

import (
    "html/template"
    "path/filepath"
    "time" // New import

    "alexedwards.net/snippetbox/pkg/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
// essentially a string-keyed map which acts as a lookup between the names of o
// custom template functions and the functions themselves.
var functions = template.FuncMap{
    "humanDate": humanDate,
}

func newTemplateCache(dir string) (map[string]*template.Template, error) {
    cache := map[string]*template.Template{}

    pages, err := filepath.Glob(filepath.Join(dir, "*.page.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
        // call the ParseFiles() method. This means we have to use template.New
        // 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(page)
        if err != nil {
            return nil, err
        }

        ts, err = ts.ParseGlob(filepath.Join(dir, "*.layout.tmpl"))
        if err != nil {
            return nil, err
        }

        ts, err = ts.ParseGlob(filepath.Join(dir, "*.partial.tmpl"))
        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/home.page.tmpl
```

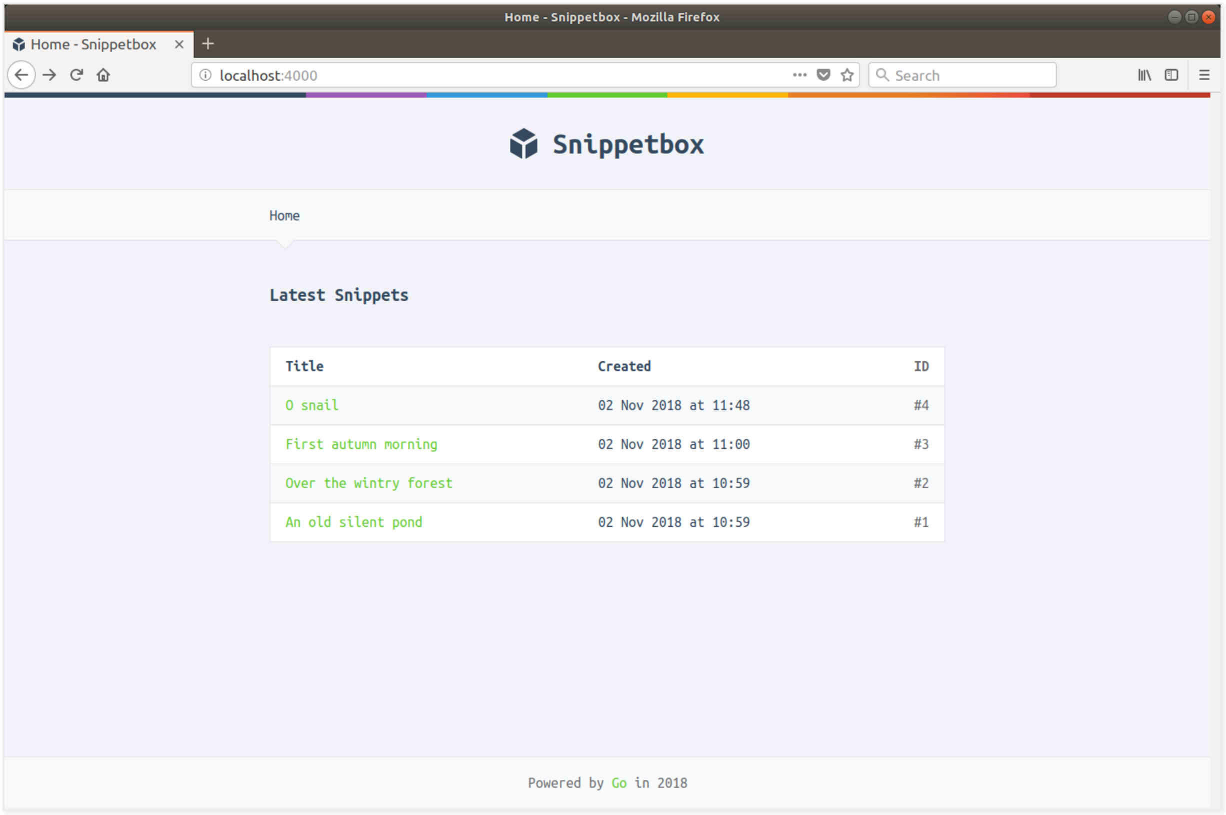
```
{{template "base" .}}

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

{{define "body"}}
    <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?id={{.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/show.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "body"}}
    {{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}}
```

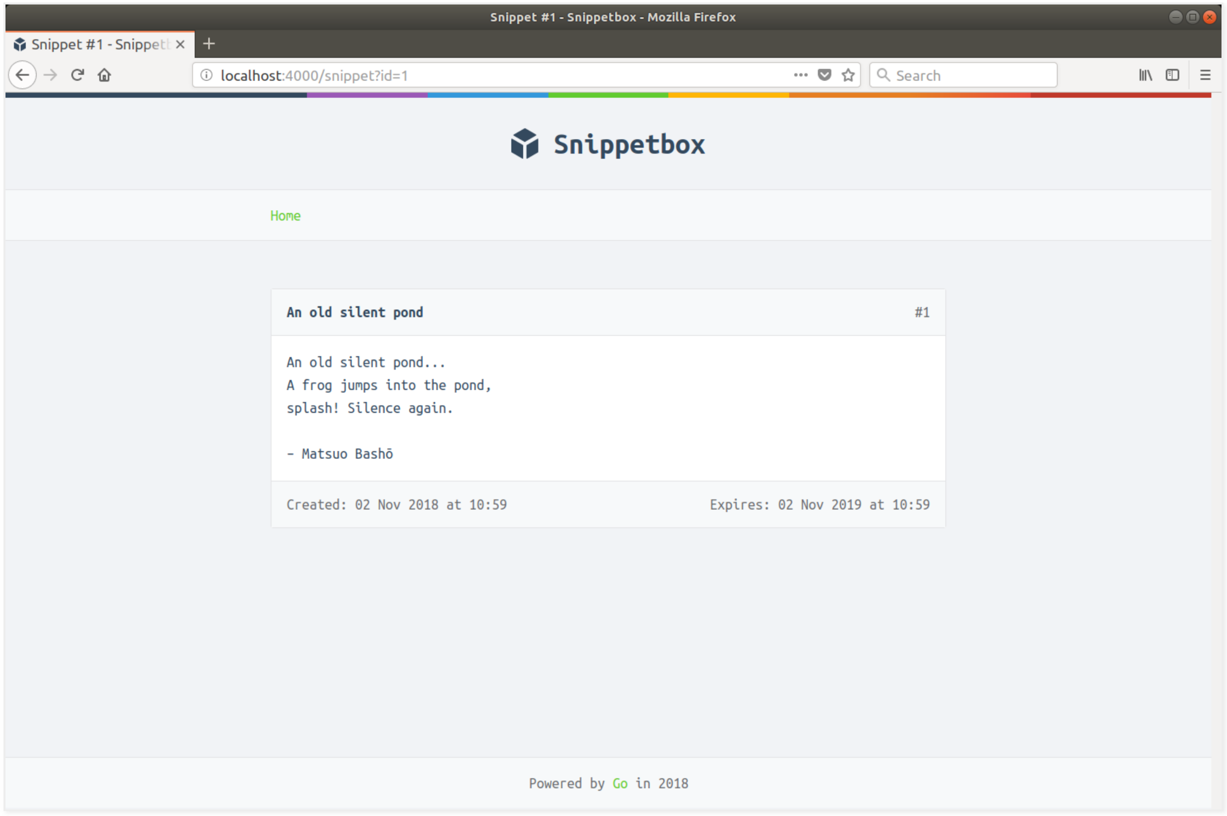
Once that's done restart the application. If you visit `http://localhost:4000` and `http://localhost:4000/snippet?id=1` in your browser you should see the new, nicely formatted, dates being used.



Latest Snippets

Title	Created	ID
O snail	02 Nov 2018 at 11:48	#4
First autumn morning	02 Nov 2018 at 11:00	#3
Over the wintry forest	02 Nov 2018 at 10:59	#2
An old silent pond	02 Nov 2018 at 10:59	#1

Powered by Go in 2018



Snippet #1 - Snippetbox - Mozilla Firefox
localhost:4000/snippet?id=1
Snippetbox
Home

```
An old silent pond                                          #1

An old silent pond...
A frog jumps into the pond,
splash! Silence again.

- Matsuo Bashō

Created: 02 Nov 2018 at 10:59          Expires: 02 Nov 2019 at 10:59

Powered by Go in 2018
```

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 pipelining outputs from one command to another in Unix terminals. We could re-write 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>
```

Chapter 6.

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 useful security 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.

Chapter 6.1.

How Middleware Works

Earlier on 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 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 the 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.
- 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 it's `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.
- 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 that executes some logic, and then calls the next handler.

Simplifying the Middleware

A tweak to this pattern is to use an anonymous function to rewrite the `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 middleware 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 specific routes.

```
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.

Chapter 6.2.

Setting Security Headers

Let's put the pattern we learned in the previous chapter to use, and make our own middleware which automatically adds the following two HTTP headers to every response:

```
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
```

If you're not familiar with these headers, they essentially instruct the user's web browser to implement some additional security measures to help prevent XSS and Clickjacking attacks. It's good practice to include them unless you have a specific reason for not doing so.

Let's begin by creating a new `middleware.go` file, which we'll use to hold all the custom middleware that we write throughout this book.

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

Then open it up and add a `secureHeaders()` function using the pattern that we introduced in the previous chapter:

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

```go
package main

import (
    "net/http"

)

func secureHeaders(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("X-XSS-Protection", "1; mode=block")
        w.Header().Set("X-Frame-Options", "deny")

        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:

```
secureHeaders → servemux → application handler
```

To do this we'll need the `secureHeaders` middleware function to *wrap our servemux*. Let's update the `routes.go` file to do exactly that:

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

```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()
    mux.HandleFunc("/", app.home)
    mux.HandleFunc("/snippet", app.showSnippet)
    mux.HandleFunc("/snippet/create", app.createSnippet)

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

    // Pass the servemux as the 'next' parameter to the secureHeaders middlewar
    // Because secureHeaders is just a function, and the function returns a
    // http.Handler we don't need to do anything else.
    return secureHeaders(mux)
}
```

Go ahead and give this a try. Run the application then open a second terminal window and play around making some requests using curl. You should see that the two security headers are now included in every response.

```
$ curl -I http://localhost:4000/
HTTP/1.1 200 OK
X-Frame-Options: deny
X-Xss-Protection: 1; mode=block
Date: Wed, 19 Sep 2018 15:24:04 GMT
Content-Length: 1028
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:

```
secureHeaders → servemux → application handler → servemux → secureHeaders
```

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:

```go
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 for restricting access to certain parts of our application.

Chapter 6.3.

Request Logging

Let’s continue in the same vein and add some middleware to *log HTTP requests*. Specifically, we’re going to use the *information logger* that we created earlier to record the IP address of the user, and which URL and method are being requested.

Open your `middleware.go` file and create a `logRequest()` method using the standard middleware pattern, like so:

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

```go
package main

...

func (app *application) logRequest(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        app.infoLog.Printf("%s - %s %s %s", r.RemoteAddr, r.Proto, r.Method, r.

        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 information 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 ↔ secureHeaders ↔ servemux ↔ application handler
```

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

```go
package main

import "net/http"

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()
    mux.HandleFunc("/", app.home)

    mux.HandleFunc("/snippet", app.showSnippet)
    mux.HandleFunc("/snippet/create", app.createSnippet)

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

    // Wrap the existing chain with the logRequest middleware.
    return app.logRequest(secureHeaders(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/*
INFO    2018/09/19 20:04:54 Starting server on :4000
INFO    2018/09/19 20:05:49 [::1]:43564 - HTTP/1.1 GET /snippet?id=2
INFO    2018/09/19 20:05:49 [::1]:43564 - HTTP/1.1 GET /static/css/main.css
INFO    2018/09/19 20:05:49 [::1]:43566 - HTTP/1.1 GET /static/js/main.js
INFO    2018/09/19 20:05:50 [::1]:43566 - HTTP/1.1 GET /static/img/logo.png
INFO    2018/09/19 20:05:50 [::1]:43566 - HTTP/1.1 GET /static/img/favicon.ico
INFO    2018/09/19 20:05:55 [::1]:43566 - HTTP/1.1 GET /
```

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.

Chapter 6.4.

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 it's own goroutine).

Specifically, following a panic our server will log a stack trace to the server error log, 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
```

```go
package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
```

```
    if r.URL.Path != "/" {
        app.notFound(w)
        return
    }

    panic("oops! something went wrong") // Deliberate panic

    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return

    }

    app.render(w, r, "home.page.tmpl", &templateData{
        Snippets: s,
    })
}

...
```

Restart your application…

```
$ go run cmd/web/*
INFO    2018/09/20 13:18:41 Starting server on :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
```

```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, 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 is an `interface{}` 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 `interface{}` 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
```

```go
package main

import "net/http"

func (app *application) routes() http.Handler {
    mux := http.NewServeMux()
    mux.HandleFunc("/", app.home)
    mux.HandleFunc("/snippet", app.showSnippet)
    mux.HandleFunc("/snippet/create", app.createSnippet)

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

    // Wrap the existing chain with the recoverPanic middleware.
    return app.recoverPanic(app.logRequest(secureHeaders(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/*
INFO    2018/09/20 16:07:49 Starting server on :4000
```

```
$ curl -i http://localhost:4000
HTTP/1.1 500 Internal Server Error
Connection: close
```

```
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Xss-Protection: 1; mode=block
Date: Thu, 20 Sep 2018 14:08:25 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
```

```go
package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        app.notFound(w)
        return
    }


    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    app.render(w, r, "home.page.tmpl", &templateData{
        Snippets: s,
    })
}
```

```
...
```

Additional Information

Panic Recovery in Other Background Goroutines

It's important to realise 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 the 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:

```go
func 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 {
                log.Println(fmt.Errorf("%s\n%s", err, debug.Stack()))
            }
        }()
        }()

        doSomeBackgroundProcessing()
    }()

    w.Write([]byte("OK"))
}
```

Chapter 6.5.

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 like 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)
```

Let’s update our `routes.go` file to use the justinas/alice package as follows:

```
package main

import (
    "net/http"

    "github.com/justinas/alice" // New import
)

func (app *application) routes() http.Handler {
    // Create a middleware chain containing our 'standard' middleware
    // which will be used for every request our application receives.
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea

    mux := http.NewServeMux()
    mux.HandleFunc("/", app.home)
    mux.HandleFunc("/snippet", app.showSnippet)
    mux.HandleFunc("/snippet/create", app.createSnippet)

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

    // Return the 'standard' middleware chain followed by the servemux.
    return standardMiddleware.Then(mux)
}
```

Save the file, then go ahead and run the application…

```
$ go run cmd/web/*
go: finding github.com/justinas/alice latest
go: downloading github.com/justinas/alice v0.0.0-20171023064455-03f45bd4b7da
INFO    2018/09/20 16:43:49 Starting server on :4000
```

Because we have modules enabled for our project, Go is clever enough to recognize that our code is importing a new third-party package and has automatically downloaded justinas/alice for us.

If you open the `go.mod` file for your project, you'll see a new `require` statement relating to this package:

```
File: go.mod
```

```
module alexedwards.net/snippetbox

require (
    github.com/go-sql-driver/mysql v1.4.0 // indirect
    github.com/justinas/alice v0.0.0-20171023064455-03f45bd4b7da // indirect
)
```

There's a couple of things to point out about this:

- When you rely on the 'automatic' method to download packages — like we have done here — Go will always retrieve the latest version of the package. If you want a different specific version you should use the `go get` command manually like we did earlier.

- Because the justinas/alice package doesn't have any semantically-versioned releases available, the 'version' identifier in the `go.mod` file is the datetime of the latest commit followed by the first 12 characters of the commit hash.

Chapter 7.

RESTful Routing

In the next section of this book we're going to add a HTML form to our web application so that users can create new snippets.

To make this work smoothly, we're going to update our application routes so that requests to `/snippet/create` are handled differently based on the request method. Specifically:

- For `GET /snippet/create` requests we want to show the user the HTML form for adding a new snippet.
- For `POST /snippet/create` requests we want to process this form data and then insert a new `snippet` record into our database.

While we're at it, it makes sense to also restrict our other routes — which simply return information — to only support `GET` (and `HEAD`) requests.

Essentially, we want our application routes to end up looking like this:

Method	Pattern	Handler	Action
GET	/	home	Display the home page
GET	/snippet?id=1	showSnippet	Display a specific snippet
GET	/snippet/create	createSnippetForm	Display the new snippet form

Method	Pattern	Handler	Action
POST	/snippet/create	createSnippet	Create a new snippet
GET	/static/	http.FileServer	Serve a specific static file

Another routing-related improvement would be to use semantic URLs so that any variables are included in the URL path and not appended as a query string, like so:

Method	Pattern	Handler	Action
GET	/	home	Display the home page
GET	/snippet/:id	showSnippet	Display a specific snippet
GET	/snippet/create	createSnippetForm	Display the new snippet form
POST	/snippet/create	createSnippet	Create a new snippet
GET	/static/	http.FileServer	Serve a specific static file

Making these changes would give us an application routing structure that follows the fundamental principles of REST, and which should feel familiar and logical to anyone who works on modern web applications.

But as I mentioned earlier in the book, Go's servemux doesn't support method based routing or semantic URLs with variables in them. There are some tricks you can use to get around this, but most people tend to decide that it's easier to reach for a third-party package to help with routing.

In this section of the book we will:

- Briefly discuss the features of a few good third-party routers.
- Update our application to use one of these routers and follow a RESTful routing structure.

Chapter 7.1.

Installing a Router

There a literally hundreds of third-party routers for Go to pick from. And (fortunately or unfortunately, depending on your perspective) they all work a bit differently. They have different APIs, different logic for matching routes, and different behavioral quirks.

Out of all the third-party routers I've tried there are probably two that I'd recommend as a starting point: Pat and Gorilla Mux. Both have good documentation, decent test coverage, and work very well with the standard patterns for handlers and middleware that we've used throughout this book so far.

- bmizerany/pat is the more focused and lightweight of the two packages. It provides method-based routing and support for semantic URLs… and not much else. But it's solid at what it does, has an elegant API, and the code itself is very clear and well-written. A possible drawback is that the package isn't really maintained anymore.

- gorilla/mux is much more full-featured. In addition to method-based routing and support for semantic URLs, you can use it to route based on scheme, host and headers. Regular expression patterns in URLs are also supported. The downside of the package is that it's comparatively slow and memory hungry — but for a database-driven web application like ours the impact over the lifetime of a whole HTTP request is likely to be small.

For the web application we're building our needs are simple, and the more advanced features that Gorilla Mux offers simply aren't required. So, out of the two packages, we'll opt for Pat.

At the time of writing, the bmizerany/pat package doesn't have any semantically-versioned releases available. So, if you're following along, go ahead and install the latest version like so:

```
$ go get github.com/bmizerany/pat
go: finding github.com/bmizerany/pat latest
go: downloading github.com/bmizerany/pat v0.0.0-20170815010413-6226ea591a40
```

Alternative Routers

I've suggested the two routers above as a starting point but if you want to explore some more alternatives then I'd suggest looking at the following options — all of which are good in their own right — to see if they feel appropriate for you and your specific project.

- go-zoo/bone provides similar functionality to Pat, but with extra convenience functions for registering handler functions and support for regular-expression based routes. A downside is that — at the time of writing — the test coverage for the package is still incomplete.

- julienschmidt/httprouter is a famously fast radix-tree based router which supports method-based routing and semantic URLs. However it can’t cope with conflicting patterns caused by wildcard parameters, which can be a problem for applications that are using a RESTful routing scheme (in our case `/snippet/create` and `/snippet/:id` are conflicting patterns). If that doesn’t apply to you though, it’s a solid choice.

  Hint: use the `Router.Handler()` and `Router.HandlerFunc()` methods to register your routes — unlike the other methods they’re compatible with the standard Go middleware and handler patterns.

- dimfeld/httptreemux is another radix-tree based router but is designed so that it doesn’t suffer from the *conflicting patterns* issue that julienschmidt/httprouter has. A downside is that it only allows you to register `http.HandlerFunc` objects — not `http.Handler` objects — which means that it doesn’t work nicely with *route-specific* middleware that uses the standard pattern we described earlier.

- go-chi/chi is another popular option which uses a radix-tree for pattern matching and has a nice, flexible API. Notably, it also includes a go-chi/chi/middleware sub-package containing a range of useful middleware.

Chapter 7.2.

Implementing RESTful Routes

The basic syntax for creating a router and registering a route with the bmizernay/pat package looks like this:

```
mux := pat.New()
mux.Get("/snippet/:id", http.HandlerFunc(app.showSnippet))
```

In this code:

- The `"/snippet/:id"` pattern includes a *named capture* `:id`. The named capture acts like a wildcard, whereas the rest of the pattern matches literally. Pat will add the contents of the named capture to the URL query string at runtime behind the scenes.

- The `mux.Get()` method is used to register a URL pattern and handler which will be called *only* if the request has a `GET` HTTP method. Corresponding `Post()`, `Put()`, `Delete()` and other methods are also provided.

- Pat doesn't allow us to register handler functions directly, so you need to convert them using the `http.HandlerFunc()` adapter.

With all that in mind, let's head over to the `routes.go` file and update it to use Pat:

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

```go
package main

import (
    "net/http"

    "github.com/bmizerany/pat" // New import
    "github.com/justinas/alice"
)

func (app *application) routes() http.Handler {
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea

    mux := pat.New()
    mux.Get("/", http.HandlerFunc(app.home))
    mux.Get("/snippet/create", http.HandlerFunc(app.createSnippetForm))
    mux.Post("/snippet/create", http.HandlerFunc(app.createSnippet))
    mux.Get("/snippet/:id", http.HandlerFunc(app.showSnippet)) // Moved down

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

    return standardMiddleware.Then(mux)
}
```

There a few important things to point out here.

- Pat matches patterns *in the order that they are registered*. In our application, a HTTP request to `GET "/snippet/create"` is actually a valid match for two routes — it's an exact match for `/snippet/create`, and a wildcard match for `/snippet/:id` (the `"create"` part of the path would be treated as the `:id` parameter). So to ensure that the exact match takes preference, we need to register the exact match routes *before* any wildcard routes.

- URL patterns which end in a trailing slash (like `"/static/"`) work in the same way as with Go's inbuilt servemux. Any request which matches the *start* of the pattern will be dispatched to the corresponding handler.

- The pattern `"/"` is a special case. It will only match requests where the URL path is exactly `"/"`.

With those things in mind, there's also a few changes we now need to make to our `handlers.go` file:

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

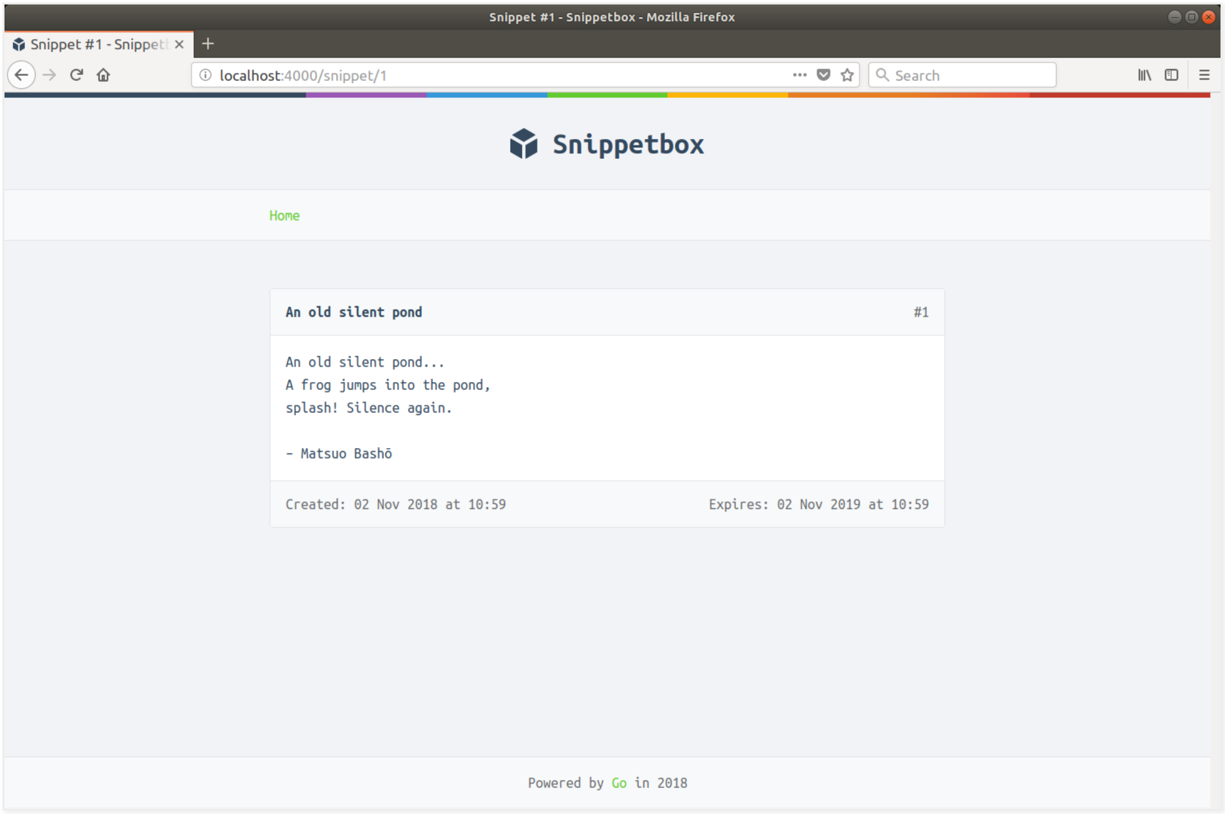
```go
package main

...

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    // Because Pat matches the "/" path exactly, we can now remove the manual c
    // of r.URL.Path != "/" from this handler.

    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    app.render(w, r, "home.page.tmpl", &templateData{
        Snippets: s,
    })
}
```

```go
func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    // Pat doesn't strip the colon from the named capture key, so we need to
    // get the value of ":id" from the query string instead of "id".
    id, err := strconv.Atoi(r.URL.Query().Get(":id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err == models.ErrNoRecord {
        app.notFound(w)
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }

    app.render(w, r, "show.page.tmpl", &templateData{
        Snippet: s,
    })
}

// Add a new createSnippetForm handler, which for now returns a placeholder res
func (app *application) createSnippetForm(w http.ResponseWriter, r *http.Reques
    w.Write([]byte("Create a new snippet..."))
}

func (app *application) createSnippet(w http.ResponseWriter, r *http.Request) {
    // The check of r.Method != "POST" is now superfluous and can be removed.

    title := "O snail"
    content := "O snail\nClimb Mount Fuji,\nBut slowly, slowly!\n\n- Kobayashi
    expires := "7"

    id, err := app.snippets.Insert(title, content, expires)
    if err != nil {
        app.serverError(w, err)
        return
    }
    // Change the redirect to use the new semantic URL style of /snippet/:id
    http.Redirect(w, r, fmt.Sprintf("/snippet/%d", id), http.StatusSeeOther)
}
```

Finally, we need to update the table in our `home.page.tmpl` file so that the links in the HTML also use the new semantic URL style of `/snippet/:id`.

```
{{template "base" .}}

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

{{define "body"}}
    <h2>Latest Snippets</h2>
    {{if .Snippets}}
     <table>
        <tr>
            <th>Title</th>
            <th>Created</th>
            <th>ID</th>
        </tr>
        {{range .Snippets}}

        <tr>
            <!-- Use the new semantic URL style-->
            <td><a href='/snippet/{{.ID}}'>{{.Title}}</a></td>
            <td>{{humanDate .Created}}</td>
            <td>#{{.ID}}</td>
        </tr>
        {{end}}
    </table>
    {{else}}
        <p>There's nothing to see here... yet!</p>
    {{end}}
{{end}}
```

With that done, restart the application and you should now be able to view the text snippets via the semantic URLs. For instance: `http://localhost:4000/snippet/1`.



An old silent pond #1

```
An old silent pond...
A frog jumps into the pond,
splash! Silence again.

- Matsuo Bashō
```

Created: 02 Nov 2018 at 10:59 Expires: 02 Nov 2019 at 10:59

Powered by Go in 2018

You can also see that requests using an unsupported HTTP method are met with a `405 Method Not Allowed` response. For example, try making a `POST` request to the same URL using curl:

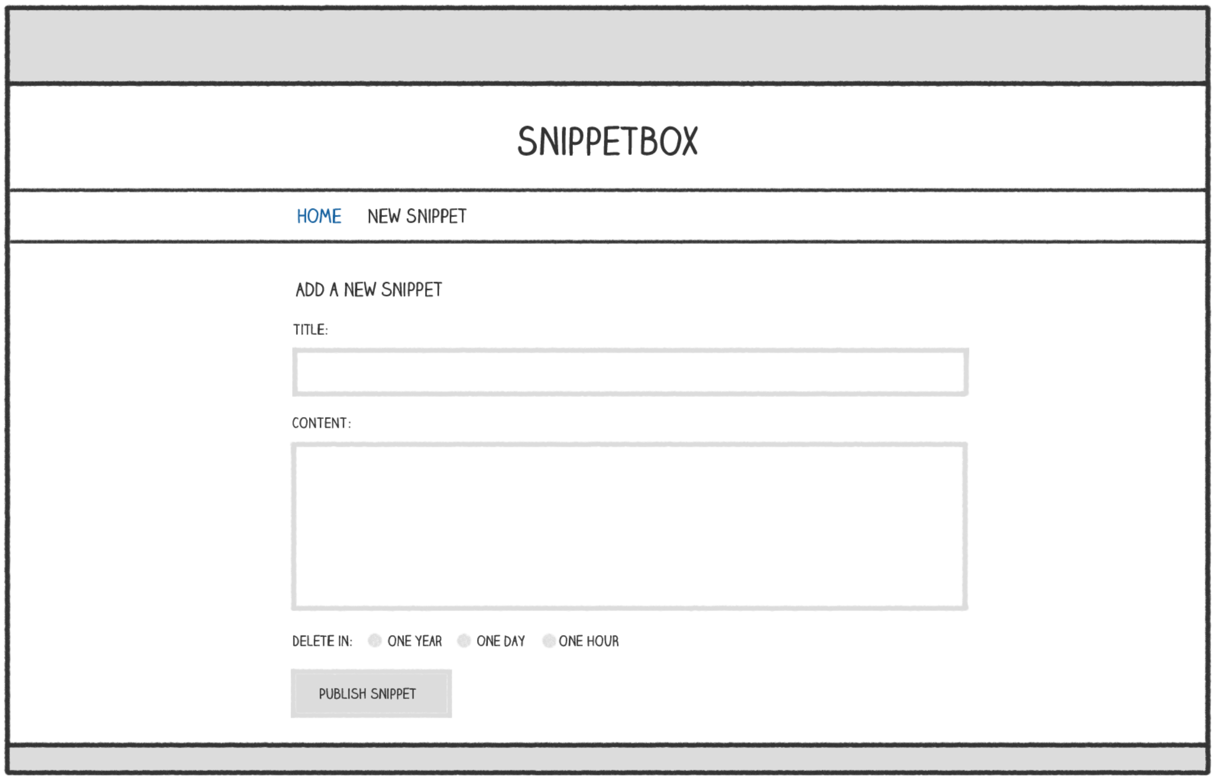
```
$ curl -I -X POST http://localhost:4000/snippet/1
HTTP/1.1 405 Method Not Allowed
Allow: HEAD, GET
Content-Type: text/plain; charset=utf-8
```

```
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Xss-Protection: 1; mode=block
Date: Mon, 23 Jul 2018 10:10:33 GMT
Content-Length: 19
```

Chapter 8.

Processing Forms

In this section of the book we're going to focus on allowing users of our web application to create new snippets via a HTML form which looks a bit like this:



The high-level workflow for processing this form will follow a standard `Post-Redirect-Get` pattern and look like this:

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 `createSnippet` 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 `"/snippet/: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 scale-up validation and keep your handlers clean by creating a form helper in a separate reusable package.

Chapter 8.1.

Setting Up a Form

Let's begin by making a new `ui/html/create.page.tmpl` file to hold the HTML for the form…

```
$ touch ui/html/create.page.tmpl
```

… and then add the following markup, using the same pattern that we used earlier in the book:

```
File: ui/html/create.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Create a New Snippet{{end}}

{{define "body"}}
<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 `body` template contains a standard web 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/base.layout.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-
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ub
    </head>
    <body>
        <header>
```

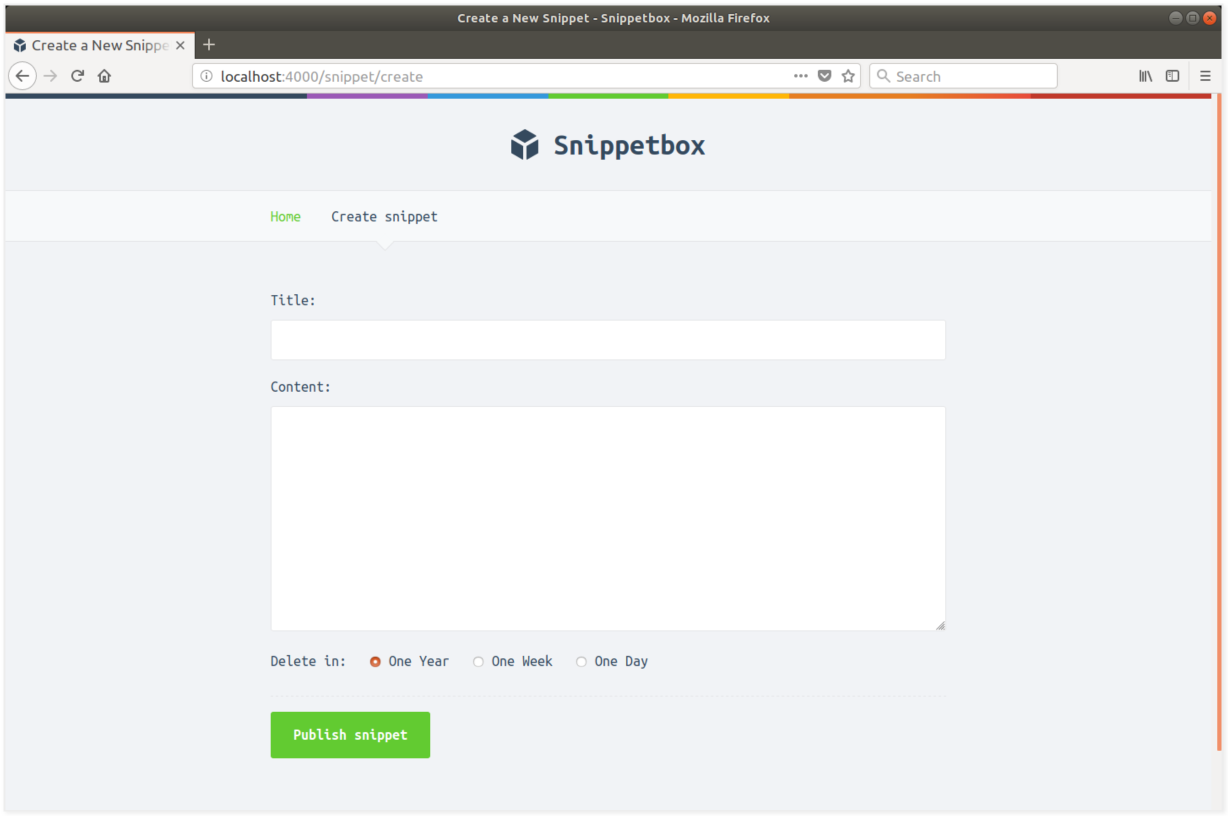
```
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <a href='/'>Home</a>
            <!-- Add a link to the new form -->
            <a href='/snippet/create'>Create snippet</a>
        </nav>
        <section>
            {{template "body" .}}
        </section>
        {{template "footer" .}}
        <script src="/static/js/main.js" type="text/javascript"></script>
    </body>
</html>
{{end}}
{{end}}
```

And finally, we need to update the `createSnippetForm` handler so that it renders our new page like so:

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

```
package main

...

func (app *application) createSnippetForm(w http.ResponseWriter, r *http.Reques
    app.render(w, r, "create.page.tmpl", nil)
}

...
```

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:


Create a New Snippet - Snippetbox - Mozilla Firefox
Create a New Snippe
localhost:4000/snippet/create
Search
Snippetbox
Home
Create snippet
Publish snippet

Title:

Content:

Delete in: [✓ One Year] One Week One Day

Chapter 8.2.

Parsing Form Data

Thanks to the work we did previously in the RESTful routing section, any `POST /snippets/create` requests are already being dispatched to our `createSnippet` 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.

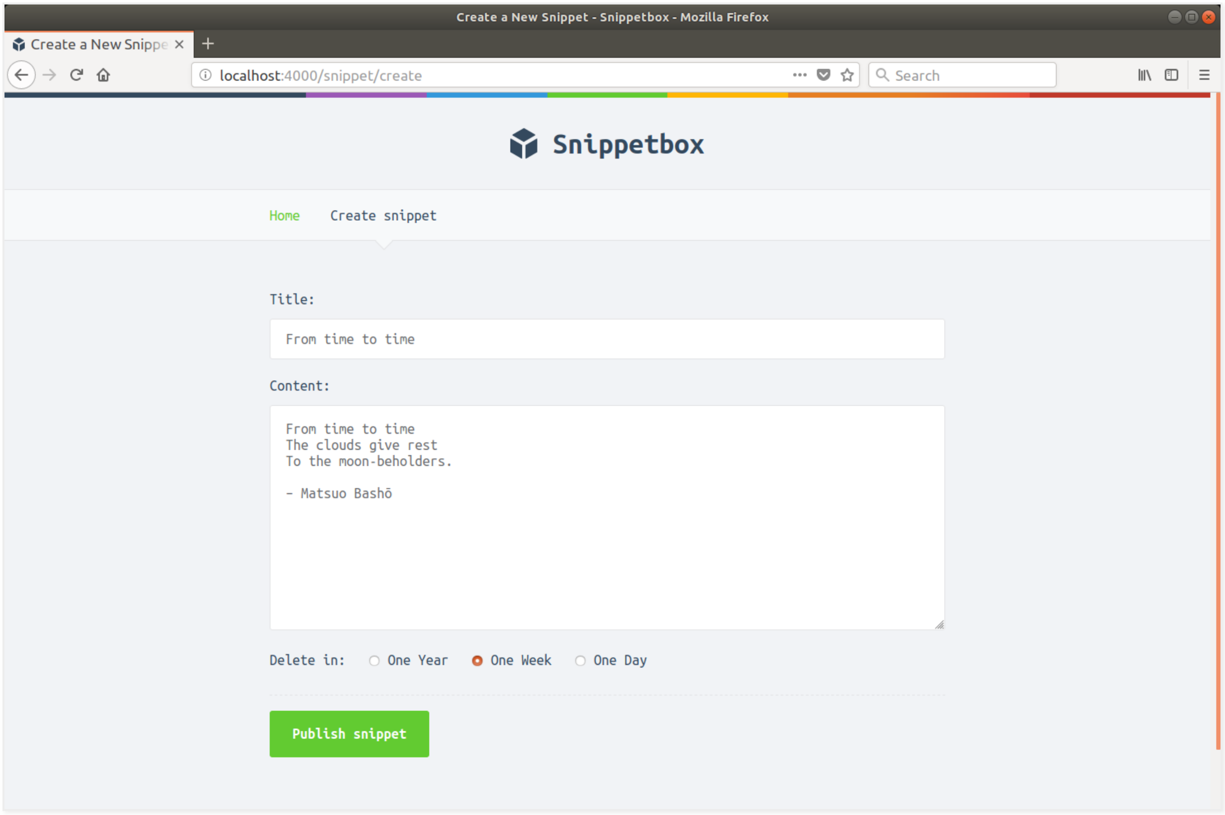
2. We can then get to the form data contained in `r.PostForm` by using the `r.PostForm.Get()` method. For example, we can retrieve the value of the `title` field with `r.PostForm.Get("title")`. If there is no matching field name in the form this will return the empty string `""`, similar to the way that query string parameters worked earlier in the book.

Open your `cmd/web/handlers.go` file and update it to include the following code:

File: cmd/web/handlers.go

```go
package main

...

func (app *application) createSnippet(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
    // 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 relevant data fields
    // from the r.PostForm map.
    title := r.PostForm.Get("title")
    content := r.PostForm.Get("content")
    expires := r.PostForm.Get("expires")

    // Create a new snippet record in the database using the form data.
    id, err := app.snippets.Insert(title, content, expires)
    if err != nil {
        app.serverError(w, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet/%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:


Create a New Snippet - Snippetbox - Mozilla Firefox

localhost:4000/snippet/create

Title:

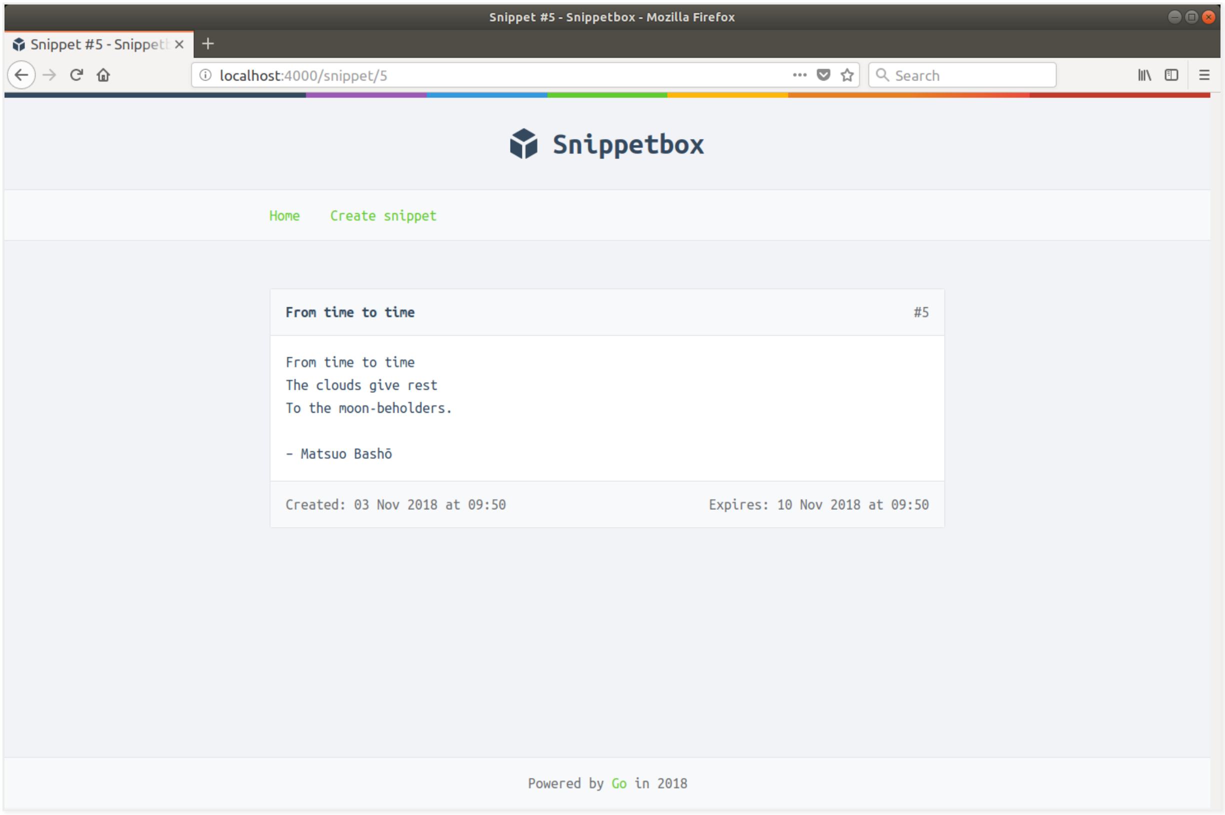
From time to time

Content:

```
From time to time
The clouds give rest
To the moon-beholders.

- Matsuo Bashō
```

Delete in: One Year [✓ One Week] One Day

And then submit the form. If everything has worked, you should be redirected to a page displaying your new snippet like so:



From time to time #5

```
From time to time
The clouds give rest
To the moon-beholders.

- Matsuo Bashō
```

Created: 03 Nov 2018 at 09:50 Expires: 10 Nov 2018 at 09:50

Powered by Go in 2018

Additional Information

The r.Form Map

In our code above, we accessed the form values via the `r.PostForm` map. But an alternative approach is to use the (subtly different) `r.Form` map.

The `r.PostForm` map is populated only for `POST`, `PATCH` and `PUT` requests, and contains the form data from the request body.

In contrast, the `r.Form` map is populated for all requests (irrespective of their HTTP method), and contains the form data from any request body **and** any query string parameters. So, if our form was submitted to `/snippet/create?foo=bar`, we could also get the value of the `foo` parameter by calling `r.Form.Get("foo")`. Note that in the event of a conflict, the request body value will take precedent over the query string parameter.

Using the `r.Form` map can be useful if your application sends data in a HTML form and in the URL, or you have an application that is agnostic about how parameters are passed. But in our case those things aren't applicable. We expect our form data to be sent in the request body only, so it's for sensible for us to access it via `r.PostForm`.

The FormValue and PostFormValue Methods

The `net/http` package also provides the methods `r.FormValue()` and `r.PostFormValue()`. These are essentially shortcut functions that call `r.ParseForm()` for you, and then fetch the appropriate field value from `r.Form` or `r.PostForm` respectively.

I recommend avoiding these shortcuts because they *silently ignore any errors* returned by `r.ParseForm()`. That's not ideal — it means our application could be encountering errors and failing for users, but there's no feedback mechanism to let them know.

Multiple-Value Fields

Strictly speaking, the `r.PostForm.Get()` method that we've used above 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: Item %s\n", i, item)
}
```

Form Size

Unless you're sending multipart data (i.e. your form has the `enctype="multipart/form-data"` attribute) then `POST`, `PUT` and `PATCH` request bodies are limited to 10MB. If this is exceeded then `r.ParseForm()` will return an error.

If you want to change this 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.

Chapter 8.3.

Data Validation

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`, `content` and `expires` fields are not empty.
- Check that the `title` field is not more than 100 characters long.
- Check that the `expires` value 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 `createSnippet` handler to include the appropriate validation rules like so:

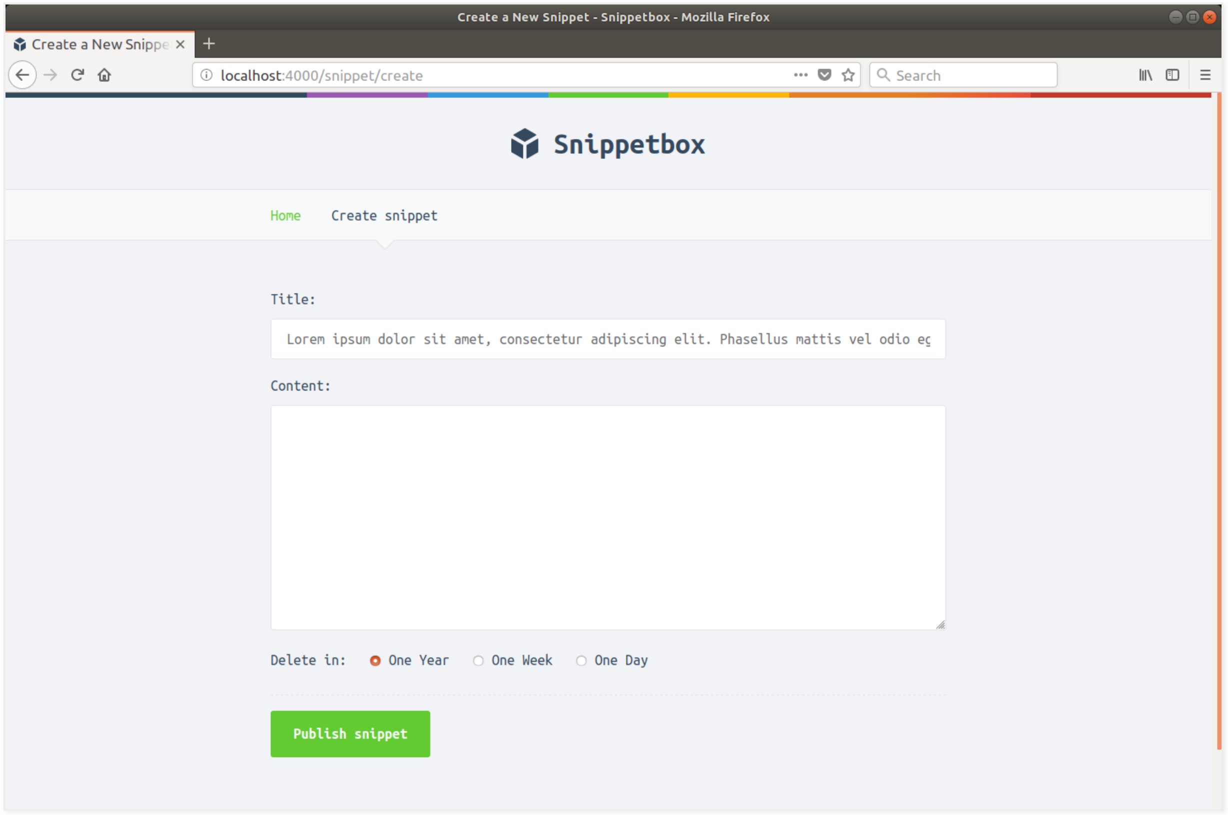
```
File: cmd/web/handlers.go
```

```
package main

import (
    "fmt"
    "net/http"
    "strconv"
    "strings"      // New import
    "unicode/utf8" // New import

    "alexedwards.net/snippetbox/pkg/models"
)

...

func (app *application) createSnippet(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 := r.PostForm.Get("expires")

    // Initialize a map to hold any validation errors.
    errors := make(map[string]string)

    // Check that the title field is not blank and is not more than 100 characte
    // 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) == "" {
        errors["title"] = "This field cannot be blank"
    } else if utf8.RuneCountInString(title) > 100 {
        errors["title"] = "This field is too long (maximum is 100 characters)"
    }

    // Check that the Content field isn't blank.
    if strings.TrimSpace(content) == "" {
        errors["content"] = "This field cannot be blank"
    }
    // Check the expires field isn't blank and matches one of the permitted
    // values ("1", "7" or "365").
    if strings.TrimSpace(expires) == "" {
        errors["expires"] = "This field cannot be blank"
    } else if expires != "365" && expires != "7" && expires != "1" {
        errors["expires"] = "This field is invalid"
    }

    // If there are any errors, dump them in a plain text HTTP response and ret
    // from the handler.
    if len(errors) > 0 {
        fmt.Fprint(w, errors)
        return
    }

    id, err := app.snippets.Insert(title, content, expires)
    if err != nil {
        app.serverError(w, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet/%d", id), http.StatusSeeOther)
}
```

Note: When we check the length of the `title` field, we're using the `utf8.RuneCount()` function — not the builtin `len()` function. This is because we want to count the number of *characters* in the title rather than the number of bytes. To illustrate the difference, the string `"Zoë"` has 3 characters but a length of 4 bytes because of the umlauted `ë` character.

Also note: You can find a bunch of code patterns for processing and validating different types of inputs in this blog post.

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…

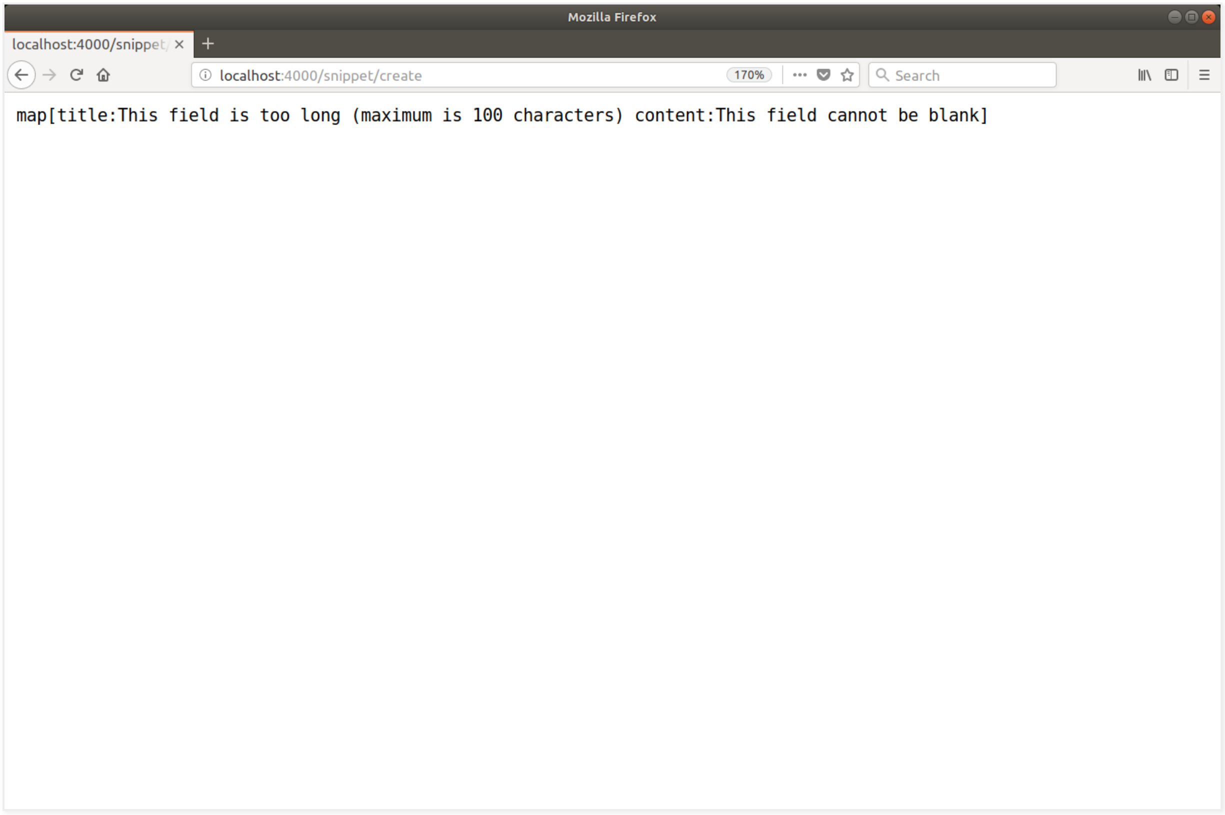

Create a New Snippet - Snippetbox - Mozilla Firefox
Create a New Snippe
localhost:4000/snippet/create
Search
Snippetbox
Home
Create snippet
Publish snippet

Title:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus mattis vel odio eg

Content:

Delete in: [✓ One Year] One Week One Day

And you should see a dump of the appropriate validation failure messages, like so:



```
map[title:This field is too long (maximum is 100 characters) content:This field cannot be blank]
```

Displaying Validation Errors and Repopulating Fields

Now that the `createSnippet` 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 form, highlighting the fields which failed validation and automatically re-populating any previously submitted data.

To do this, let's begin by adding two new fields to our `templateData` struct: `FormErrors` to hold any validation errors and `FormData` to hold any previously submitted data, like so:

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

```go
package main

import (
    "html/template"
    "net/url" // New import
    "path/filepath"
    "time"

    "alexedwards.net/snippetbox/pkg/models"
)

// Add FormData and FormErrors fields to the templateData struct.
type templateData struct {
    CurrentYear int
    FormData    url.Values
    FormErrors  map[string]string
    Snippet     *models.Snippet
    Snippets    []*models.Snippet
}

...
```

Note: The `FormData` field has the type `url.Values`, which is the same underlying type as the `r.PostForm` map that held the data sent in the request body. This will make it easy for us to pass back the previously-submitted data.

Now let's update the `createSnippet` handler again, so that if any validation errors are encountered the form is re-displayed with the relevant errors and form data passed to the template. Like so:

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

```go
package main

...

func (app *application) createSnippet(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 := r.PostForm.Get("expires")

    errors := make(map[string]string)

    if strings.TrimSpace(title) == "" {
        errors["title"] = "This field cannot be blank"
    } else if utf8.RuneCountInString(title) > 100 {
        errors["title"] = "This field is too long (maximum is 100 characters)"
    }

    if strings.TrimSpace(content) == "" {
        errors["content"] = "This field cannot be blank"
    }

    if strings.TrimSpace(expires) == "" {
        errors["expires"] = "This field cannot be blank"
    } else if expires != "365" && expires != "7" && expires != "1" {
        errors["expires"] = "This field is invalid"
    }

    // If there are any validation errors, re-display the create.page.tmpl
    // template passing in the validation errors and previously submitted
    // r.PostForm data.
    if len(errors) > 0 {
        app.render(w, r, "create.page.tmpl", &templateData{

            FormErrors: errors,
            FormData:   r.PostForm,
        })
        return
    }

    id, err := app.snippets.Insert(title, content, expires)
    if err != nil {
        app.serverError(w, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet/%d", id), http.StatusSeeOther)
}
```

So now when there are any validation errors we are re-displaying the `create.page.tmpl` template, passing in the map of errors in the `FormErrors` field of the template data, and passing in the previously submitted data in the `FormData` field.

So how will we render these errors in the template?

The underlying type of the `FormErrors` field is a `map[string]string` (keyed by form field name). And for maps, it's possible to access the value for a given key by simply postfixing dot with the key name. For example, to render any error message for the `title` field we can use the tag `{{.FormErrors.title}}` in our template. It's important to mentions that — unlike struct fields — map key names *don't* have to be capitalized to access them from a template.

The underlying type of the `FormData` is `url.Values` and we can use it's `Get()` method to retrieve the value for a field, just like we did in our `createSnippet` handler. For example, to render the previously submitted value for the `title` field we can use the tag `{{.FormData.Get "title"}}` in our template.

With that in mind, let's update the `create.page.tmpl` file to display the data and validation error messages for each field, if they exist:

```
File: ui/html/create.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Create a New Snippet{{end}}

{{define "body"}}
<form action='/snippet/create' method='POST'>
    <div>
        <label>Title:</label>
        {{with .FormErrors.title}}
            <label class='error'>{{.}}</label>
        {{end}}
        <input type='text' name='title' value='{{.FormData.Get "title"}}'>
    </div>
    <div>
        <label>Content:</label>
        {{with .FormErrors.content}}
            <label class='error'>{{.}}</label>
        {{end}}
        <textarea name='content'>{{.FormData.Get "content"}}</textarea>
    </div>
    <div>
        <label>Delete in:</label>
        {{with .FormErrors.expires}}
            <label class='error'>{{.}}</label>
        {{end}}
        {{$exp := or (.FormData.Get "expires") "365"}}
        <input type='radio' name='expires' value='365' {{if (eq $exp "365")}}ch
        <input type='radio' name='expires' value='7' {{if (eq $exp "7")}}checke
        <input type='radio' name='expires' value='1' {{if (eq $exp "1")}}checke
    </div>
    <div>
        <input type='submit' value='Publish snippet'>
    </div>
</form>
{{end}}
```

Hopefully this markup and our use of the `{{with}}` action to control the display of dynamic data is generally clear — it’s just using techniques that we’ve already seen and discussed earlier in the book.

But let’s take a moment to talk about the line:

```
{{$exp := or (.FormData.Get "expires") "365"}}
```

This is essentially creating a new `$exp` template variable which uses the `or` template function to set the variable to the value yielded by `.FormData.Get "expires"` *or* if that’s empty then the default value of `"365"` instead.

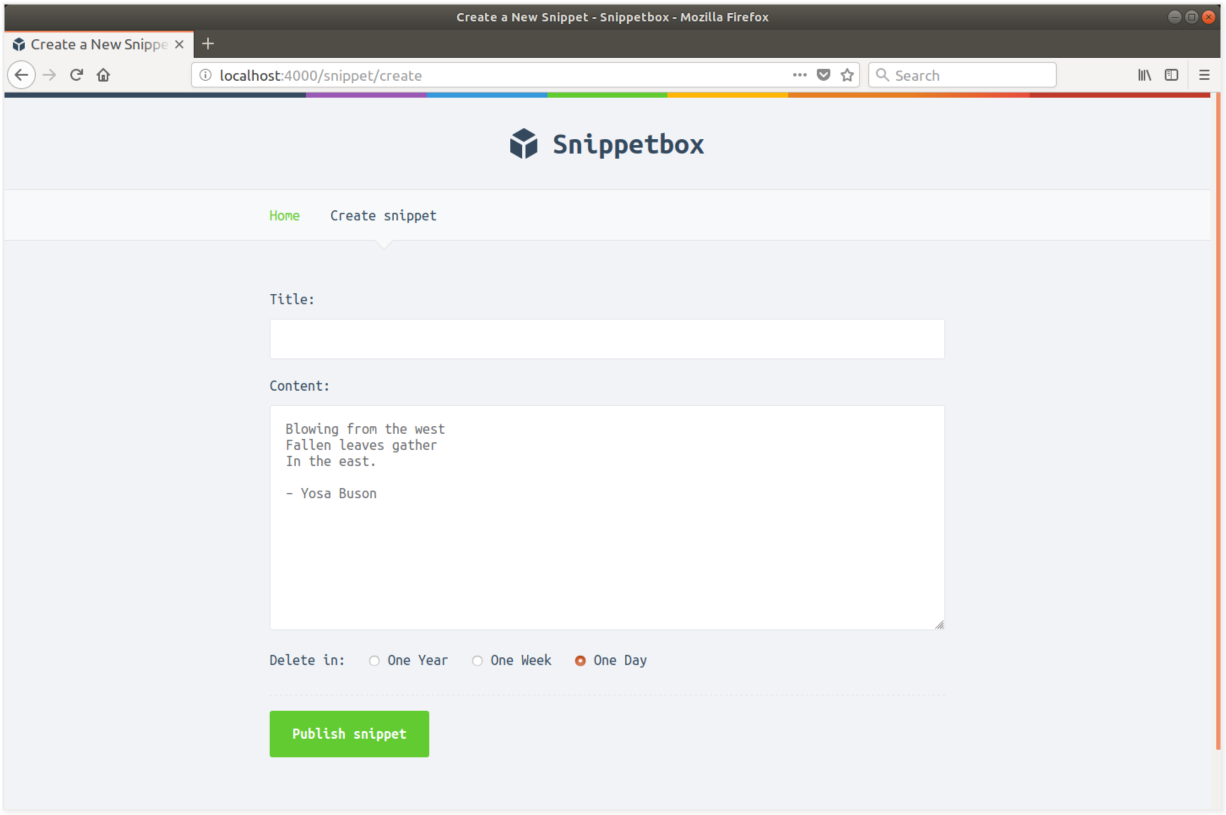
Notice here how we’ve used `()` parentheses to group the `.FormData.Get` method and its parameters in order to pass its output to the `or` action?

We then use this variable in conjunction with the `eq` function to add the `checked` attribute to the appropriate radio button, like so:

```
{{if (eq $exp "365")}}checked{{end}}
```

Anyway, restart the application and visit `http://localhost:4000/snippet/create` in your browser.

Try adding some content and changing the default expiry time, but *leave the title field blank* like so:


Create a New Snippet - Snippetbox - Mozilla Firefox
Create a New Snippe
localhost:4000/snippet/create
Search
Snippetbox
Home
Create snippet
Publish snippet

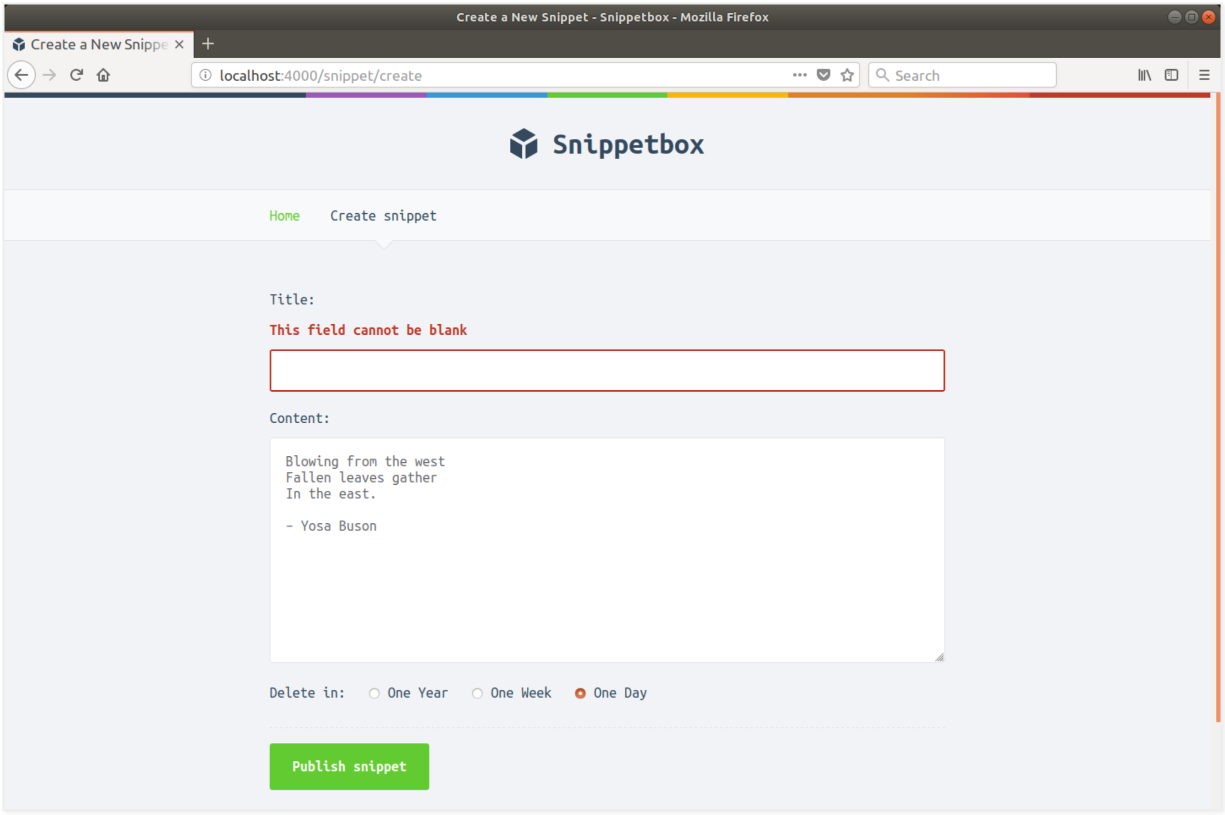
Title:

Content:

```
Blowing from the west
Fallen leaves gather
In the east.

- Yosa Buson
```

Delete in: One Year One Week [✓ One Day]

After submission you should now see the form re-displayed, with correctly re-populated content and expiry option and a “This field cannot be blank” error message alongside the title field:


Create a New Snippet - Snippetbox - Mozilla Firefox
Create a New Snippe
localhost:4000/snippet/create
Search
Snippetbox
Home
Create snippet

Title:

This field cannot be blank

Content:

```
Blowing from the west
Fallen leaves gather
In the east.

- Yosa Buson
```

Delete in: One Year One Week [✓ One Day]

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.

Chapter 8.4.

Scaling Data Validation

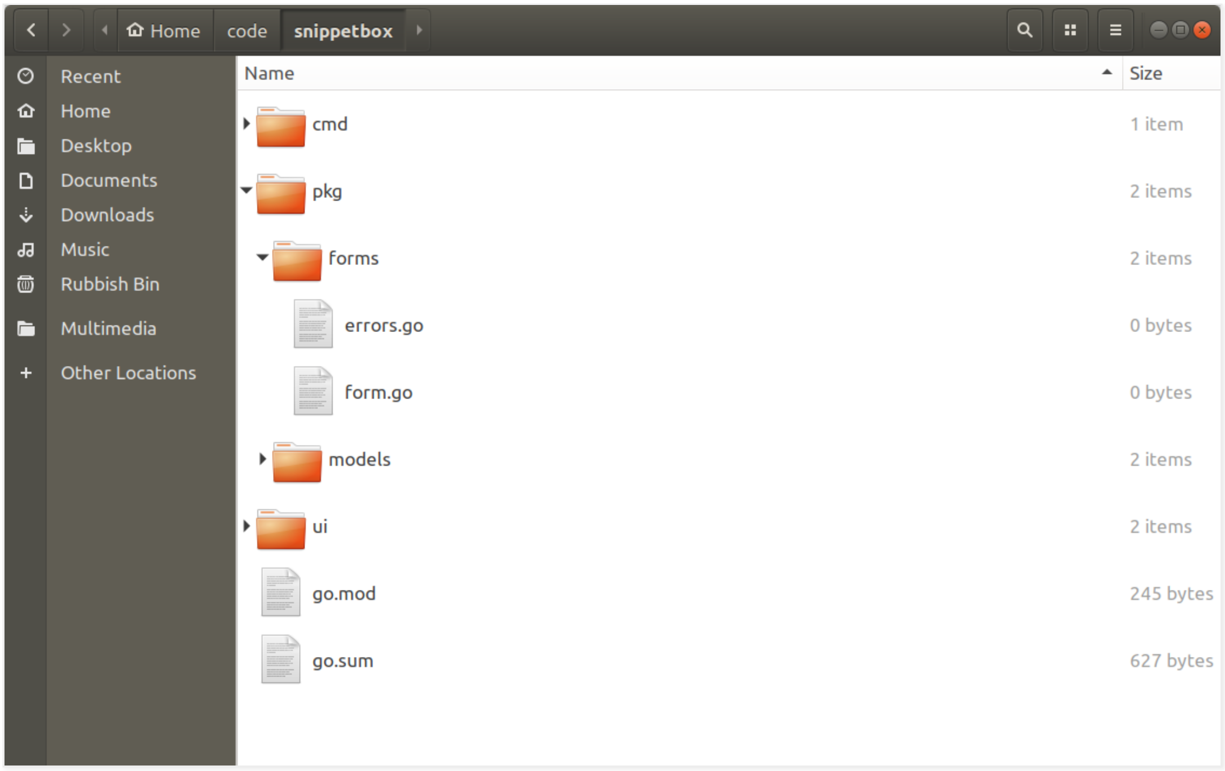
OK, so we're now in the position where our application is validating the form data according to our business rules and gracefully handling any validation errors. That's great, but it's taken quite a bit of work to get there.

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.

Let's address this by creating a `forms` package to abstract some of this behavior and reduce the boilerplate code in our handler. We won't actually change how the application works for the user at all; it's just a refactoring of our codebase.

We'll begin by making a new `pkg/forms` folder in the project directory and adding two files — `forms.go` and `errors.go` — like so:

```
$ cd $HOME/code/snippetbox
$ mkdir pkg/forms
$ touch pkg/forms/errors.go
$ touch pkg/forms/form.go
```



Name	Size
cmd	1 item
pkg	2 items
forms	2 items
errors.go	0 bytes
form.go	0 bytes
models	2 items
ui	2 items
go.mod	245 bytes
go.sum	627 bytes

Once that's done, open the `errors.go` file and create a new `errors` type, which we will use to hold the validation error messages for forms, like so:

```
File: pkg/forms/errors.go
```

```
package forms

// Define a new errors type, which we will use to hold the validation error
// messages for forms. The name of the form field will be used as the key in
// this map.
type errors map[string][]string

// Implement an Add() method to add error messages for a given field to the map
func (e errors) Add(field, message string) {
    e[field] = append(e[field], message)
}

// Implement a Get() method to retrieve the first error message for a given
// field from the map.
func (e errors) Get(field string) string {
    es := e[field]
    if len(es) == 0 {
        return ""
    }
    return es[0]
}
```

And then in the `form.go` file add the following code:

```
File: pkg/forms/form.go
```

```
package forms

import (
    "fmt"
    "net/url"
    "strings"
    "unicode/utf8"
)

// Create a custom Form struct, which anonymously embeds a url.Values object
// (to hold the form data) and an Errors field to hold any validation errors
// for the form data.
type Form struct {
    url.Values
    Errors errors
}

// Define a New function to initialize a custom Form struct. Notice that
// this takes the form data as the parameter?
func New(data url.Values) *Form {
    return &Form{
        data,
        errors(map[string][]string{}),
    }
}

// Implement a Required method to check that specific fields in the form
// data are present and not blank. If any fields fail this check, add the
// appropriate message to the form errors.
func (f *Form) Required(fields ...string) {
    for _, field := range fields {
        value := f.Get(field)
        if strings.TrimSpace(value) == "" {
            f.Errors.Add(field, "This field cannot be blank")
        }
    }
}

// Implement a MaxLength method to check that a specific field in the form
// contains a maximum number of characters. If the check fails then add the
// appropriate message to the form errors.
func (f *Form) MaxLength(field string, d int) {
    value := f.Get(field)
    if value == "" {
        return
    }
    if utf8.RuneCountInString(value) > d {
        f.Errors.Add(field, fmt.Sprintf("This field is too long (maximum is %d
    }
}

// Implement a PermittedValues method to check that a specific field in the for
// matches one of a set of specific permitted values. If the check fails
// then add the appropriate message to the form errors.
func (f *Form) PermittedValues(field string, opts ...string) {
    value := f.Get(field)
    if value == "" {
        return
    }
    for _, opt := range opts {
        if value == opt {
            return
        }
    }
    f.Errors.Add(field, "This field is invalid")
}

// Implement a Valid method which returns true if there are no errors.
func (f *Form) Valid() bool {
    return len(f.Errors) == 0
}
```

The next step is to update the `templateData` struct so that we can pass this new `forms.Form` struct to our templates:

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

```
package main

import (
    "html/template"
    "path/filepath"
    "time"

    "alexedwards.net/snippetbox/pkg/forms" // New import
    "alexedwards.net/snippetbox/pkg/models"
)

// Update the templateData fields, removing the individual FormData and
// FormErrors fields and replacing them with a single Form field.
type templateData struct {
    CurrentYear int
    Form        *forms.Form
    Snippet     *models.Snippet
    Snippets    []*models.Snippet
}
```

```
...
```

And now for the payoff… Open your `handlers.go` file and update it to use the `forms.Form` struct and validation methods that we just created, like so:

```
File: cmd/web/handlers
```

```
package main

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

    "alexedwards.net/snippetbox/pkg/forms" // New import
    "alexedwards.net/snippetbox/pkg/models"
)

...

func (app *application) createSnippetForm(w http.ResponseWriter, r *http.Reques
    app.render(w, r, "create.page.tmpl", &templateData{
        // Pass a new empty forms.Form object to the template.
        Form: forms.New(nil),
    })
}

func (app *application) createSnippet(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }
    // Create a new forms.Form struct containing the POSTed data from the
    // form, then use the validation methods to check the content.
    form := forms.New(r.PostForm)
    form.Required("title", "content", "expires")
    form.MaxLength("title", 100)
    form.PermittedValues("expires", "365", "7", "1")

    // If the form isn't valid, redisplay the template passing in the
    // form.Form object as the data.
    if !form.Valid() {
        app.render(w, r, "create.page.tmpl", &templateData{Form: form})
        return
    }

    // Because the form data (with type url.Values) has been anonymously embedd
    // in the form.Form struct, we can use the Get() method to retrieve
    // the validated value for a particular form field.
    id, err := app.snippets.Insert(form.Get("title"), form.Get("content"), form
    if err != nil {
        app.serverError(w, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet/%d", id), http.StatusSeeOther)
}
```

And all that's left is to update the `create.page.tmpl` file to use the data contained in the new `form.Form` struct, like so:

```
File: ui/html/create.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Create a New Snippet{{end}}
```

```
{{define "body"}}
<form action='/snippet/create' method='POST'>
    {{with .Form}}
        <div>
            <label>Title:</label>
            {{with .Errors.Get "title"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='text' name='title' value='{{.Get "title"}}'>
        </div>
        <div>
            <label>Content:</label>
            {{with .Errors.Get "content"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <textarea name='content'>{{.Get "content"}}</textarea>
        </div>
        <div>
            <label>Delete in:</label>
            {{with .Errors.Get "expires"}}
                <label class='error'>{{.}}</label>
            {{end}}
            {{$exp := or (.Get "expires") "365"}}
            <input type='radio' name='expires' value='365' {{if (eq $exp "365")
            <input type='radio' name='expires' value='7' {{if (eq $exp "7")}}ch
            <input type='radio' name='expires' value='1' {{if (eq $exp "1")}}ch
        </div>
        <div>
            <input type='submit' value='Publish snippet'>
        </div>
    {{end}}
</form>
{{end}}
```

This is shaping up nicely.

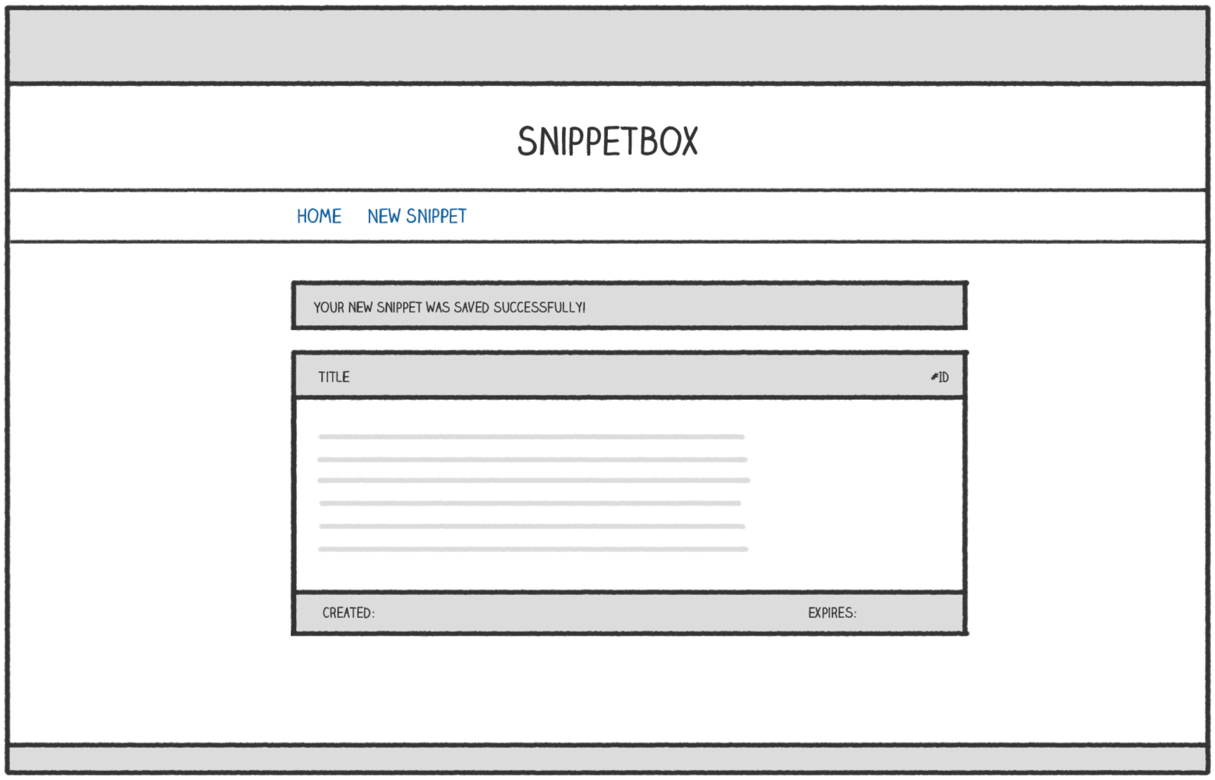
We've now got a `forms` package with validation rules and logic that can be reused across our application, and can be easily extended to include additional rules in the future (which we will do later in the book). Both form data and errors are neatly encapsulated in a single `forms.Form` object — which we can easily pass to our templates — and it provides a simple and consistent API for retrieving and displaying both the form data and any error messages.

Go ahead and re-run the application now. All being well, you should find that the form and validation rules are working correctly in the same manner as before.

Chapter 9.

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're coming from a background of Rails, Django, Laravel or similar frameworks you might know this type of functionality as a *flash message*.

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 you can customize session behavior (including timeouts and cookie settings) based on your application's needs.
- How to use sessions to safely and securely share data between requests for a particular user.

Chapter 9.1.

Installing a Session Manager

There's a lot of security considerations when it comes to working with sessions, and proper implementation is non-trivial. Unless you really need to roll your own implementation it's a good idea to use an existing, well-tested, third-party package.

Unlike routers, there's are only a few good session management packages for Go which aren't framework-specific:

- gorilla/sessions is the most established and well-known package. It has a simple and easy-to-use API and supports a huge range of third-party session stores including MySQL, PostgreSQL and Redis. However, importantly (and unfortunately) there are problems with memory leaks and it doesn't provide a mechanism to renew session tokens — which means that it's vulnerable to session fixation attacks if you're using one of the third-party server-side session stores. Both problems are due to be addressed in version 2 of the package, but this isn't yet available at the time of writing.

- alexedwards/scs is another session manager which supports a variety of server-side session stores including MySQL, PostgreSQL and Redis. It doesn't suffer from the same security and memory leak issues as Gorilla Sessions, supports automatic loading and saving of session data via middleware, and has a nice interface for type-safe manipulation of data.

- golangcollege/sessions provides a cookie-based session store using encrypted and authenticated cookies. It's lightweight and focused, with a simple API, and supports automatic loading and saving of session data via middleware. Because it uses cookies to store session data it's very performant and easy to setup, but there's a couple of important downsides: the amount of information you can store is limited (to 4KB) and you can't *revoke* sessions in the same way that you can using a server-side store.

In general, if you're happy to use cookie-based sessions then I recommend using the golangcollege/sessions package, which is exactly what we will do in the rest of this section of the book.

If you're following along, go ahead and install `v1` of the package like so:

```
$ cd $HOME/code/snippetbox
$ go get github.com/golangcollege/sessions@v1
go: finding github.com/golangcollege/sessions v1.0.0
go: downloading github.com/golangcollege/sessions v1.0.0
go: finding golang.org/x/crypto v0.0.0-20181009213950-7c1a557ab941
go: downloading golang.org/x/crypto v0.0.0-20181009213950-7c1a557ab941
```

Notice how the golang.org/x/crypto package — which is a sub-dependency of golangcollege/sessions — has been automatically downloaded too?

Your `go.mod` file should now look similar to this:

File: go.mod

```
module alexedwards.net/snippetbox

require (
    github.com/bmizerany/pat v0.0.0-20170815010413-6226ea591a40 // indirect
    github.com/go-sql-driver/mysql v1.4.0 // indirect
    github.com/golangcollege/sessions v1.0.0 // indirect
    github.com/justinas/alice v0.0.0-20171023064455-03f45bd4b7da // indirect
)
```

Chapter 9.2.

Setting Up the Session Manager

In this chapter I'll run through the process of setting up and using the golangcollege/sessions 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 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.

The other thing that we'll need is a 32-byte long *secret key* for encrypting and authenticating the session cookies. We'll update our application so that it can accept this secret key via a new command-line flag.

Open your `main.go` file and update it as follows:

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

```go
package main

import (
    "database/sql"
    "flag"
    "html/template"
    "log"
    "net/http"
    "os"
    "time" // New import

    "alexedwards.net/snippetbox/pkg/models/mysql"

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

// Add a new session field to the application struct.
type application struct {
    errorLog      *log.Logger
    infoLog       *log.Logger
    session       *sessions.Session
    snippets      *mysql.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 dat
    // Define a new command-line flag for the session secret (a random key whic
    // will be used to encrypt and authenticate session cookies). It should be
    // bytes long.
    secret := flag.String("secret", "s6Ndh+pPbnzHbS*+9Pk8qGWhTzbpa@ge", "Secret
    flag.Parse()

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    db, err := openDB(*dsn)
    if err != nil {
        errorLog.Fatal(err)
    }
    defer db.Close()

    templateCache, err := newTemplateCache("./ui/html/")
    if err != nil {
        errorLog.Fatal(err)
    }

    // Use the sessions.New() function to initialize a new session manager,
    // passing in the secret key as the parameter. Then we configure it so
    // sessions always expires after 12 hours.
    session := sessions.New([]byte(*secret))
    session.Lifetime = 12 * time.Hour

    // And add the session manager to our application dependencies.
    app := &application{
        errorLog:      errorLog,
        infoLog:       infoLog,
        session:       session,
        snippets:      &mysql.SnippetModel{DB: db},
        templateCache: templateCache,
    }

    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  app.routes(),
    }

    infoLog.Printf("Starting server on %s", *addr)
    err = srv.ListenAndServe()
    errorLog.Fatal(err)
}

...
```

Note: The `sessions.New()` function returns a `Session` struct which holds the configuration settings for the session. In the code above we've set the `Lifetime` field of this struct so that sessions expire after 12 hours, 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 `Session.Enable()` method. This middleware loads and saves session data to and from the session cookie with every HTTP request and response as appropriate.

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 `/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 `standardMiddleware` chain.

Instead, let's create a new `dynamicMiddleware` 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
```

```go
package main

import (
    "net/http"

    "github.com/bmizerany/pat"
    "github.com/justinas/alice"
)

func (app *application) routes() http.Handler {
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea

    // Create a new middleware chain containing the middleware specific to
    // our dynamic application routes. For now, this chain will only contain
    // the session middleware but we'll add more to it later.
    dynamicMiddleware := alice.New(app.session.Enable)

    mux := pat.New()
    // Update these routes to use the new dynamic middleware chain followed
    // by the appropriate handler function.
    mux.Get("/", dynamicMiddleware.ThenFunc(app.home))
    mux.Get("/snippet/create", dynamicMiddleware.ThenFunc(app.createSnippetForm
    mux.Post("/snippet/create", dynamicMiddleware.ThenFunc(app.createSnippet))
    mux.Get("/snippet/:id", dynamicMiddleware.ThenFunc(app.showSnippet))

    // Leave the static files route unchanged.
    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Get("/static/", http.StripPrefix("/static", fileServer))

    return standardMiddleware.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 simply need to wrap your handler functions with the session middleware instead. Like this:

```
mux := pat.New()
mux.Get("/", app.session.Enable(http.HandlerFunc(app.home)))
mux.Get("/snippet/create", app.session.Enable(http.HandlerFunc(app.createSnippe
// ... etc
```

Chapter 9.3.

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.

To add the confirmation message to the session data for a user we should use the `*Session.Put()` method. The second parameter to this is the *key* for the data, which we'll also use to subsequently retrieve the data from the session. It'll look a bit like this:

```
app.session.Put(r, "flash", "Snippet successfully created!")
```

To retrieve the data from the session we have two choices. We could use the `*Session.Get()` method (which returns an `interface{}` type) and type assert the value to a string, a bit like this:

```
flash, ok := app.session.Get(r, "flash").(string)
if !ok {
    app.serverError(w, errors.New("type assertion to string failed"))
}
```

Or alternatively, we can use the `*Session.GetString()` method which takes care of the type conversion for us. If there's no matching key in the session data — or the retrieved value couldn’t be asserted to a string — this method will return the empty string `""`.

```
flash := app.session.GetString(r, "flash")
```

The golangcollege/sessions package also provides similar helpers for retrieving `bool`, `[]byte`, `float64`, `int` and `time.Time` types.

However, because we want our confirmation message to be displayed once — and only once — we’ll also need to *remove* the message from the session data after retrieving it.

We could use the `*Session.Remove()` method to do this, but a better option is the `*Session.PopString()` method, which retrieves a string value for a given key and then deletes it from the session data in one step.

```
flash := app.session.PopString(r, "flash")
```

That’s a very quick rundown of the basic functions for adding and removing session data, but again I recommend familiarizing yourself with the complete package documentation if you’re planning on using it in production applications.

Using the Session Data in Practice

Let’s put what we’ve just learned into action and update the `createSnippet` handler, so that it adds a confirmation message to the current user’s session data like so:

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

```go
package main

...

func (app *application) createSnippet(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    form := forms.New(r.PostForm)
    form.Required("title", "content", "expires")
    form.MaxLength("title", 100)
    form.PermittedValues("expires", "365", "7", "1")

    if !form.Valid() {
        app.render(w, r, "create.page.tmpl", &templateData{Form: form})
        return
    }

    id, err := app.snippets.Insert(form.Get("title"), form.Get("content"), form
    if err != nil {
        app.serverError(w, err)
        return
    }

    // Use the Put() method to add a string value ("Your snippet was saved
    // successfully!") and the corresponding key ("flash") to the session
    // data. Note that 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.
    app.session.Put(r, "flash", "Snippet successfully created!")
    http.Redirect(w, r, fmt.Sprintf("/snippet/%d", id), http.StatusSeeOther)
}
```

Next up we want our `showSnippet` handler to retrieve the confirmation message (if one exists in the session for the current user) and pass it to the template for subsequent display.

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

```
package main

...

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get(":id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err == models.ErrNoRecord {
        app.notFound(w)
        return
    } else if err != nil {

        app.serverError(w, 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.session.PopString(r, "flash")
    flash := app.session.PopString(r, "flash")

    // Pass the flash message to the template.
    app.render(w, r, "show.page.tmpl", &templateData{
        Flash:   flash,
        Snippet: s,
    })
}

...
```

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"

    "alexedwards.net/snippetbox/pkg/forms"
    "alexedwards.net/snippetbox/pkg/models"
)

// Add a Flash field to the templateData struct.
type templateData struct {
    CurrentYear int
    Flash       string
    Form        *forms.Form
    Snippet     *models.Snippet
    Snippets    []*models.Snippet
}
```

```
...
```

And now, we can update our `base.layout.tmpl` file to display the flash message, if one exists.

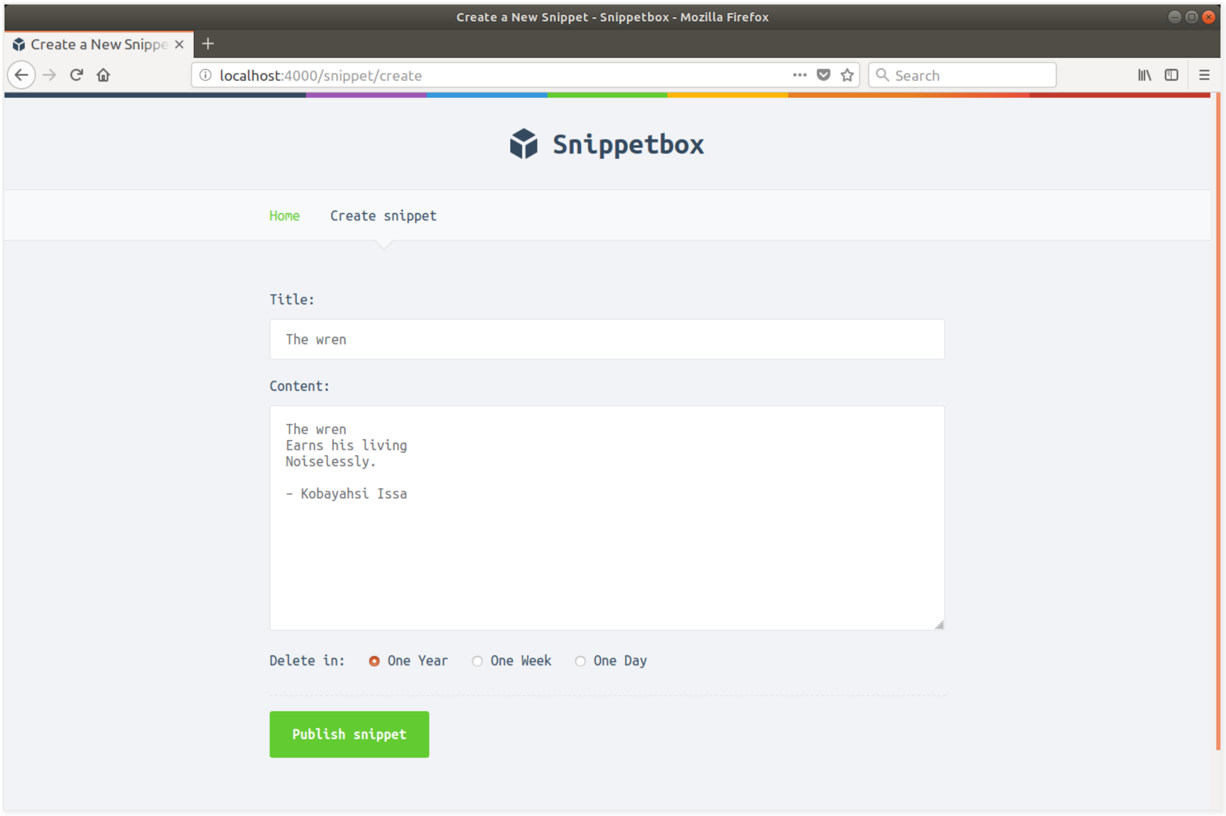
```
File: ui/html/base.layout.tmpl
```

```
{{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-
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ub
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <a href='/'>Home</a>
            <a href='/snippet/create'>Create snippet</a>
        </nav>
        <section>
            {{with .Flash}}
            <div class='flash '>{{.}}</div>
            {{end}}
            {{template "body" .}}
        </section>
        {{template "footer" .}}
        <script src="/static/js/main.js" type="text/javascript"></script>
    </body>
```

```
</html>
{{end}}
```

Remember, the `{{with .Flash}}` block will only be evaluated 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…


Create a New Snippet - Snippetbox - Mozilla Firefox
Create a New Snippe
localhost:4000/snippet/create
Search
Snippetbox
Home
Create snippet

Title:

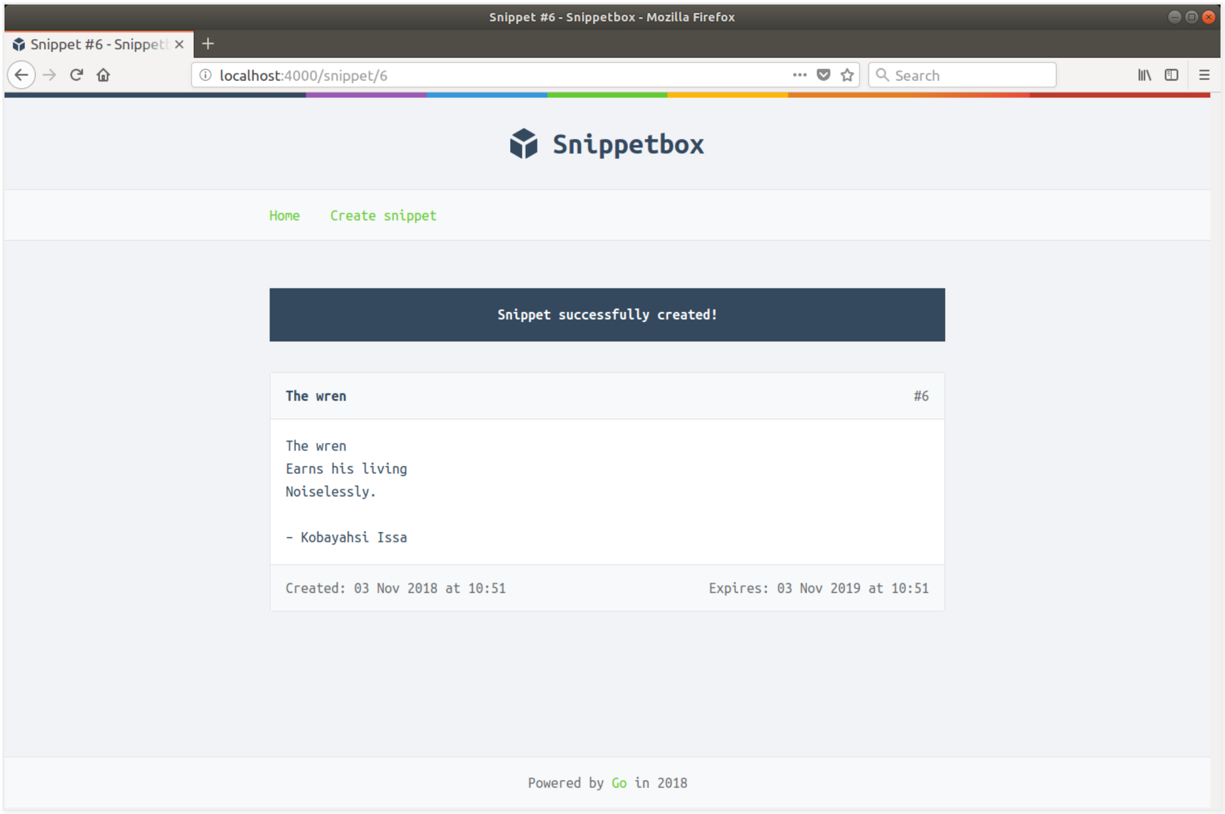
The wren

Content:

```
The wren
Earns his living
Noiselessly.

- Kobayahsi Issa
```

Delete in: [✓ One Year] One Week One Day

And after redirection you should see the flash message now being displayed:



Snippet successfully created!

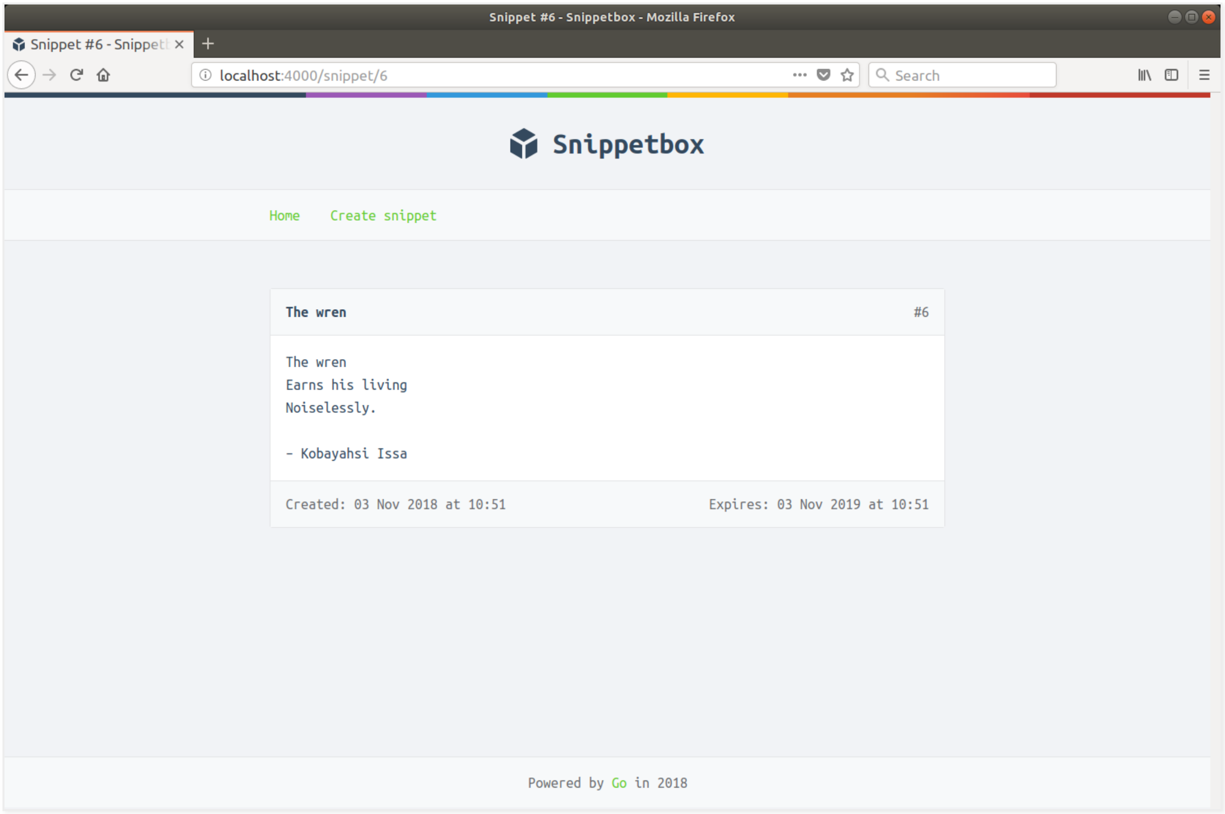
The wren #6

```
The wren
Earns his living
Noiselessly.

- Kobayahsi Issa
```

Created: 03 Nov 2018 at 10:51 Expires: 03 Nov 2019 at 10:51

Powered by Go in 2018

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.



```
The wren                                                    #6

The wren
Earns his living
Noiselessly.

- Kobayahsi Issa

Created: 03 Nov 2018 at 10:51          Expires: 03 Nov 2019 at 10:51
```

Auto-displaying Flash Messages

A little improvement we can make (which will save us some work later in the build) 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 `addDefaultData()` helper method that we made earlier, like so:

File: cmd/web/helpers.go

```
package main

...
```

```
func (app *application) addDefaultData(td *templateData, r *http.Request) *temp
    if td == nil {
        td = &templateData{}
    }
    td.CurrentYear = time.Now().Year()
    // Add the flash message to the template data, if one exists.
    td.Flash = app.session.PopString(r, "flash")
    return td
}

...
```

Making that change means that we no longer need to check for the flash message within the `showSnippet` handler, and the code can be reverted to look like this:

```
package main


...

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get(":id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err == models.ErrNoRecord {
        app.notFound(w)
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }
    }

    app.render(w, r, "show.page.tmpl", &templateData{
        Snippet: s,
    })
}

...
```

Feel free to try running the application again and creating another snippet. You should find that the flash message functionality still works as expected.

Chapter 10.

Security Improvements

In this section of the book we're going to make some improvements to our application so that our data is kept secure during transit and our server is better able to deal with some common types of Denial-of-Service attacks.

You'll learn:

- 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 on our server to mitigate Slowloris and other slow-client attacks.

Chapter 10.1.

Generating a Self-Signed TLS Certificate

HTTPS is essentially HTTP sent across a TLS (*Transport Layer Security*) connection. Because it's sent over a TLS connection the data is encrypted and signed, which helps ensure its privacy and integrity during transit.

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/1.11.4/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:

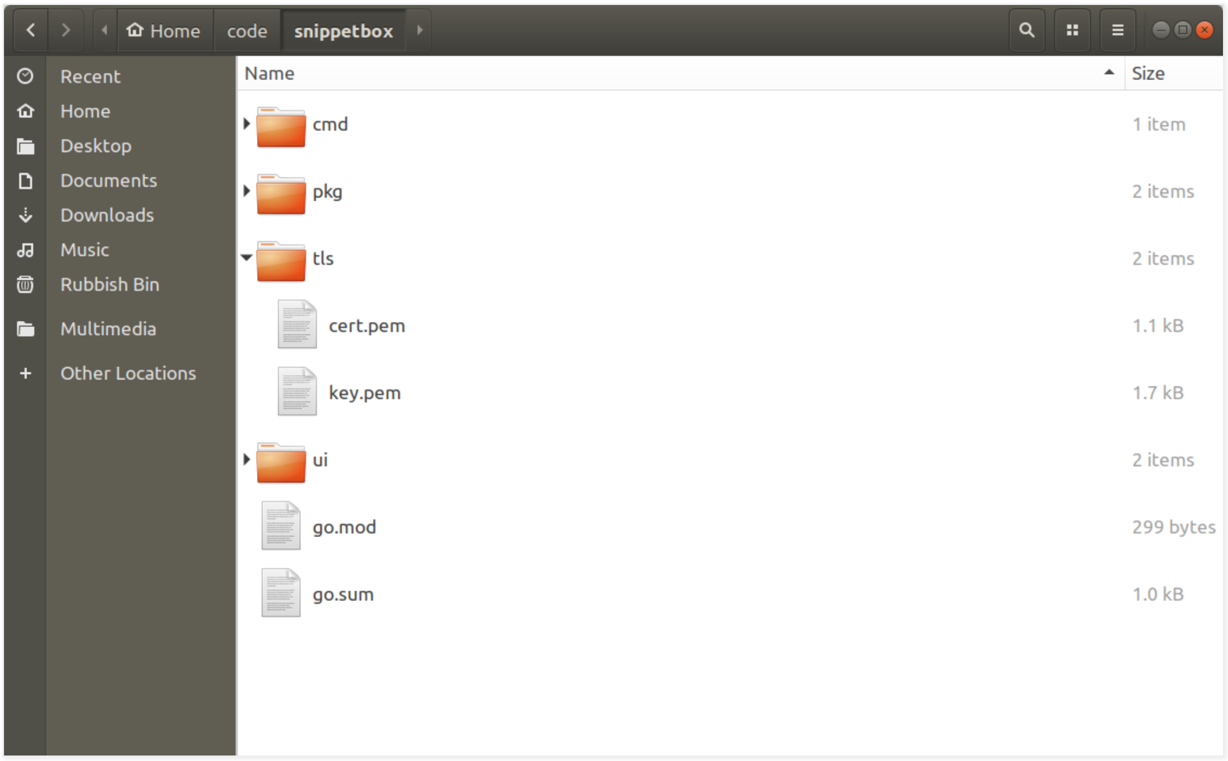
```
$ go run /usr/local/go/src/crypto/tls/generate_cert.go --rsa-bits=2048 --host=localhost
2018/10/16 11:50:14 wrote cert.pem
2018/10/16 11:50:14 wrote key.pem
```

Behind the scenes the `generate_cert.go` tool 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:



Name	Size
cmd	1 item
pkg	2 items
tls	2 items
cert.pem	1.1 kB
key.pem	1.7 kB
ui	2 items
go.mod	299 bytes
go.sum	1.0 kB

And that's it! We've now got a self-signed TLS certificate (and corresponding private key) that we can use during development.

Chapter 10.2.

Running a HTTPS Server

Now 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
```

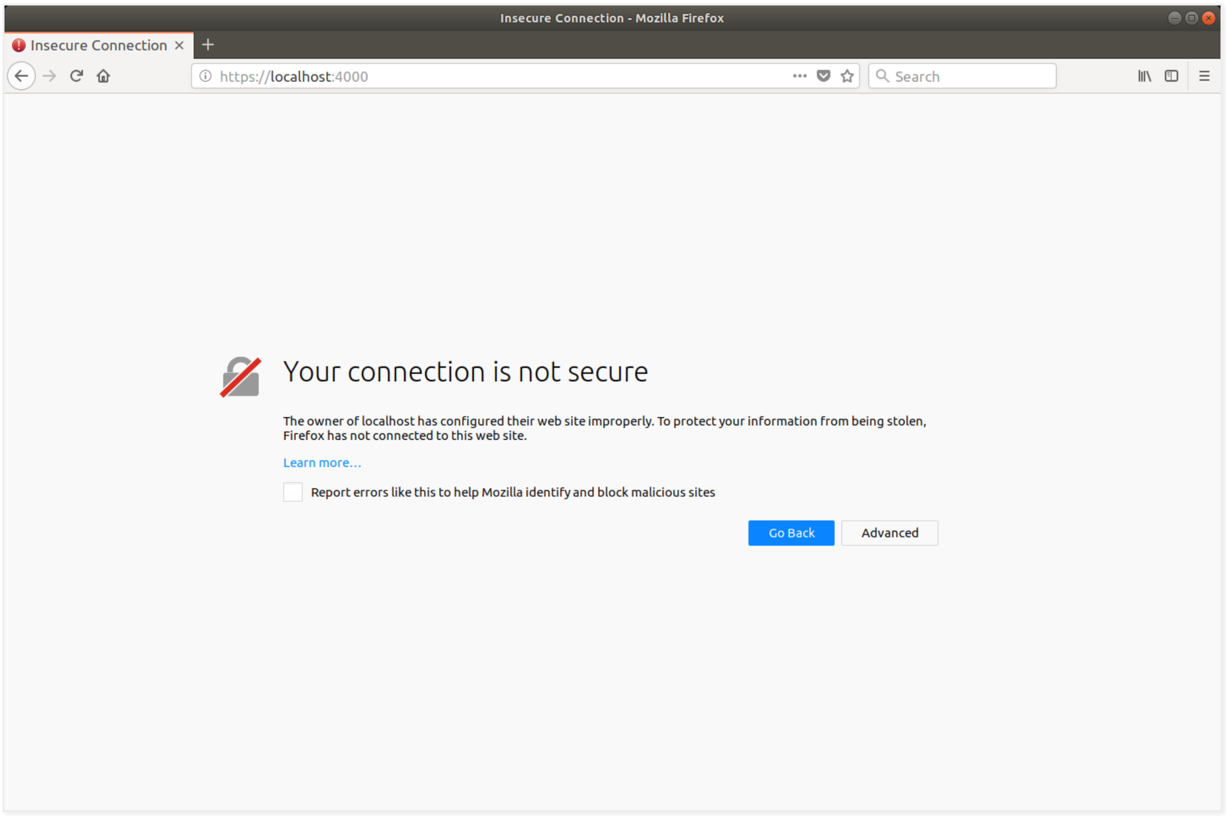
```
package main

...

func main() {
    dsn := flag.String("dsn", "web:pass@/snippetbox?parseTime=true", "MySQL dat
    addr := flag.String("addr", ":4000", "HTTP network address")
    secret := flag.String("secret", "s6Ndh+pPbnzHbS*+9Pk8qGWhTzbpa@ge", "Secret
    flag.Parse()

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfil

    db, err := openDB(*dsn)
    if err != nil {
        errorLog.Fatal(err)
    }
    defer db.Close()
    templateCache, err := newTemplateCache("./ui/html/")
    if err != nil {
        errorLog.Fatal(err)
    }

    session := sessions.New([]byte(*secret))
    session.Lifetime = 12 * time.Hour
    session.Secure = true // Set the Secure flag on our session cookies

    app := &application{
        errorLog:      errorLog,
        infoLog:       infoLog,
        session:       session,
        snippets:      &mysql.SnippetModel{DB: db},
        templateCache: templateCache,
    }

    srv := &http.Server{
        Addr:     *addr,
        ErrorLog: errorLog,
        Handler:  app.routes(),
    }

    infoLog.Printf("Starting server on %s", *addr)
    // Use the ListenAndServeTLS() method to start the HTTPS server. We
    // pass in the paths to the TLS certificate and corresponding private key a
    // the two parameters.
    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    errorLog.Fatal(err)
}

...
```

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/*
INFO    2018/10/16 12:14:28 Starting server on :4000
```

If you open up your web browser and visit https://localhost:4000/ you will probably get a browser warning similar to the screenshot below.

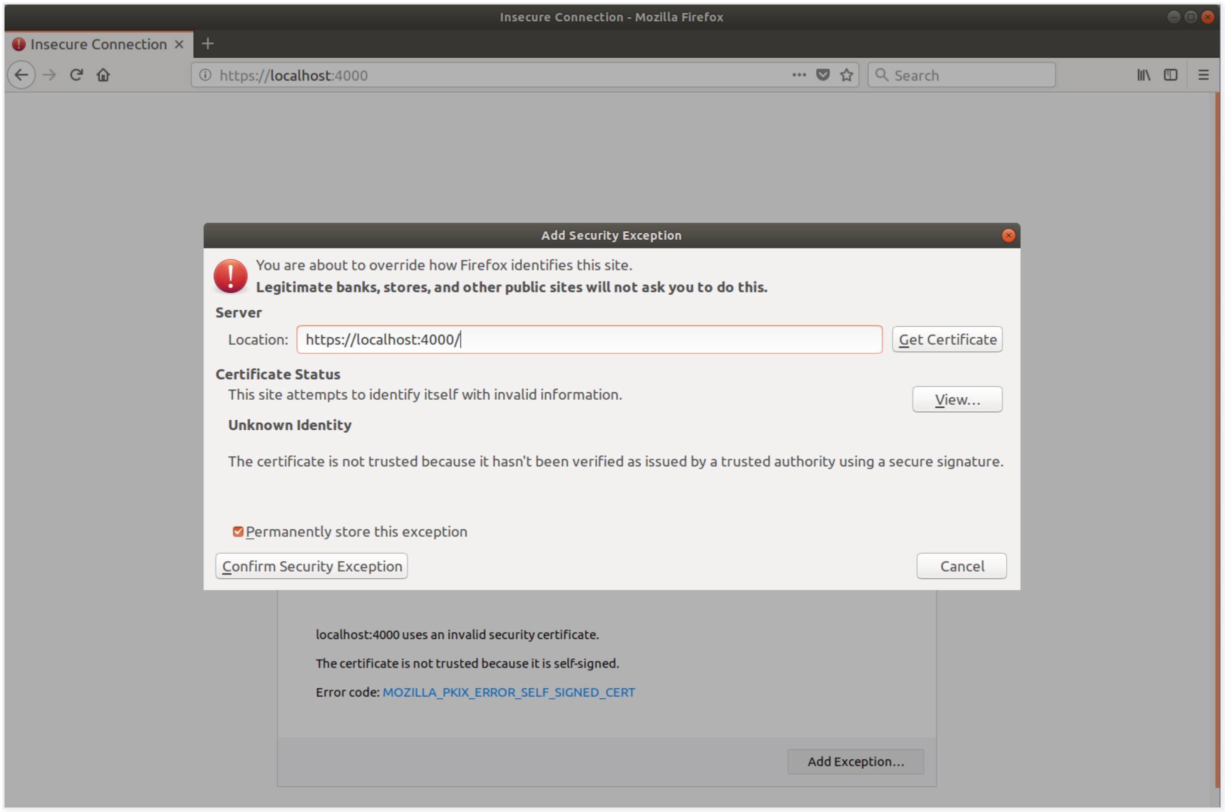

Insecure Connection - Mozilla Firefox
Insecure Connection
https://localhost:4000
Search

Your connection is not secure

The owner of localhost has configured their web site improperly. To protect your information from being stolen, Firefox has not connected to this web site.

Learn more...
Report errors like this to help Mozilla identify and block malicious sites
Go Back
Advanced

If you're using Firefox like me, click "Advanced" then "Add Exception", and in the dialog box that appears click "Confirm Security Exception".


Insecure Connection - Mozilla Firefox

Add Security Exception

You are about to override how Firefox identifies this site.
Legitimate banks, stores, and other public sites will not ask you to do this.

Server
Location: https://localhost:4000/

Certificate Status
This site attempts to identify itself with invalid information.

Unknown Identity

The certificate is not trusted because it hasn't been verified as issued by a trusted authority using a secure signature.

Permanently store this exception

localhost:4000 uses an invalid security certificate.

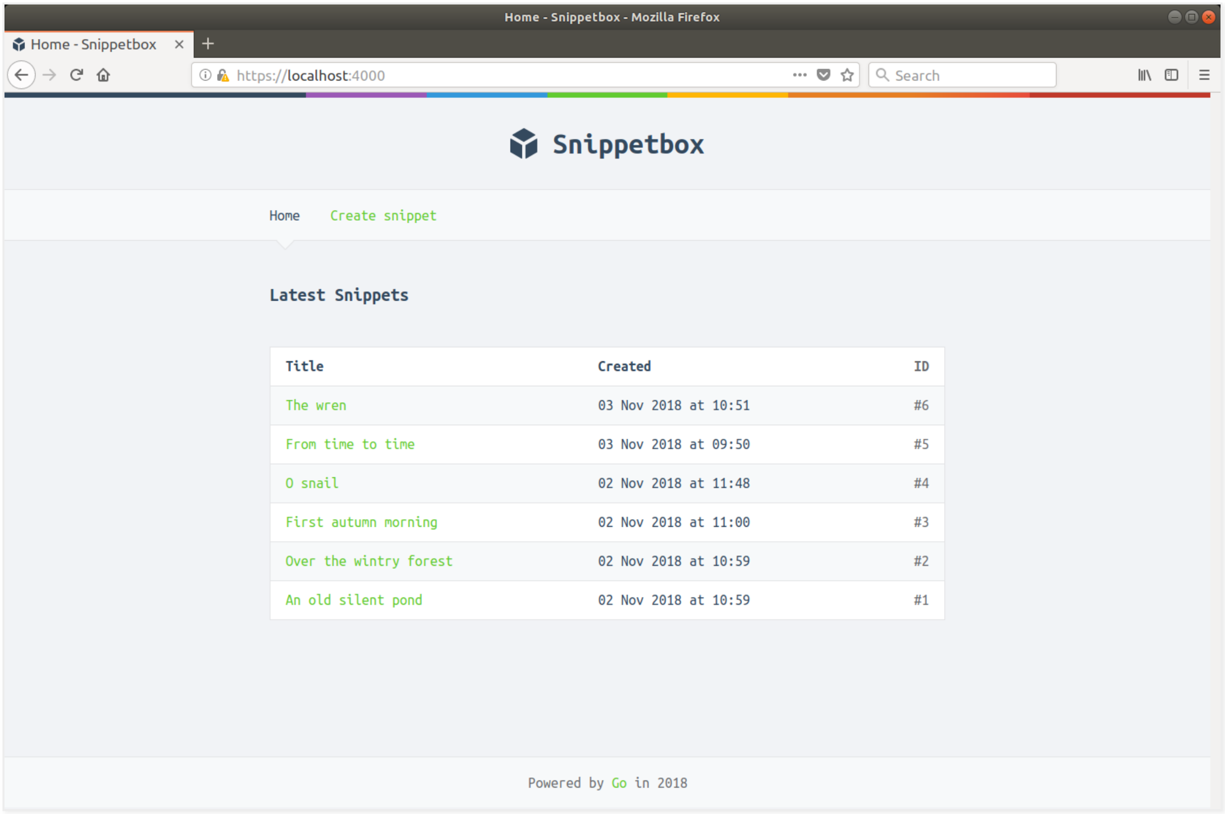
The certificate is not trusted because it is self-signed.

Error code: MOZILLA_PKIX_ERROR_SELF_SIGNED_CERT

If you're using Chrome or Chromium, click "Advanced" and then the "Proceed to localhost" link.

After that the application homepage should appear (although it will still carry a warning in the URL bar because the TLS certificate is self-signed).

In Firefox, it should look a bit like this:

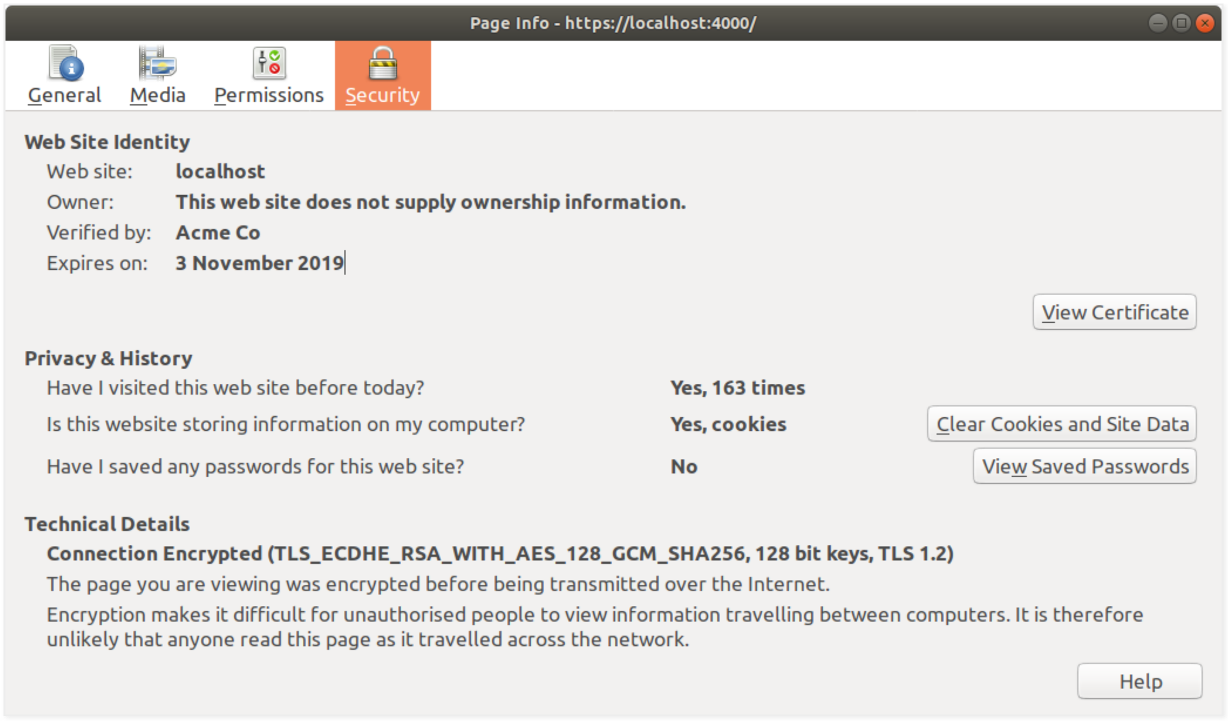


Latest Snippets

Title	Created	ID
The wren	03 Nov 2018 at 10:51	#6
From time to time	03 Nov 2018 at 09:50	#5
O snail	02 Nov 2018 at 11:48	#4
First autumn morning	02 Nov 2018 at 11:00	#3
Over the wintry forest	02 Nov 2018 at 10:59	#2
An old silent pond	02 Nov 2018 at 10:59	#1

Powered by Go in 2018

If you're using Firefox, I recommend pressing `Ctrl+i` to inspect the Page Info for your homepage:



Web Site Identity

Web site: localhost
Owner: This web site does not supply ownership information.
Verified by: Acme Co
Expires on: 3 November 2019

Privacy & History

Have I visited this web site before today? Yes, 163 times
Is this website storing information on my computer? Yes, cookies
Have I saved any passwords for this web site? No

Technical Details

Connection Encrypted (TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, 128 bit keys, TLS 1.2)
The page you are viewing was encrypted before being transmitted over the Internet.
Encryption makes it difficult for unauthorised people to view information travelling between computers. It is therefore unlikely that anyone read this page as it travelled across the network.

The 'Technical Details' section here confirms that our connection is encrypted and working as expected.

In my case, I can see that TLS version 1.2 is being used, and the *cipher suite* for my HTTPS connection is `TLS_ECDHE_RSA_WITH_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, 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, it won’t work. Our server will write the bytes `15 03 01 00 02 02 0A` to the underlying TCP connection, which essentially is TLS-speak for “I don’t understand”.

You should also see a corresponding log message in your terminal relating to this.

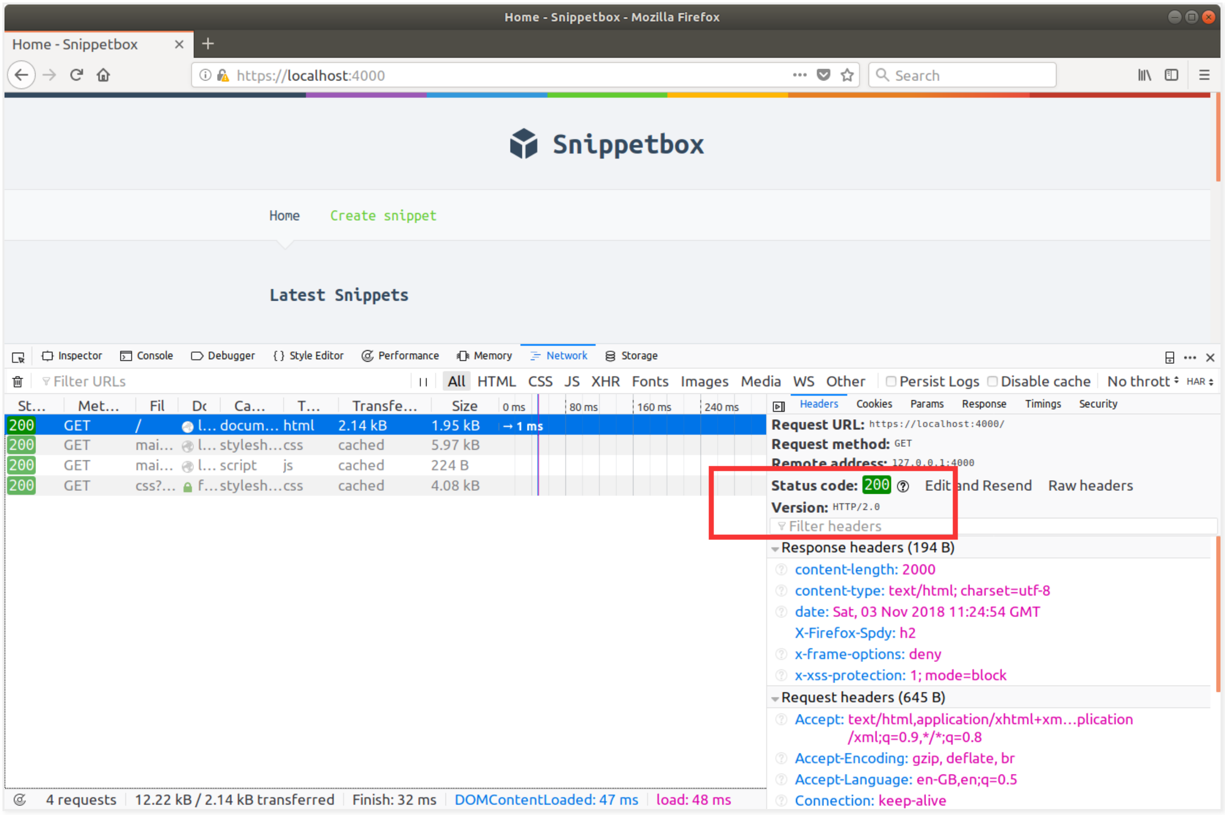
```
ERROR	2018/10/16 12:36:53 server.go:2977: http: TLS handshake error from [::1
```

HTTP/2 Connections

A big plus of using HTTPS is that — if a client supports HTTP/2 connections — Go’s HTTPS server will automatically upgrade the connection to use HTTP/2.

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 has been implemented behind the scenes 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.



St…	Met…	Fil	Dc	Ca…	T…	Transfe…	Size	0 ms 80 ms 160 ms 240 ms
200	GET	/	l…	docum…	html	2.14 kB	1.95 kB	→ 1 ms
200	GET	mai…	l…	stylesh…	css	cached	5.97 kB	
200	GET	mai…	l…	script	js	cached	224 B	
200	GET	css?…	f…	stylesh…	css	cached	4.08 kB	

4 requests | 12.22 kB / 2.14 kB transferred | Finish: 32 ms | DOMContentLoaded: 47 ms | load: 48 ms

Request URL: `https://localhost:4000/`
Request method: GET
Remote address: 127.0.0.1:4000
Status code: 200
Version: HTTP/2.0

Response headers (194 B)
content-length: 2000
content-type: text/html; charset=utf-8
date: Sat, 03 Nov 2018 11:24:54 GMT
X-Firefox-Spdy: h2
x-frame-options: deny
x-xss-protection: 1; mode=block

Request headers (645 B)
Accept: text/html,application/xhtml+xm…plication
/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en;q=0.5
Connection: keep-alive

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-r--r-- 1 alex alex 1090 Oct 16 11:50 cert.pem
-rw------- 1 alex alex 1679 Oct 16 11:50 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:

```
$ echo 'tls/' >> .gitignore
```

Chapter 10.3.

Configuring HTTPS Settings

Go has pretty good default settings for its HTTPS server, but there are a couple of improvements and optimizations that we can make.

If you're new to HTTPS and TLS then I recommend taking some time to understand the principles behind TLS before you start changing the default settings. To help, I've included a high-level summary of how TLS works in the appendix.

But, that caveat aside, there are a couple of tweaks that it's almost always a good idea to make.

To change the default TLS settings we need to do two things:

- First, we need to create a `tls.Config` struct which contains the non-default TLS settings that we want to use.
- Second, we need to add this to our `http.Server` struct before we start the server.

I'll demonstrate:

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

```
package main

```

```go
import (
    "crypto/tls" // New import
    "database/sql"
    "flag"
    "html/template"
    "log"
    "net/http"
    "os"
    "time"

    "alexedwards.net/snippetbox/pkg/models/mysql"

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

...

func main() {
    ...

    app := &application{
        errorLog:      errorLog,
        infoLog:       infoLog,
        session:       session,
        snippets:      &mysql.SnippetModel{DB: db},
        templateCache: templateCache,
    }

    // Initialize a tls.Config struct to hold the non-default TLS settings we w
    // the server to use.
    tlsConfig := &tls.Config{
        PreferServerCipherSuites: true,
        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,
        ErrorLog: errorLog,
        Handler:  app.routes(),
        TLSConfig: tlsConfig,
    }

    infoLog.Printf("Starting server on %s", *addr)
    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    errorLog.Fatal(err)
}

...
```

In the `tls.Config` struct we've used two settings:

- The `tls.Config.PreferServerCipherSuites` field controls whether the HTTPS connection should use Go's favored cipher suites or the user's favored cipher suites. By setting this to true — like we have above — Go's favored cipher suites are given preference and we help increase the likelihood that a strong cipher suite which also supports forward secrecy is used.

- The `tls.Config.CurvePreferences` field lets us specify which *elliptic curves* should be given preference during the TLS handshake. Go supports a few elliptic curves, but as of Go 1.11 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.

Additional Information

Restricting Cipher Suites

The full range of cipher suites that Go supports are defined in the `crypto/tls` package constants.

For some applications, it may be desirable to limit your HTTPS server to only support some of these 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{
    PreferServerCipherSuites: true,
    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,
    },
}
```

Note: 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.

Also note: If you have `PreferServerCipherSuites` set to true then the order of elements in the `CipherSuites` field is important — the cipher suites which come first in the slice will be preferred.

TLS Versions

TLS versions are also defined as constants in the `crypto/tls` package. By default, Go's HTTPS server supports TLS versions 1.0, 1.1 and 1.2.

But you can configure the minimum and maximum TLS versions via the `tls.Config.MinVersion` and `MaxVersion` fields. For instance, if you know that all computers in your user base support TLS 1.2 then you may wish to change your server to only support TLS 1.2. For example:

```go
tlsConfig := &tls.Config{
    MinVersion: tls.VersionTLS12,
    MaxVersion: tls.VersionTLS12,
}
```

Chapter 10.4.

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
```

```go
package main

...

func main() {
    ...

    tlsConfig := &tls.Config{
        PreferServerCipherSuites: true,
        CurvePreferences:         []tls.CurveID{tls.X25519, tls.CurveP256},
    }

    srv := &http.Server{
        Addr:         *addr,

        ErrorLog:     errorLog,
        Handler:      app.routes(),
        TLSConfig:    tlsConfig,
        // Add Idle, Read and Write timeouts to the server.
        IdleTimeout:  time.Minute,
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    infoLog.Printf("Starting server on %s", *addr)
    infoLog.Printf("Starting server on %s", addr)
    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    errorLog.Fatal(err)
}

...
```

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.

IdleTimeout

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 handshake.

Also by default, Go will automatically close keep-alive connections after 3 minutes of inactivity. This helps to clear-up connections where the user has unexpectedly disappeared (e.g. due to a power cut client-side).

There is no way to increase this cut-off above 3 minutes (unless you roll your own `net.Listener`), but you can reduce it via the `IdleTimeout` setting. In our case, we've set it to 1 minute, which means that all keep-alive connections will be automatically closed after 1 minute of inactivity.

ReadTimeout

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.

One important thing. 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.

WriteTimeout

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 tham 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

ReadHeaderTimeout

The `http.Server` object 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`.

Warning: The `ReadHeaderTimeout` setting also affects the `IdleTimeout` behavior. Specifically, if you set `ReadHeaderTimeout` but don't set `ReadTimeout` *and* don't set `IdleTimeout`, then `IdleTimeout` will default to using the same setting as `ReadHeaderTimeout`. Again, it's safest (and clearest) to simply get into the habit of setting an explicit `IdleTimeout`.

MaxHeaderBytes

The `http.Server` object also provides 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:

```go
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.

Chapter 11.

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.

This begs the question: *how will we know if a user has logged in or not?*

For our application, the process will work 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 then check the database to see if the credentials they entered match one of the users in the `users` table. If there's a match, we add the relevant `id` value for the user to their session data, using the key `"userID"`.

4. When we receive any subsequent requests, we can check the user's session data for a `"userID"` value. If it exists, we know that the user has 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 `"userID"` 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 securely in your database using Bcrypt.
- 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.

Chapter 11.1.

Routes Setup

Let's begin this section by adding five new routes to our application, so that it looks like this:

Method	Pattern	Handler	Action
GET	/	home	Display the home page
GET	/snippet?id=1	showSnippet	Display a specific snippet
GET	/snippet/create	createSnippetForm	Display the new snippet form
POST	/snippet/create	createSnippet	Create a new snippet
GET	/user/signup	signupUserForm	Display the user signup form
POST	/user/signup	signupUser	Create a new user
GET	/user/login	loginUserForm	Display the user login form
POST	/user/login	loginUser	Authenticate and login the user
POST	/user/logout	logoutUser	Logout the user
GET	/static/	http.FileServer	Serve a specific static file

Notice how the new state-changing actions — `signupUser`, `loginUser` and `logoutUser` — are all using `POST` requests, not `GET`?

Open up your `handlers.go` file and add placeholders for the five new handler functions as follows:

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

```go
package main

...

func (app *application) signupUserForm(w http.ResponseWriter, r *http.Request)
    fmt.Fprintln(w, "Display the user signup form...")
}

func (app *application) signupUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Create a new user...")
}

func (app *application) loginUserForm(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Display the user login form...")
}

func (app *application) loginUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Authenticate and login the user...")
}

func (app *application) logoutUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Logout the user...")
}
```

Then when that's done, let's create the corresponding routes in our `routes.go` file:

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

```
package main

...

func (app *application) routes() http.Handler {
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea
    dynamicMiddleware := alice.New(app.session.Enable)

    mux := pat.New()
    mux.Get("/", dynamicMiddleware.ThenFunc(app.home))
    mux.Get("/snippet/create", dynamicMiddleware.ThenFunc(app.createSnippetForm
    mux.Post("/snippet/create", dynamicMiddleware.ThenFunc(app.createSnippet))
    mux.Get("/snippet/:id", dynamicMiddleware.ThenFunc(app.showSnippet))

    // Add the five new routes.
    mux.Get("/user/signup", dynamicMiddleware.ThenFunc(app.signupUserForm))
    mux.Post("/user/signup", dynamicMiddleware.ThenFunc(app.signupUser))
    mux.Get("/user/login", dynamicMiddleware.ThenFunc(app.loginUserForm))
    mux.Post("/user/login", dynamicMiddleware.ThenFunc(app.loginUser))
    mux.Post("/user/logout", dynamicMiddleware.ThenFunc(app.logoutUser))

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

    return standardMiddleware.Then(mux)
}
```

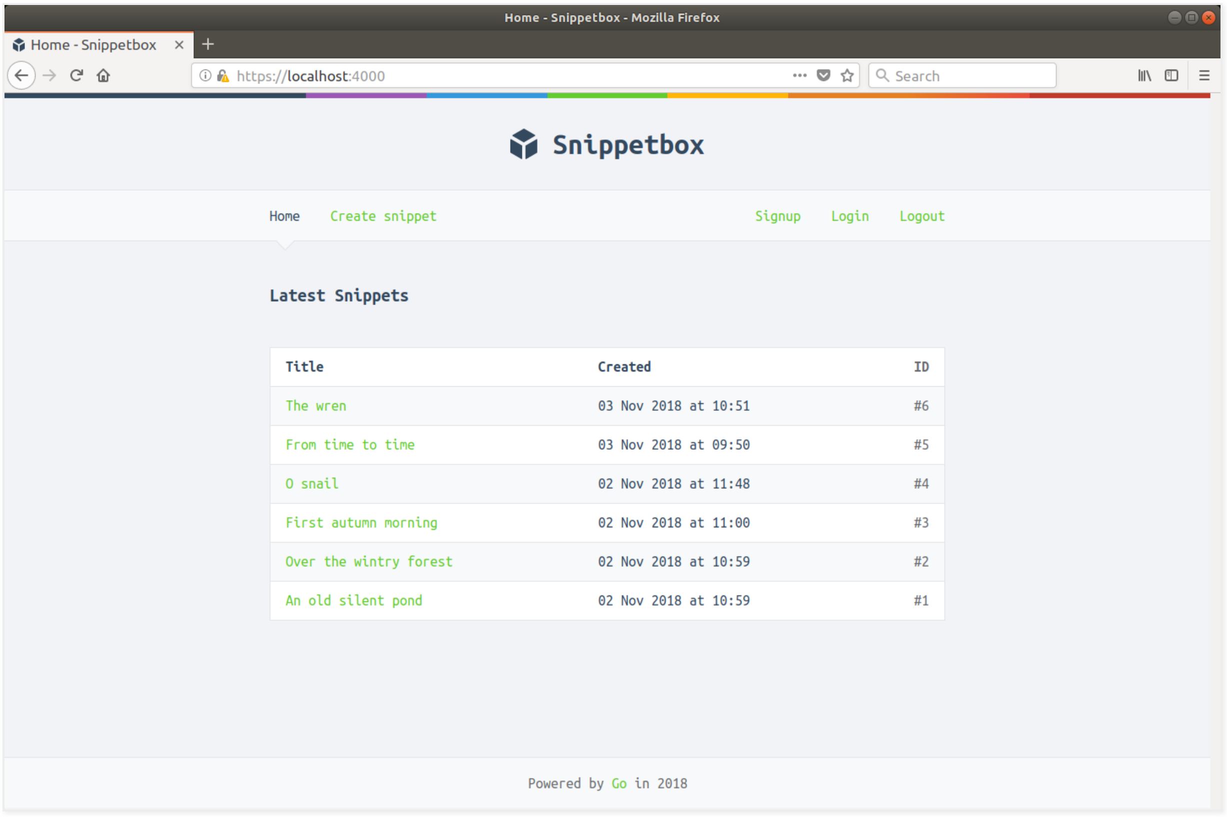
Finally, we'll also need to update the `base.layout.tmpl` file to add navigation items for the new pages:

```
File: ui/html/base.layout.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-
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ub
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <!-- Update the navigation to include signup, login and
            logout links -->
            <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>
        <section>
            {{with .Flash}}
            <div class='flash '>{{.}}</div>
            {{end}}
            {{template "body" .}}
        </section>
        {{template "footer" .}}
        <script src="/static/js/main.js" type="text/javascript"></script>
    </body>
</html>
{{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:

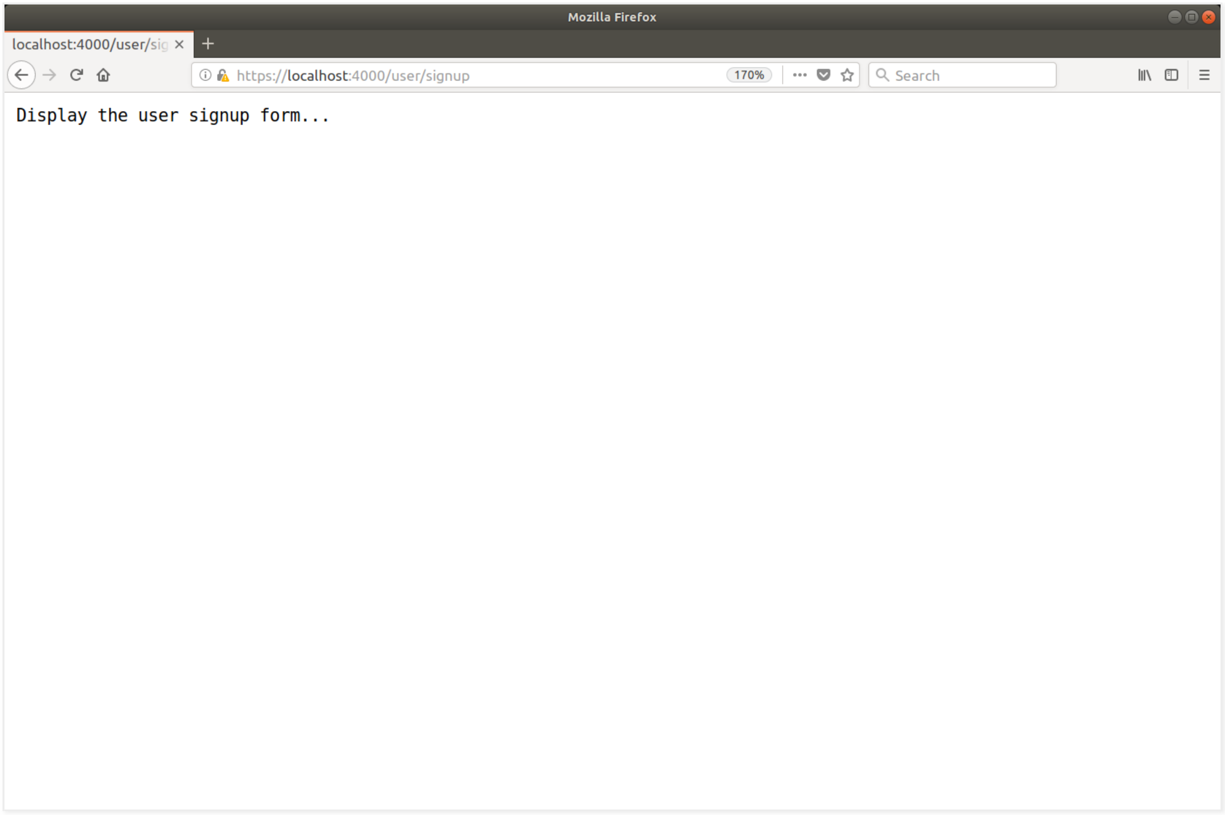


Latest Snippets

Title	Created	ID
The wren	03 Nov 2018 at 10:51	#6
From time to time	03 Nov 2018 at 09:50	#5
O snail	02 Nov 2018 at 11:48	#4
First autumn morning	02 Nov 2018 at 11:00	#3
Over the wintry forest	02 Nov 2018 at 10:59	#2
An old silent pond	02 Nov 2018 at 10:59	#1

Powered by Go in 2018

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 the user signup form...
```

Chapter 11.2.

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).

- I’ve set the type of the `hashed_password` field to `CHAR(60)`. This is because we’ll be storing hashes of the user passwords in the database — not the passwords themselves — and the hashed versions will always be exactly 60 characters long.

- I’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: Duplicate 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 `pkg/models/model.go` file that you created earlier and add a new `User` struct to hold the data for each user, plus a couple of new error types:

```
File: pkg/models/models.go
```

```go
package models

import (
    "errors"
    "time"
)
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")
)

type Snippet struct {
    ID      int

    Title   string
    Content string
    Created time.Time
    Expires time.Time
}

// Define a new User type. 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
}
```

Now that the types have been set up, we need to make the actual database model. Create a new file at `pkg/models/mysql/users.go`…

```
$ cd $HOME/code/snippetbox
$ touch pkg/models/mysql/users.go
```

…And then create a new `UserModel` type with some placeholder functions like so:

```
File: pkg/models/mysql/users.go
```

```go
package mysql

import (
    "database/sql"

    "alexedwards.net/snippetbox/pkg/models"
)

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 Get method to fetch details for a specific user based
// on their user ID.
func (m *UserModel) Get(id int) (*models.User, error) {
    return nil, 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
```

```go
package main

...

// Add a new users field to the application struct.
type application struct {
    errorLog      *log.Logger
    infoLog       *log.Logger
    session       *sessions.Session
    snippets      *mysql.SnippetModel
    templateCache map[string]*template.Template
    users         *mysql.UserModel
}

func main() {
    ...

    session := sessions.New([]byte(*secret))
    session.Lifetime = 12 * time.Hour
    session.Secure = true

    // Initialize a mysql.UserModel instance and add it to the application
    // dependencies.
    app := &application{
        errorLog:      errorLog,
        infoLog:       infoLog,
        session:       session,
        snippets:      &mysql.SnippetModel{DB: db},
        templateCache: templateCache,
        users:         &mysql.UserModel{DB: db},
    }
    tlsConfig := &tls.Config{
        PreferServerCipherSuites: true,
        CurvePreferences:         []tls.CurveID{tls.X25519, tls.CurveP256},
    }

    srv := &http.Server{
        Addr:         *addr,
        ErrorLog:     errorLog,
        Handler:      app.routes(),
        TLSConfig:    tlsConfig,
        IdleTimeout:  time.Minute,
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    infoLog.Printf("Starting server on %s", *addr)
    err = srv.ListenAndServeTLS("./tls/cert.pem", "./tls/key.pem")
    errorLog.Fatal(err)
}

...
```

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.

Chapter 11.3.

User Signup and Password Encryption

Before we can log in any users to our Snippetbox application we first 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/signup.page.tmpl` file containing the following markup.

```
$ cd $HOME/code/snippetbox
$ touch ui/html/signup.page.tmpl
```

File: ui/html/signup.page.tmpl

```
{{template "base" .}}

{{define "title"}}Signup{{end}}

{{define "body"}}
<form action='/user/signup' method='POST' novalidate>
    {{with .Form}}
        <div>
            <label>Name:</label>
            {{with .Errors.Get "name"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='text' name='name' value='{{.Get "name"}}'>
        </div>
        <div>
            <label>Email:</label>
            {{with .Errors.Get "email"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='email' name='email' value='{{.Get "email"}}'>
        </div>
        <div>
            <label>Password:</label>
            {{with .Errors.Get "password"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='password' name='password'>
        </div>
        <div>

            <input type='submit' value='Signup'>
        </div>
    {{end}}
</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).

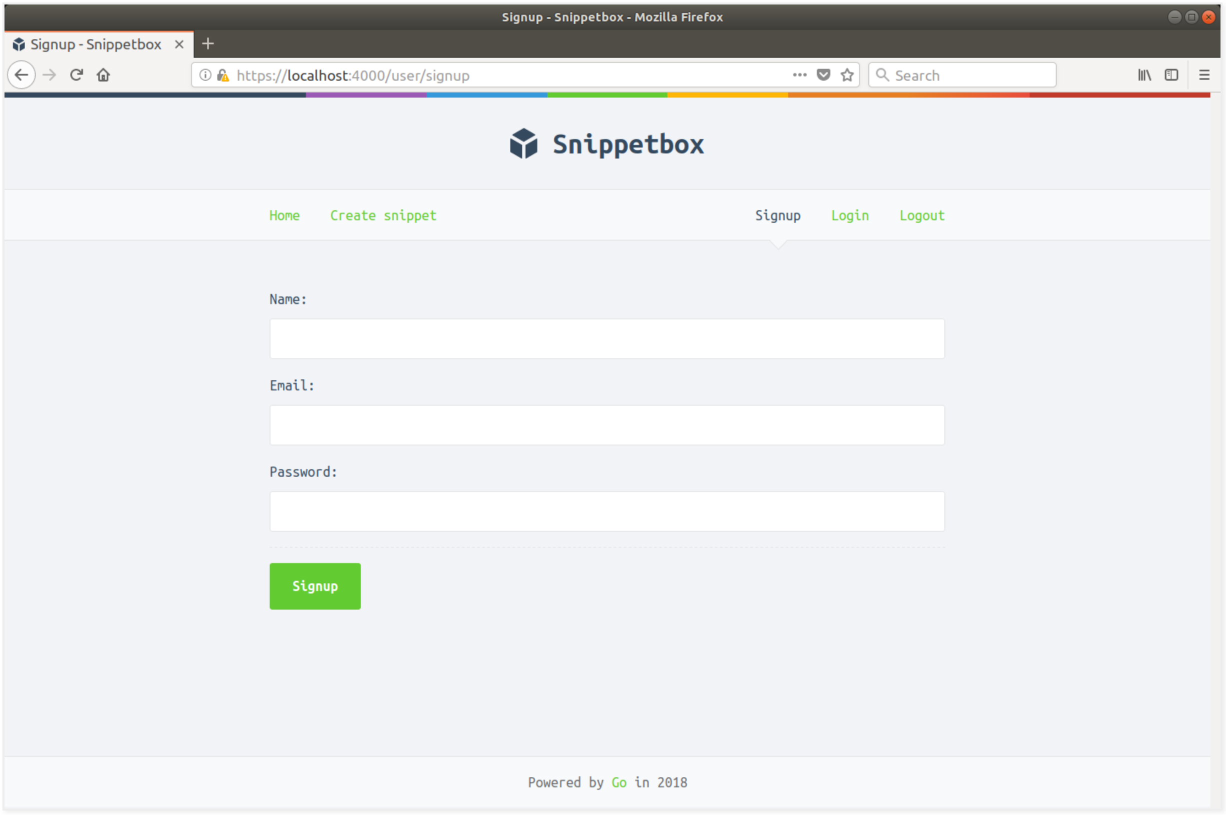
Importantly, notice that we're not re-displaying the password if the form fails validation — we don't want there to be any risk of the browser (or other intermediary) caching the plain-text password entered by the user.

Then let's hook this up to the `signupUserForm` handler like so:

File: cmd/web/handlers.go

```go
package main

...

func (app *application) signupUserForm(w http.ResponseWriter, r *http.Request)
    app.render(w, r, "signup.page.tmpl", &templateData{
        Form: forms.New(nil),
    })
}

...
```

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 `signupUser` handler that we made earlier.

The first task of this handler will be to validate the data and 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 user's 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 10 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 `pkg/forms/form.go` file and creating two helper new methods — `MinLength()` and `MatchesPattern()` — along with a regular expression for sanity checking an email address.

```
File: pkg/forms/form.go
```

```go
package forms

import (
    "fmt"
    "net/url"
    "regexp" // New import
    "strings"
    "unicode/utf8"
)
```

```
// Use the regexp.MustCompile() function to parse a pattern and compile a
// regular expression for sanity checking the format of an email address.
// This returns a *regexp.Regexp object, or panics in the event of an error.
// Doing this once at runtime, and storing the compiled regular expression
// object in a variable, is more performant than re-compiling the pattern with
// every request.
var EmailRX = regexp.MustCompile("^[a-zA-Z0-9.!#$%&'*+\\/=?^_`{|}~-]+@[a-zA-Z0-

type Form struct {
    url.Values
    Errors errors
}

...

// Implement a MinLength method to check that a specific field in the form
// contains a minimum number of characters. If the check fails then add the
// appropriate message to the form errors.
func (f *Form) MinLength(field string, d int) {
    value := f.Get(field)
    if value == "" {
        return
    }
    if utf8.RuneCountInString(value) < d {
        f.Errors.Add(field, fmt.Sprintf("This field is too short (minimum is %d
    }
}

// Implement a MatchesPattern method to check that a specific field in the form
// matches a regular expression. If the check fails then add the
// appropriate message to the form errors.
func (f *Form) MatchesPattern(field string, pattern *regexp.Regexp) {
    value := f.Get(field)
    if value == "" {
        return
    }
    if !pattern.MatchString(value) {
        f.Errors.Add(field, "This field is invalid")
    }
}

func (f *Form) Valid() bool {
    return len(f.Errors) == 0
}
```

Important: The pattern we're using to sanity check email address format is the one currently recommended by the W3C and Web Hypertext Application Technology Working Group. If you're reading this book in PDF format or on a narrow device, and can't see the entire line, then you can find the complete pattern here and here. But if there's an alternative pattern that you prefer to use for email address sanity checking, then feel free to swap it in instead.

Another (also important) note: Because, in our code, the 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 for the pattern because it 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.

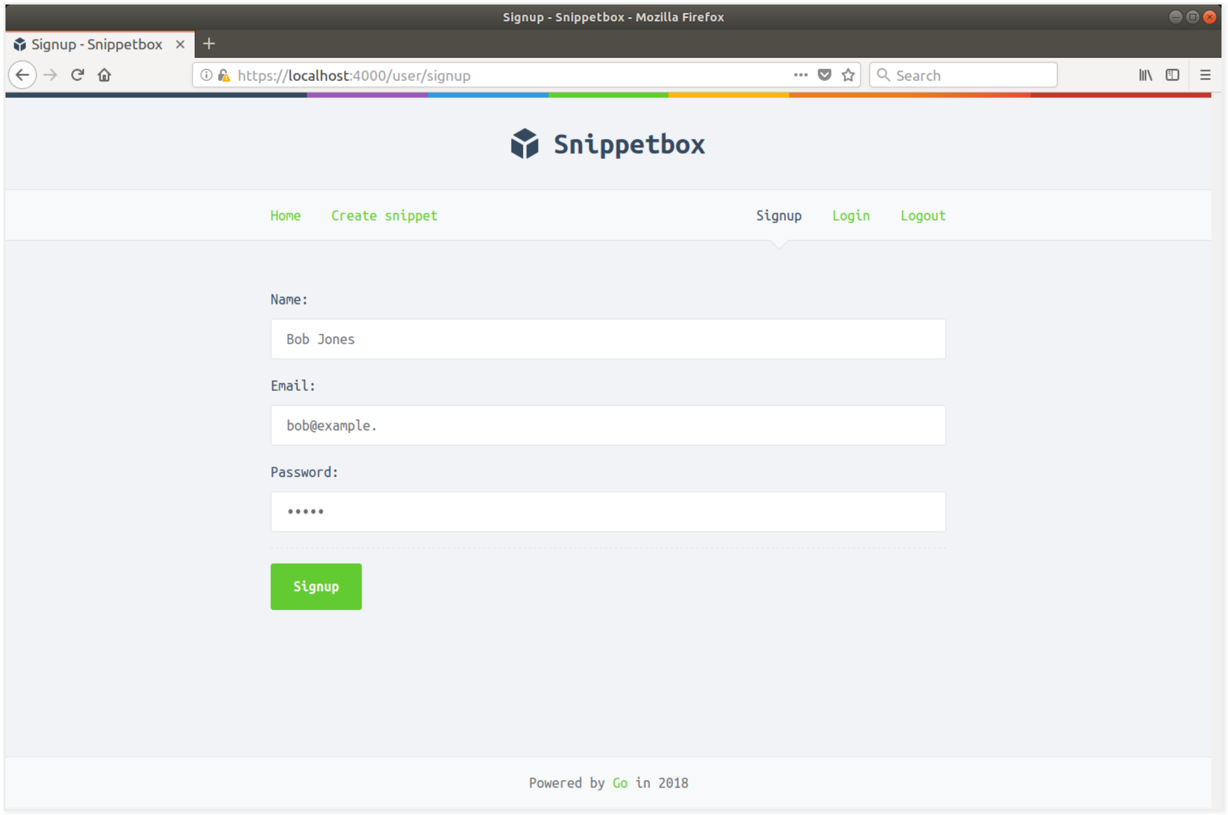
Head over to your `handlers.go` file and add some code to process the form and run the validation checks like so:

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

```
package main
```

```
...

func (app *application) signupUser(w http.ResponseWriter, r *http.Request) {
    // Parse the form data.
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    // Validate the form contents using the form helper we made earlier.
    form := forms.New(r.PostForm)
    form.Required("name", "email", "password")
    form.MatchesPattern("email", forms.EmailRX)
    form.MinLength("password", 10)

    // If there are any errors, redisplay the signup form.
    if !form.Valid() {
        app.render(w, r, "signup.page.tmpl", &templateData{Form: form})
        return
    }

    // Otherwise send a 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:



Name:

Bob Jones

Email:

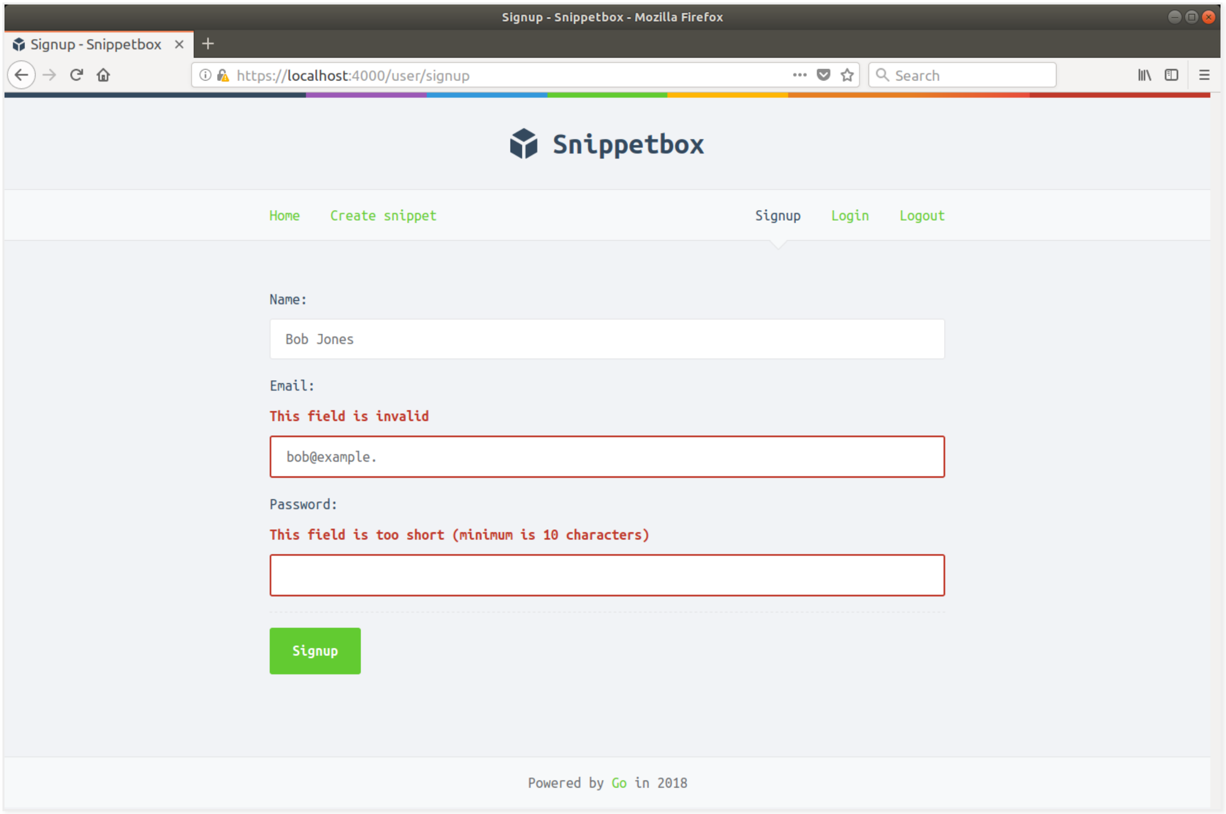
bob@example.

Password:

•••••

Powered by Go in 2018

And if you try to submit it, you should see the appropriate validation failures returned like so:



Name:

Bob Jones

Email:

This field is invalid

bob@example.

Password:

This field is too short (minimum is 10 characters)

Powered by Go in 2018

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 good implementations of all 3 algorithms, but a plus-point of the brcypt implementation is that it includes some helper functions specifically designed for hashing and checking passwords.

Note: If you’ve been following along, the bcrypt package should have already been downloaded to your computer because it’s part of the wider golang.org/x/crypto package that was installed as a dependency along with the golangcollege/sessions package earlier in the book.

There are two functions in the bcrypt package 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)
```

The second parameter that we pass in here indicates the *cost*, which is represented by an integer between 4 and 31. The code above uses a cost of 12, which means that that 4096 (2^12) bcrypt iterations will be used to hash the password. I wouldn’t recommend using less than that. This function will return a 60-character long hash which looks a bit like this: `$2a$12$NuTjWXm3KKntReFwyBVHyuf/to.HEwTy.eS206TNfkGfr6HzGJSWG`.

It’s worth pointing out that the `bcrypt.GenerateFromPassword()` function also adds a random salt to the password to help avoid rainbowtable attacks.

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.eS206TNfkGfr6GzGJSWG")
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 `pkg/models/mysql/users.go` file and update it to include the following code:

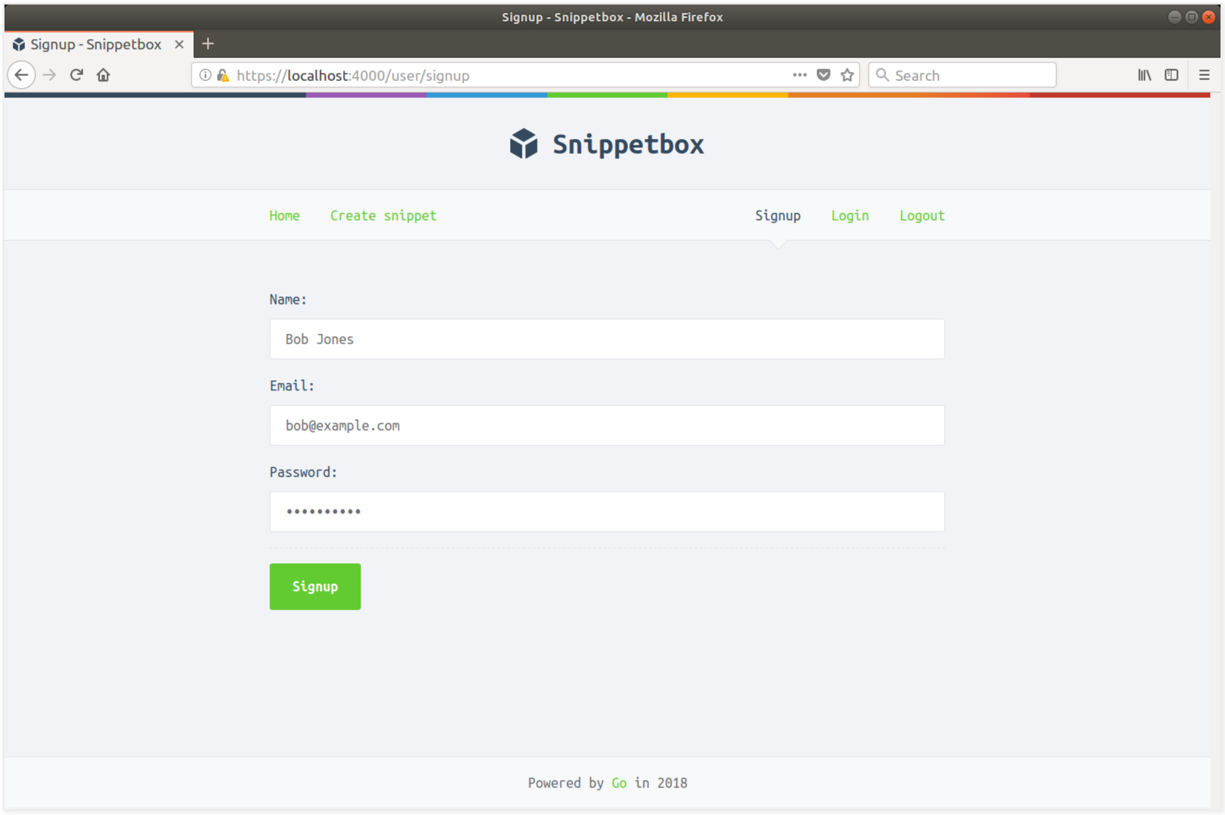
```
File: pkg/models/mysql/users.go
```

```go
package mysql

import (
    "database/sql"
    "strings" // New import

    "alexedwards.net/snippetbox/pkg/models"

    "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. If this returns an error, we try to type assert
    // it to a *mysql.MySQLError object so we can check if the error number is
    // 1062 and, if it is, we also check whether or not the error relates to
    // our users_uc_email key by checking the contents of the message string.
    // If it does, we return an ErrDuplicateEmail error. Otherwise, we just
    // return the original error (or nil if everything worked).
    _, err = m.DB.Exec(stmt, name, email, string(hashedPassword))
    if err != nil {
        if mysqlErr, ok := err.(*mysql.MySQLError); ok {
            if mysqlErr.Number == 1062 && strings.Contains(mysqlErr.Message, "u
                return models.ErrDuplicateEmail
            }
        }
    }
    return err
}

...
```

We can then finish this all off by updating the `signupUser` handler like so:

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

package main

...

func (app *application) signupUser(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    form := forms.New(r.PostForm)
    form.Required("name", "email", "password")
    form.MatchesPattern("email", forms.EmailRX)
    form.MinLength("password", 10)

    if !form.Valid() {
        app.render(w, r, "signup.page.tmpl", &templateData{Form: form})
        return
    }

    // Try to create a new user record in the database. If the email already ex
    // add an error message to the form and re-display it.
    err = app.users.Insert(form.Get("name"), form.Get("email"), form.Get("passw
    if err == models.ErrDuplicateEmail {
        form.Errors.Add("email", "Address is already in use")
        app.render(w, r, "signup.page.tmpl", &templateData{Form: form})
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }

    // Otherwise add a confirmation flash message to the session confirming tha
    // their signup worked and asking them to log in.
    app.session.Put(r, "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!



Name:

Bob Jones

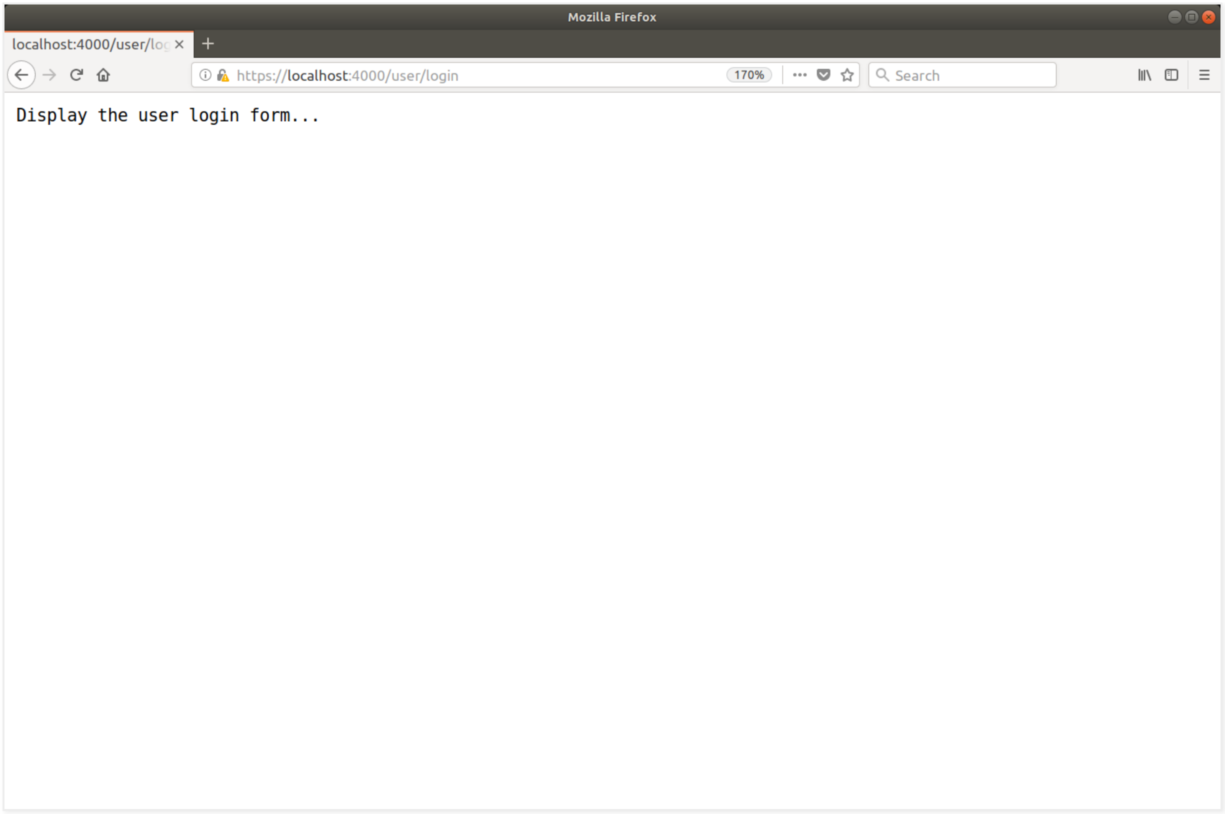
Email:

bob@example.com

Password:

••••••••••

If everything works correctly, you should find that your browser redirects you to https://localhost:4000/user/login after you submit the form.

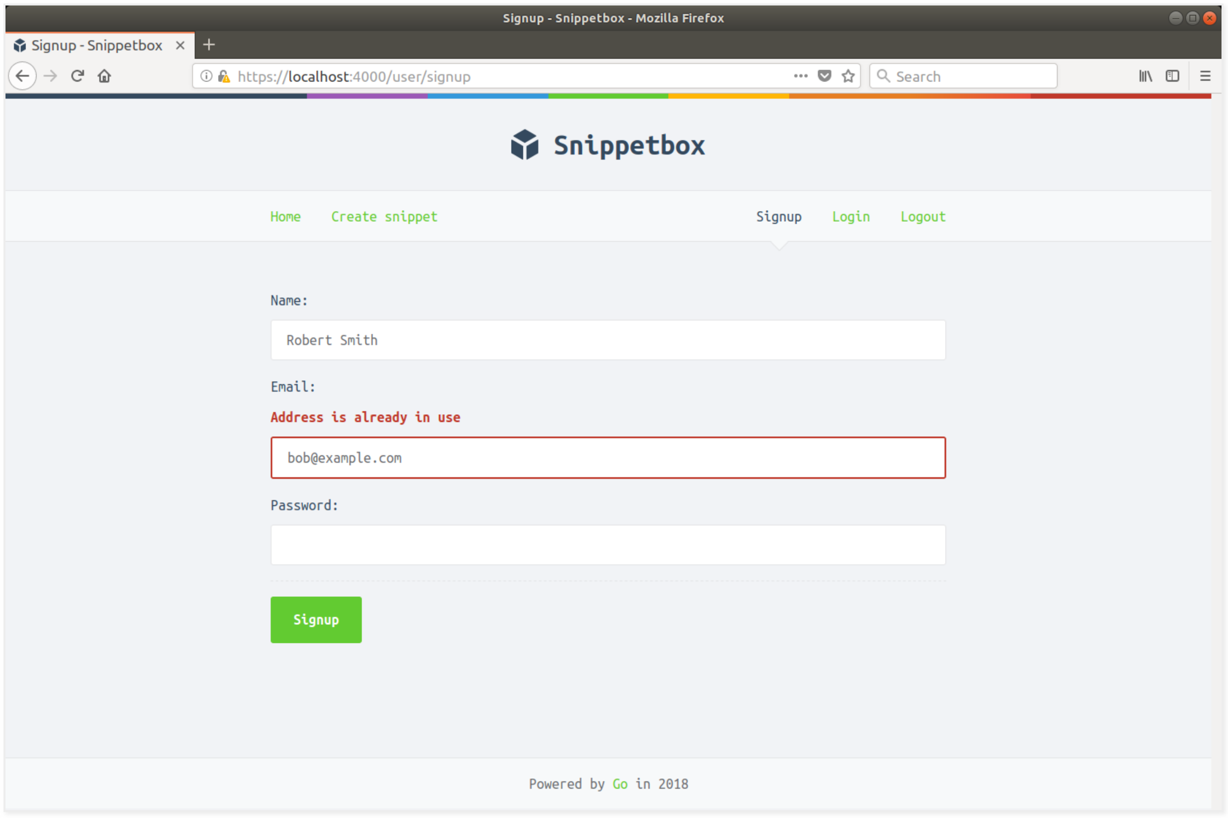


```
Display the user login form...
```

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
+----+-----------+-----------------+----------------------------------------
|  1 | Bob Jones | bob@example.com | $2a$12$mNXQrOwVWp/TqAzCCyDoyegtpV40EXwrzVL
+----+-----------+-----------------+----------------------------------------
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:



Name:

Robert Smith

Email:

Address is already in use

bob@example.com

Password:

Powered by Go in 2018

Additional Information

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 a `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 my view is that 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.

Chapter 11.4.

User Login

The process for creating the user login page follows the same general pattern as the user signup. First, let’s create a `ui/html/login.page.tmpl` template containing the markup below:

```
cd $HOME/code/snippetbox
$ touch ui/html/login.page.tmpl
```

File: ui/html/login.page.tmpl

```
{{template "base" .}}

{{define "title"}}Login{{end}}

{{define "body"}}
<form action='/user/login' method='POST' novalidate>
    {{with .Form}}
        {{with .Errors.Get "generic"}}
            <div class='error'>{{.}}</div>
        {{end}}
        <div>
            <label>Email:</label>
            <input type='email' name='email' value='{{.Get "email"}}'>
        </div>
        <div>
            <label>Password:</label>
            <input type='password' name='password'>
        </div>
        <div>
            <input type='submit' value='Login'>
        </div>
    {{end}}
</form>
{{end}}
```

Notice how we've included a `{{with .Errors.Get "generic"}}` action at the top of the form, instead of displaying of error messages for the individual fields? We'll use this to present the user with a generic "your email address or password is wrong" message if their login fails, instead of explicitly indicating which of the fields is wrong.

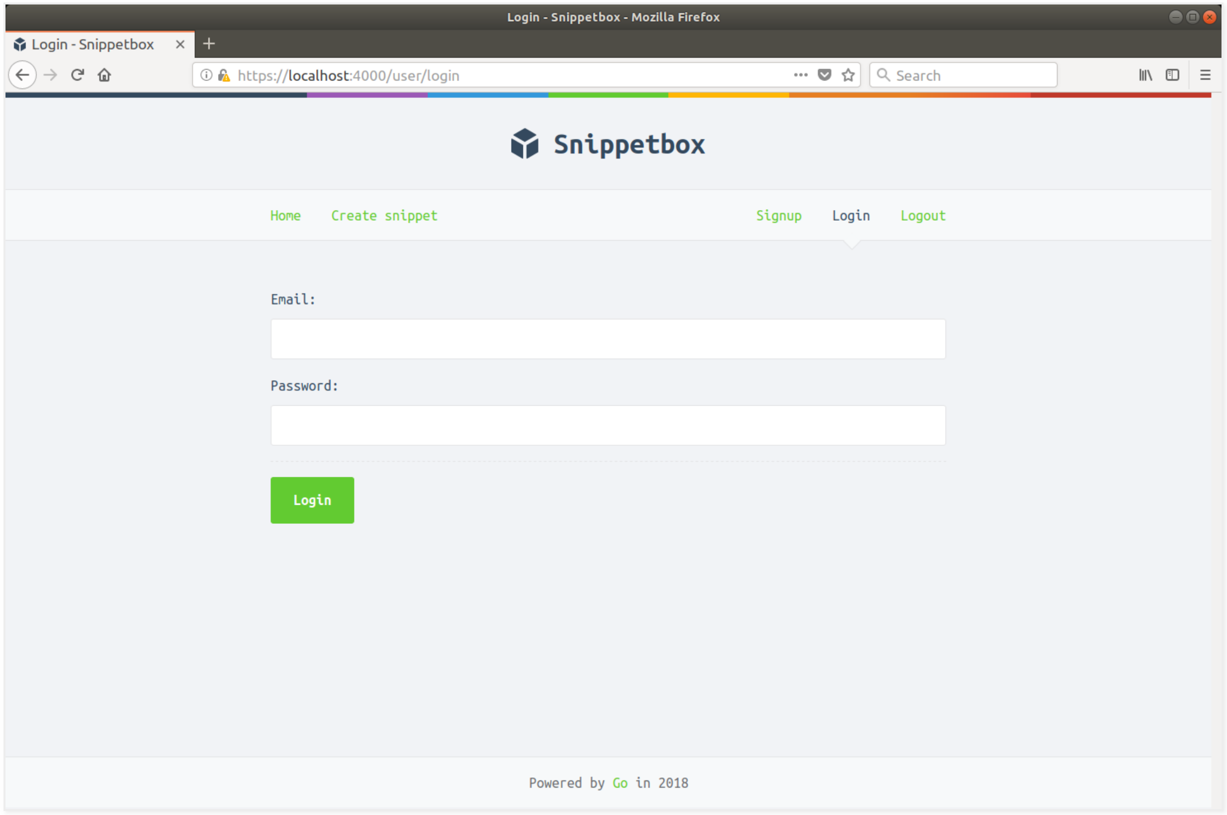
Again, for security, we're not re-displaying the user's password if the login fails.

We can then hook this up so it's rendered by our `loginUserForm` handler like so:

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

```go
package main

...

func (app *application) loginUserForm(w http.ResponseWriter, r *http.Request) {
    app.render(w, r, "login.page.tmpl", &templateData{
        Form: forms.New(nil),
    })
}

...
```

If you run the application and visit https://localhost:4000/user/login you should 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 want to return the `ErrInvalidCredentials` error that we made earlier.

2. If the email does exist, we want to compare the bcrypt-hashed password to 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 `pkg/models/mysql/users.go` file:

```
File: pkg/models/mysql/users.go
```

```go
package mysql

...

type UserModel struct {
    DB *sql.DB
}

...

func (m *UserModel) Authenticate(email, password string) (int, error) {
    // Retrieve the id and hashed password associated with the given email. If
    // matching email exists, we return the ErrInvalidCredentials error.
    var id int
    var hashedPassword []byte
    row := m.DB.QueryRow("SELECT id, hashed_password FROM users WHERE email = ?
    err := row.Scan(&id, &hashedPassword)
    if err == sql.ErrNoRows {
        return 0, models.ErrInvalidCredentials
    } else if err != nil {
        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 == bcrypt.ErrMismatchedHashAndPassword {
        return 0, models.ErrInvalidCredentials
    } else if err != nil {
        return 0, err
    }

    // Otherwise, the password is correct. Return the user ID.
    return id, nil
}
```

Our next step involves updating the `loginUser` handler so that it parses the submitted login form data and calls this `UserModel.Authenticate()` method.

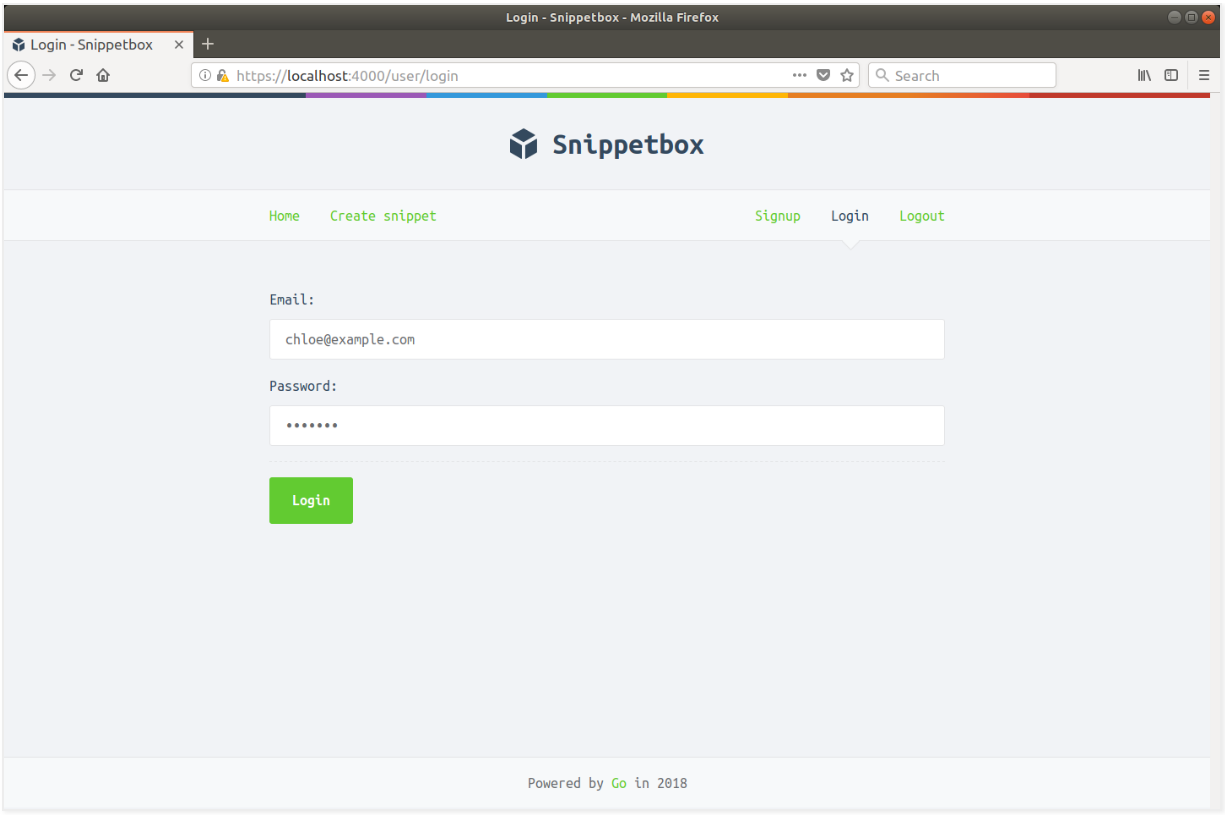
If the login details are valid, we then want to add the user's `id` to their session data so that — for future requests — we know that they have authenticated successfully and which user they are.

Head over to your `handlers.go` file and update it as follows:

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

```
package main

...

func (app *application) loginUser(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.clientError(w, http.StatusBadRequest)
        return
    }

    // Check whether the credentials are valid. If they're not, add a generic e
    // message to the form failures map and re-display the login page.
    form := forms.New(r.PostForm)
    id, err := app.users.Authenticate(form.Get("email"), form.Get("password"))
    if err == models.ErrInvalidCredentials {
        form.Errors.Add("generic", "Email or Password is incorrect")
        app.render(w, r, "login.page.tmpl", &templateData{Form: form})
        return
    } else if err != nil {
        app.serverError(w, err)
        return
    }

    // Add the ID of the current user to the session, so that they are now 'log
    // in'.
    app.session.Put(r, "userID", id)

    // Redirect the user to the create snippet page.
    http.Redirect(w, r, "/snippet/create", http.StatusSeeOther)
}

...
```

So, let's give this a try. Restart the application and try submitting some invalid user credentials…


Login - Snippetbox - Mozilla Firefox
Login - Snippetbox
https://localhost:4000/user/login
Search
Snippetbox
Home
Create snippet
Signup
Login
Logout

Email:

chloe@example.com

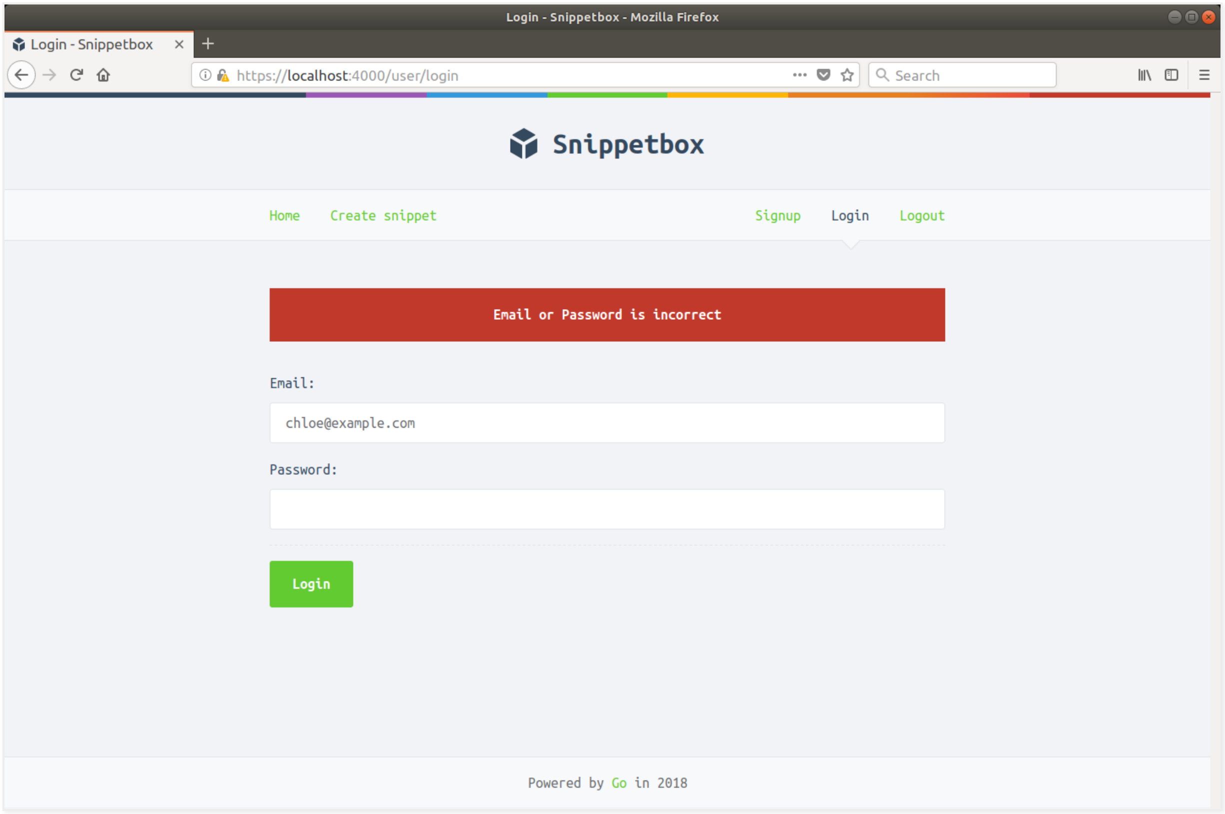
Password:

•••••••

Login

Powered by Go in 2018

You should get a validation failure which looks like this:


Login - Snippetbox - Mozilla Firefox

https://localhost:4000/user/login

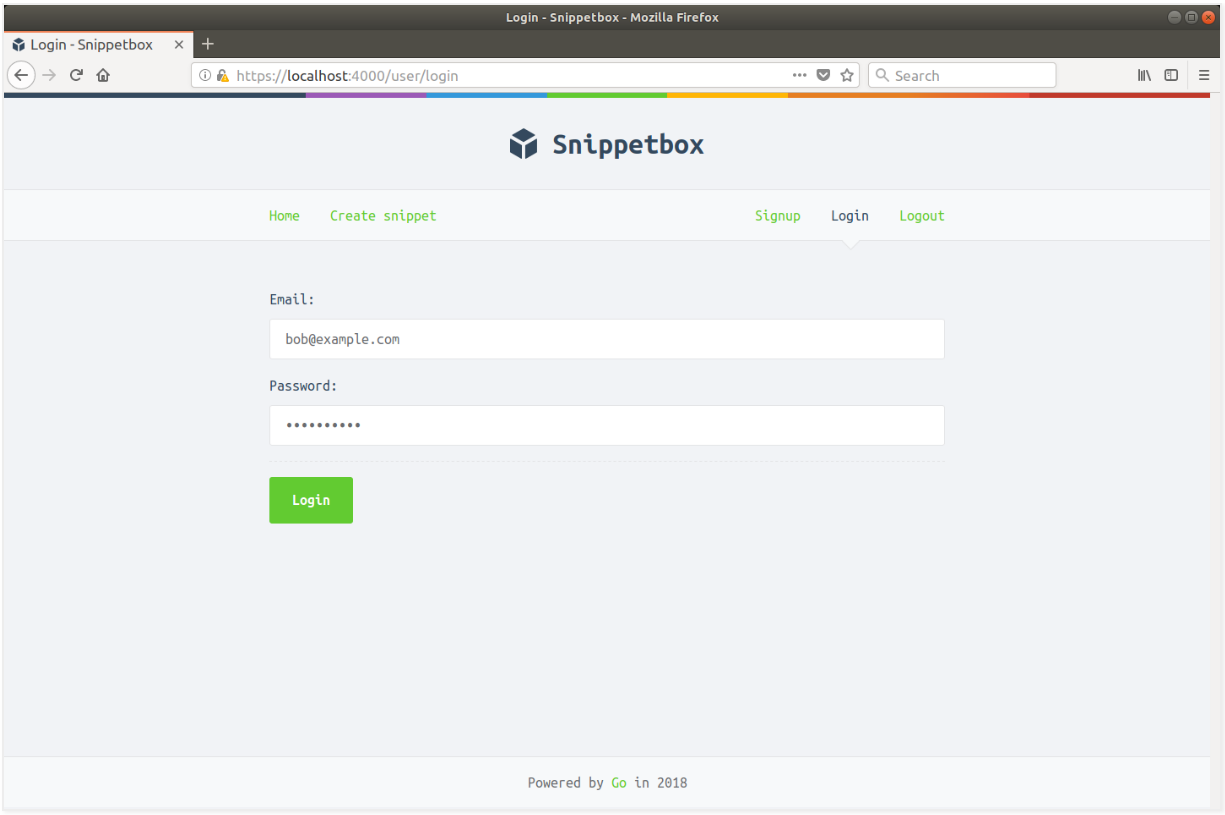
Email or Password is incorrect

Email: chloe@example.com

Password:

Powered by Go in 2018

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:



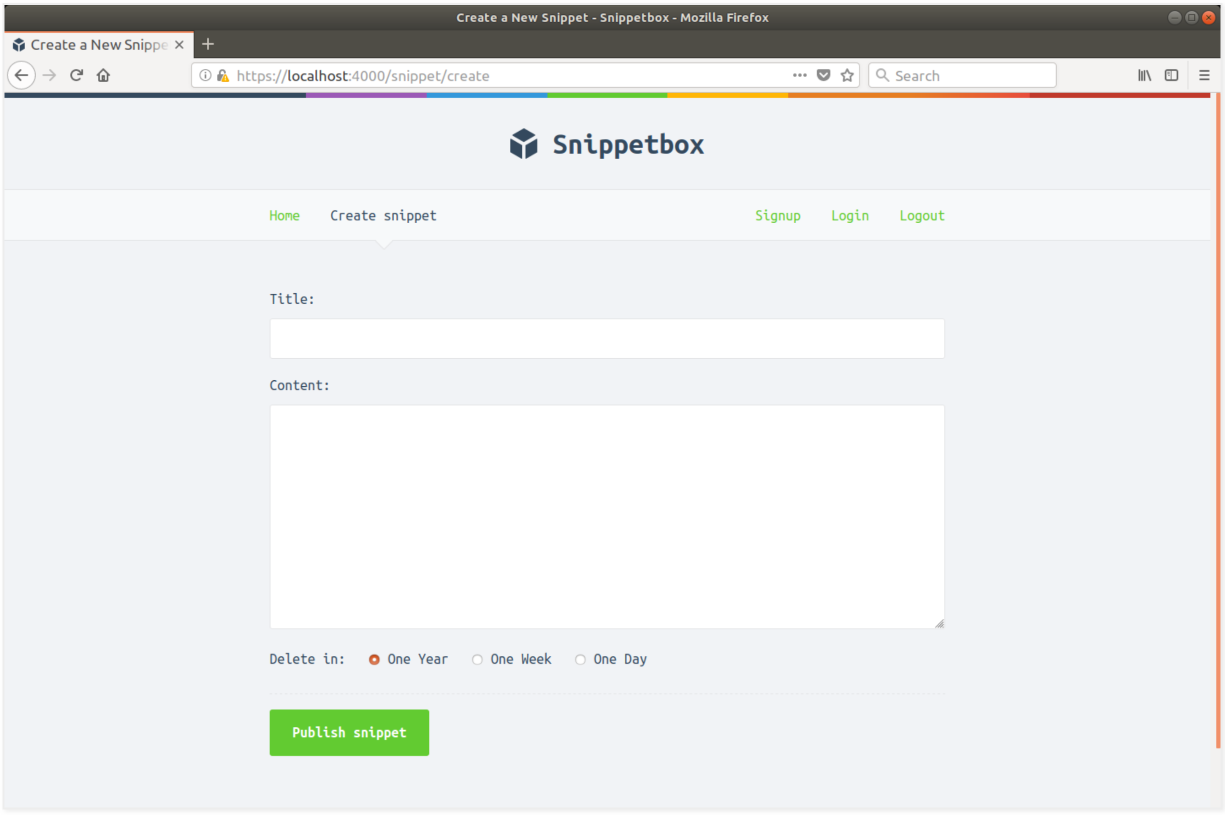
Email:

bob@example.com

Password:

••••••••••

Powered by Go in 2018


Create a New Snippet - Snippetbox - Mozilla Firefox
Create a New Snippe
https://localhost:4000/snippet/create
Search
Snippetbox
Home
Create snippet
Signup
Login
Logout
Publish snippet

Title:

Content:

Delete in: [✓ One Year] One Week One Day

Additional Information

Session Fixation Attacks

Because we are using an encrypted cookie to store the session data (and because the encrypted cookie value changes unpredictably every time the underlying session data changes) we don't need to worry about session fixation attacks .

However, if you were using a server-side data store for sessions (with a session ID stored in a cookie) then to mitigate the risk of session fixation attacks it's important that you change the value of the session ID before making any changes to privilege levels (e.g. login and logout operations).

If you're using the SCS package to manage server-side sessions you can use the `Session.RenewToken()` method to do this.

If you're using Gorilla Sessions to manage server-side sessions there's unfortunately no way to refresh the session ID while retaining the session data, so you'll need to either accept the risk of a session fixation attack or create a new session and copy the data across manually.

Chapter 11.5.

User Logout

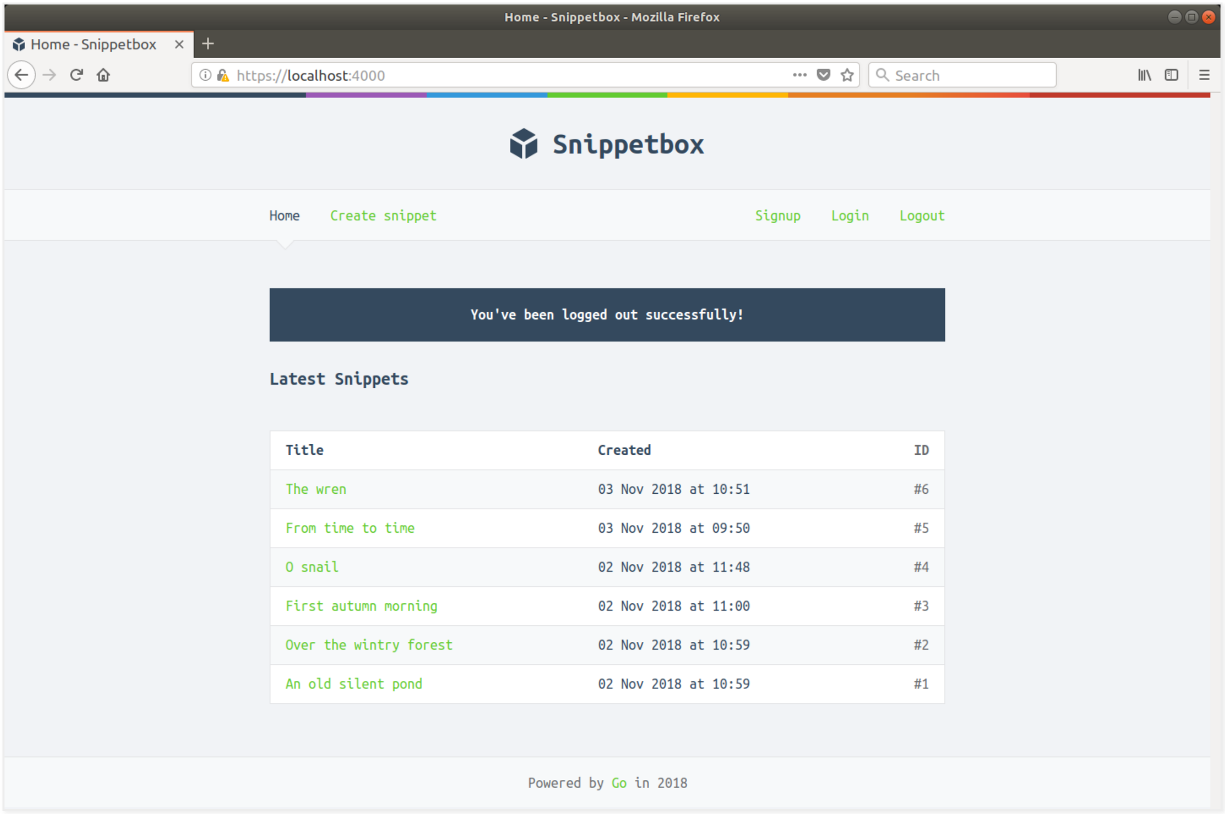
This brings us nicely to logging out a user. Implementing the user logout is straightforward in comparison to the signup and login — all we need to do is remove the `"userID"` value from the session.

Let's update the `logoutUser` hander to do exactly that.

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

```
package main

...

func (app *application) logoutUser(w http.ResponseWriter, r *http.Request) {
    // Remove the userID from the session data so that the user is 'logged out'
    app.session.Remove(r, "userID")
    // Add a flash message to the session to confirm to the user that they've b
    app.session.Put(r, "flash", "You've been logged out successfully!")
    http.Redirect(w, r, "/", 303)
}
```

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:


Home - Snippetbox - Mozilla Firefox

https://localhost:4000

Snippetbox

Home Create snippet Signup Login Logout

You've been logged out successfully!

Latest Snippets

Title	Created	ID
The wren	03 Nov 2018 at 10:51	#6
From time to time	03 Nov 2018 at 09:50	#5
O snail	02 Nov 2018 at 11:48	#4
First autumn morning	02 Nov 2018 at 11:00	#3
Over the wintry forest	02 Nov 2018 at 10:59	#2
An old silent pond	02 Nov 2018 at 10:59	#1

Powered by Go in 2018

Chapter 11.6.

User Authorization

Having the ability to log users in and out 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. 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 a `"userID"` value in their session data.

So let's first open the `cmd/web/helpers.go` file and add an `authenticatedUser()` helper function to retrieve and return the `"userID"` value from the session:

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

```go
package main

...

// The authenticatedUser method returns the ID of the current user from the
// session, or zero if the request is from an unauthenticated user.
func (app *application) authenticatedUser(r *http.Request) int {
    return app.session.GetInt(r, "userID")
}
```

Important: In this code we're using the `GetInt()` method to retrieve the `int` value associated with the `"userID"` key in the current session. If there is no `"userID"` value in the session, or the value isn't an integer *this will return zero*.

So, in essence, we can confirm that a request is coming from a logged in user by simply checking that the return value from `authenticatedUser()` is not zero.

The next step for us is to find a way to pass the `"userID"` value from `authenticatedUser()` to our HTML templates, so that we can toggle the contents of the navigation bar based on whether it is zero or not.

There's two parts to this. First, we'll need to add a new `AuthenticatedUser` field to our `templateData` struct:

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

```
package main

import (
    "html/template"
    "path/filepath"
    "time"

    "alexedwards.net/snippetbox/pkg/forms"
    "alexedwards.net/snippetbox/pkg/models"
)
```

```
)

// Add a new AuthenticatedUser field to the templateData struct.
type templateData struct {
    AuthenticatedUser int
    CurrentYear       int
    Flash             string
    Form              *forms.Form
    Snippet           *models.Snippet
    Snippets          []*models.Snippet
}

...
```

And the second step is to update our `addDefaultData()` helper method so that the user ID 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) addDefaultData(td *templateData, r *http.Request) *temp
    if td == nil {
        td = &templateData{}
    }

    td.AuthenticatedUser = app.authenticatedUser(r)
    td.CurrentYear = time.Now().Year()
    td.Flash = app.session.PopString(r, "flash")
    return td
}
```

```
...
```

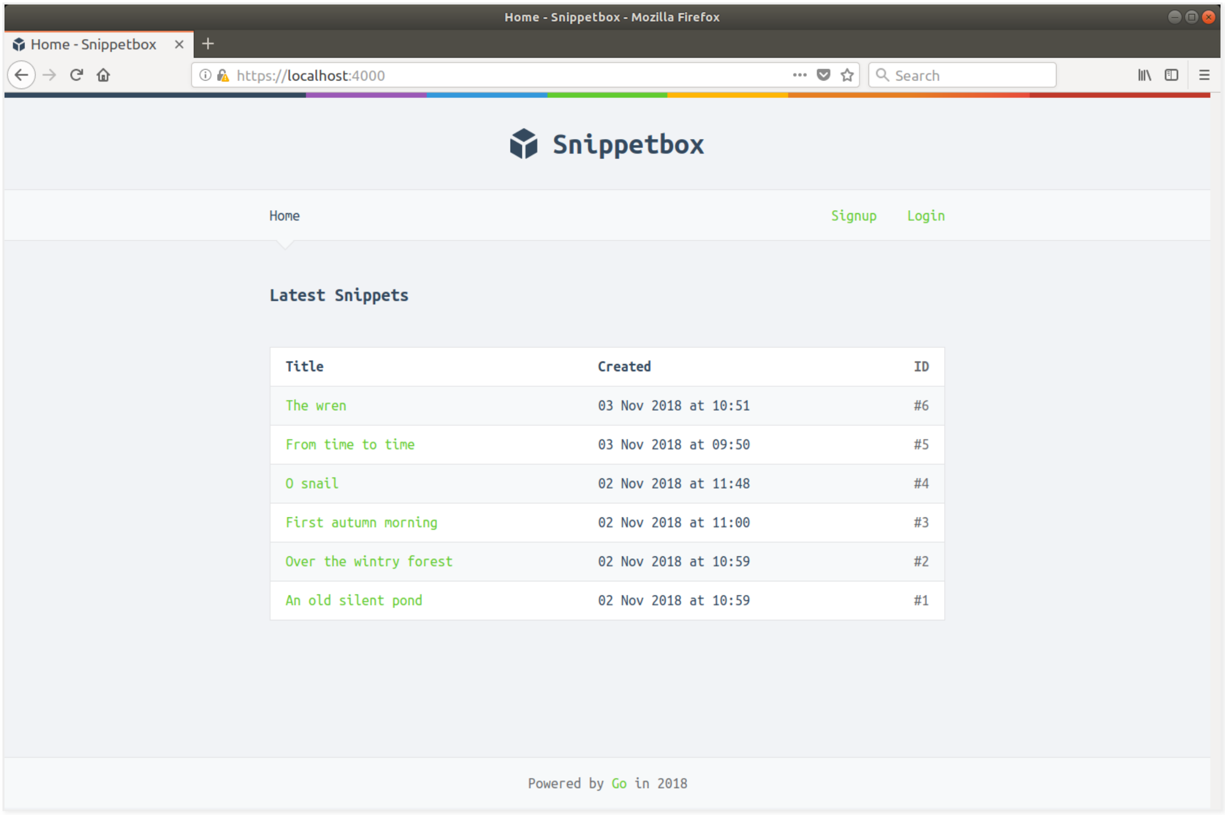
And now we can update the `ui/html/base.layout.tmpl` file to toggle the navigation links using the `{{if .AuthenticatedUser}}` action like so:

```
File: ui/html/base.layout.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-
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ub
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <div>
                <a href='/'>Home</a>
                {{if .AuthenticatedUser}}
                    <a href='/snippet/create'>Create snippet</a>
                {{end}}
            </div>
            <div>
                {{if .AuthenticatedUser}}
                    <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>
        <section>
            {{with .Flash}}
            <div class='flash '>{{.}}</div>
            {{end}}
            {{template "body" .}}
        </section>
        {{template "footer" .}}
        <script src="/static/js/main.js" type="text/javascript"></script>
    </body>
</html>
{{end}}
```

Remember: The `{{if ...}}` action considers empty values (false, 0, any nil pointer or interface value, and any array, slice, map, or string of length zero) to be false.

Save all the files and try running the application now. If you're not currently logged in, your application homepage should look like this:


Home - Snippetbox - Mozilla Firefox

https://localhost:4000

Snippetbox

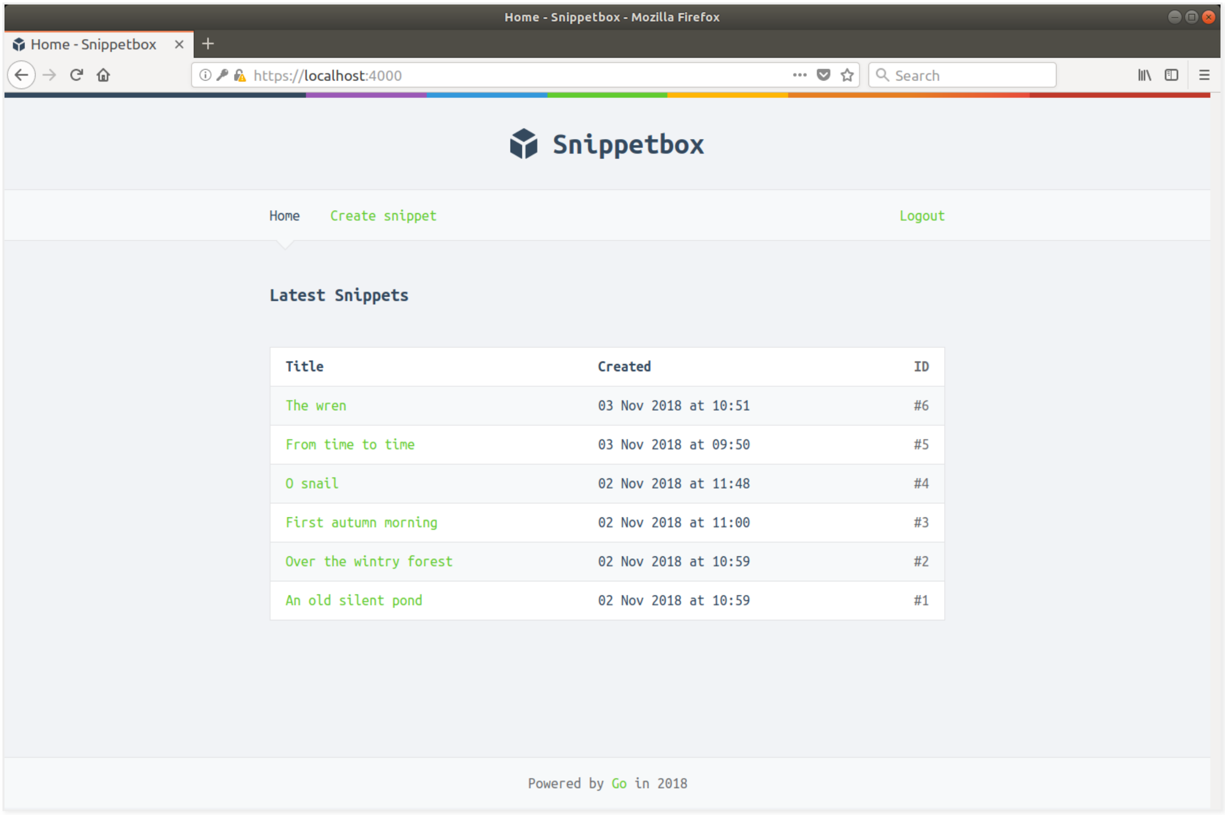
Home Signup Login

Latest Snippets

Title	Created	ID
The wren	03 Nov 2018 at 10:51	#6
From time to time	03 Nov 2018 at 09:50	#5
O snail	02 Nov 2018 at 11:48	#4
First autumn morning	02 Nov 2018 at 11:00	#3
Over the wintry forest	02 Nov 2018 at 10:59	#2
An old silent pond	02 Nov 2018 at 10:59	#1

Powered by Go in 2018

Otherwise — if you are logged in — your homepage should look like this:



Latest Snippets

Title	Created	ID
The wren	03 Nov 2018 at 10:51	#6
From time to time	03 Nov 2018 at 09:50	#5
O snail	02 Nov 2018 at 11:48	#4
First autumn morning	02 Nov 2018 at 11:00	#3
Over the wintry forest	02 Nov 2018 at 10:59	#2
An old silent pond	02 Nov 2018 at 10:59	#1

Powered by Go in 2018

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 `requireAuthenticatedUser()` middleware function, following the same pattern that we used earlier in the book:

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

```go
package main

...

func (app *application) requireAuthenticatedUser(next http.Handler) http.Handle
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // If the user is not authenticated, redirect them to the login page an
        // return from the middleware chain so that no subsequent handlers in
        // the chain are executed.
        if app.authenticatedUser(r) == 0 {
            http.Redirect(w, r, "/user/login", 302)
            return
        }

        // Otherwise 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.

If you're using the justinas/alice package to manage your middleware chains, you can add the new `requireAuthenticatedUser()` middleware to the `dynamicMiddleware` chain on a per-route basis by using the `Append()` method like so:

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

```go
package main

import (
    "net/http"

    "github.com/bmizerany/pat"
    "github.com/justinas/alice"
)

func (app *application) routes() http.Handler {
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea
    dynamicMiddleware := alice.New(app.session.Enable)

    mux := pat.New()
    mux.Get("/", dynamicMiddleware.ThenFunc(app.home))
    // Add the requireAuthenticatedUser middleware to the chain.
    mux.Get("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticate
    // Add the requireAuthenticatedUser middleware to the chain.
    mux.Post("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticat
    mux.Get("/snippet/:id", dynamicMiddleware.ThenFunc(app.showSnippet))

    mux.Get("/user/signup", dynamicMiddleware.ThenFunc(app.signupUserForm))
    mux.Post("/user/signup", dynamicMiddleware.ThenFunc(app.signupUser))
    mux.Get("/user/login", dynamicMiddleware.ThenFunc(app.loginUserForm))
    mux.Post("/user/login", dynamicMiddleware.ThenFunc(app.loginUser))
    // Add the requireAuthenticatedUser middleware to the chain.
    mux.Post("/user/logout", dynamicMiddleware.Append(app.requireAuthenticatedU

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

    return standardMiddleware.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 -X POST https://localhost:4000/snippet/create
HTTP/2 302
location: /user/login
x-frame-options: deny
x-xss-protection: 1; mode=block
content-length: 0
date: Sun, 21 Oct 2018 11:00:13 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:

```
mux.Get("/snippet/create", app.session.Enable(app.requireAuthenticatedUser(http
```

Chapter 11.7.

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 form of cross-domain 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 a malicious website which contains some code that sends a request to `POST /snippets/create` to add a new snippet to our database.
- Since the user is still logged in to our application, the request is processed with their privileges. Completely unknown to them, a new snippet will be added to our database.

As well as 'traditional' CSRF attacks like the above (which focus on processing a request 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 set on our session cookie.

By default the golangcollege/sessions package that we're using always sets `SameSite=Lax` on the session cookie, which means that session cookie *won't* be sent by the user's browser for cross-site usage, apart from when the usage is deemed to be a safe request which doesn't change the state of the target application.

If you want to change this setting to `SameSite=Strict` instead you can, by configuring the sessions like this in your `main.go` file:

```
...
session := sessions.New([]byte(*secret))
session.Lifetime = 12 * time.Hour
session.Secure = true
session.SameSite = http.SameSiteStrictMode
...
```

Note: Using `SameSite=Strict` will block the session cookie being sent by the user's browser for *all* cross-site usage. This includes when a user clicks on an external link to your application, meaning that after clicking the link they will initially be treated as 'not logged in' even if they have an active session containing their `"userID"` value.

Unfortunately, at the time of writing, the SameSite attribute is only supported by 71% of browsers worldwide. So, although it's something than 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_Prevention_Cheat_Sheet). Like session management and password hashing, when it comes to this there's a lot that you can get wrong so it's probably safer 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_Prevention_Cheat_Sheet#Double_Submit_Cookie) 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 form that's 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
```

```
go: finding github.com/justinas/nosurf latest
go: downloading github.com/justinas/nosurf v0.0.0-20171023064657-7182011986c4
```

Using the nosurf Package

Open up your `cmd/web/middleware.go` file and create a new `noSurf()` function like so:

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

```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 flags set.
func noSurf(next http.Handler) http.Handler {
    csrfHandler := nosurf.New(next)
    csrfHandler.SetBaseCookie(http.Cookie{
        HttpOnly: true,
        Path:     "/",
        Secure:   true,
    })

    return csrfHandler
}
```

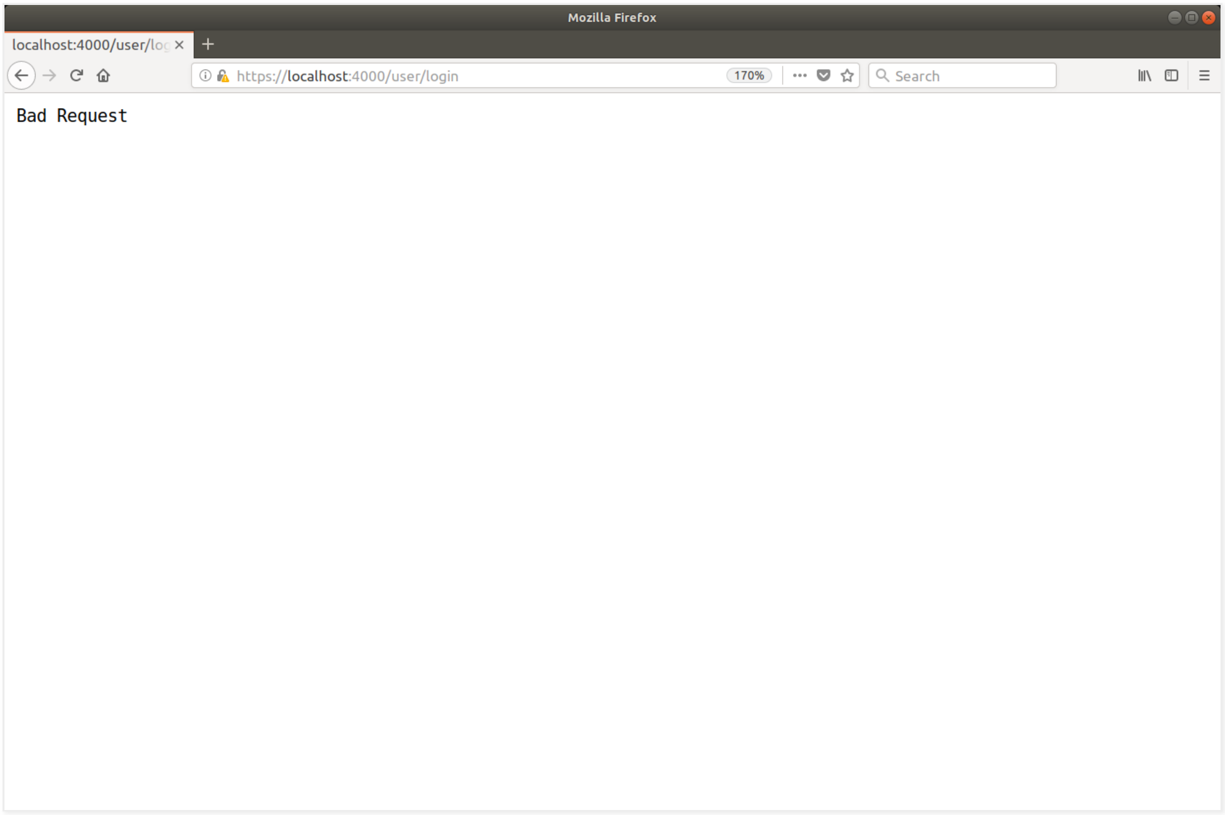
One of the forms that we need to protect from CSRF attacks is our logout form, which is included in our `base.layout.tmpl` file and could potentially appear on any page of our application. So, because of this, we need to use our `noSurf()` middleware on all our application routes (apart from `/static/`).

So, let's update the `cmd/web/routes.go` file to add this `noSurf()` middleware to the `dynamicMiddleware` chain that we made earlier:

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

```go
package main

...

func (app *application) routes() http.Handler {
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea
    // Use the nosurf middleware on all our 'dynamic' routes.
    dynamicMiddleware := alice.New(app.session.Enable, noSurf)

    mux := pat.New()
    mux.Get("/", dynamicMiddleware.ThenFunc(app.home))
    mux.Get("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticate
    mux.Post("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticat
    mux.Get("/snippet/:id", dynamicMiddleware.ThenFunc(app.showSnippet))

    mux.Get("/user/signup", dynamicMiddleware.ThenFunc(app.signupUserForm))
    mux.Post("/user/signup", dynamicMiddleware.ThenFunc(app.signupUser))
    mux.Get("/user/login", dynamicMiddleware.ThenFunc(app.loginUserForm))
    mux.Post("/user/login", dynamicMiddleware.ThenFunc(app.loginUser))
    mux.Post("/user/logout", dynamicMiddleware.Append(app.requireAuthenticatedU

    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Get("/static/", http.StripPrefix("/static", fileServer))
    return standardMiddleware.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.



```
Bad Request
```

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
```

```go
package main

import (
    "html/template"
    "path/filepath"
    "time"

    "alexedwards.net/snippetbox/pkg/forms"
    "alexedwards.net/snippetbox/pkg/models"
)

// Add a new CSRFToken field to the templateData.
type templateData struct {
    AuthenticatedUser int
    CSRFToken         string
    CurrentYear       int
    Flash             string
    Form              *forms.Form
    Snippet           *models.Snippet
    Snippets          []*models.Snippet
}

...
```

And because the logout form can potentially appear on every page, it makes sense to add the CSRF token to the template data via our `addDefaultData()` helper. This will mean it's automatically available to our templates each time we render a page. Update the `cmd/web/helpers.go` file as follows:

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

```
package main

import (
    "bytes"
    "fmt"
    "net/http"
    "runtime/debug"
    "time"

    "github.com/justinas/nosurf" // New import
)

func (app *application) addDefaultData(td *templateData, r *http.Request) *temp
    if td == nil {
        td = &templateData{}
    }

    // Add the CSRF token to the templateData struct.
    td.CSRFToken = nosurf.Token(r)
    td.CurrentYear = time.Now().Year()
    td.Flash = app.session.PopString(r, "flash")
    td.AuthenticatedUser = app.authenticatedUser(r)
    return td
}

...
```

Finally, let’s update all the forms in our application to use this token.

```
File: ui/html/base.layout.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-
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ub
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <div>
                <a href='/'>Home</a>
                {{if .AuthenticatedUser}}
                    <a href='/snippet/create'>Create snippet</a>
                {{end}}
            </div>
            <div>
                {{if .AuthenticatedUser}}
                    <form action='/user/logout' method='POST'>
                        <!-- Include the CSRF token -->
                        <input type='hidden' name='csrf_token' value='{{.CSRFTo
                        <button>Logout</button>
                    </form>
                {{else}}
                    <a href='/user/signup'>Signup</a>
                    <a href='/user/login'>Login</a>
                {{end}}
            </div>
        </nav>
        <section>
            {{with .Flash}}
            <div class='flash '>{{.}}</div>
            {{end}}
            {{template "body" .}}
        </section>
        {{template "footer" .}}
        <script src="/static/js/main.js" type="text/javascript"></script>
    </body>
```

```
</html>
{{end}}
```

```
File: ui/html/create.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Create a New Snippet{{end}}

{{define "body"}}
<form action='/snippet/create' method='POST'>
    <!-- Include the CSRF token -->
    <input type='hidden' name='csrf_token' value='{{.CSRFToken}}'>
    {{with .Form}}
        <div>
            <label>Title:</label>
            {{with .Errors.Get "title"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='text' name='title' value='{{.Get "title"}}'>
        </div>
        <div>
            <label>Content:</label>
            {{with .Errors.Get "content"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <textarea name='content'>{{.Get "content"}}</textarea>
        </div>
        <div>
            <label>Delete in:</label>
            {{with .Errors.Get "expires"}}
                <label class='error'>{{.}}</label>
            {{end}}
            {{$exp := or (.Get "expires") "365"}}
            <input type='radio' name='expires' value='365' {{if (eq $exp "365")
            <input type='radio' name='expires' value='7' {{if (eq $exp "7")}}ch
            <input type='radio' name='expires' value='1' {{if (eq $exp "1")}}ch
        </div>
        <div>
            <input type='submit' value='Publish snippet'>
        </div>
    {{end}}
</form>
{{end}}
```

File: ui/html/login.page.tmpl

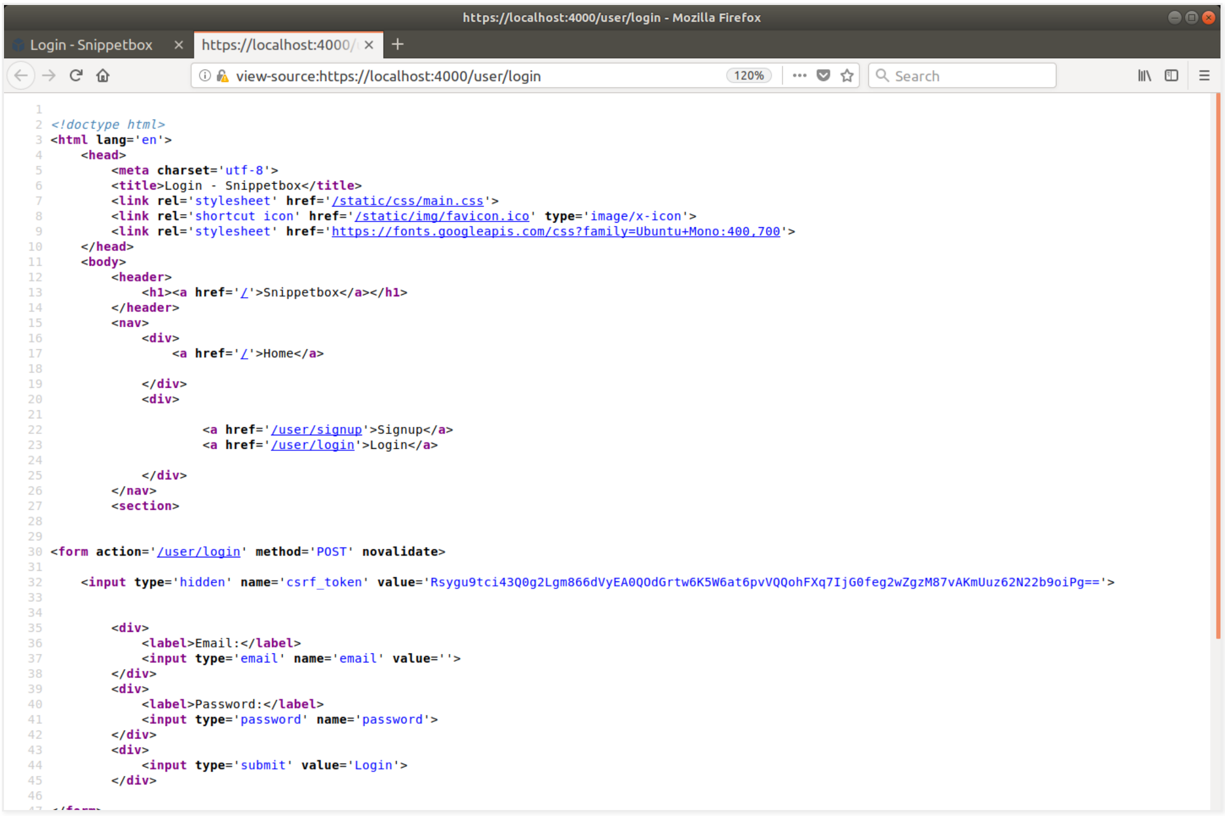
```
{{template "base" .}}

{{define "title"}}Login{{end}}

{{define "body"}}
<form action='/user/login' method='POST' novalidate>
    <!-- Include the CSRF token -->
    <input type='hidden' name='csrf_token' value='{{.CSRFToken}}'>
    {{with .Form}}
        {{with .Errors.Get "generic"}}
            <div class='error'>{{.}}</div>
        {{end}}
        <div>
            <label>Email:</label>
            <input type='email' name='email' value='{{.Get "email"}}'>
        </div>
        <div>

            <label>Password:</label>
            <input type='password' name='password'>
        </div>
        <div>
            <input type='submit' value='Login'>
        </div>
    {{end}}
</form>
```

```
{{end}}
```

```
File: ui/html/signup.page.tmpl
```

```
{{template "base" .}}

{{define "title"}}Signup{{end}}

{{define "body"}}
<form action='/user/signup' method='POST' novalidate>
    <!-- Include the CSRF token -->
    <input type='hidden' name='csrf_token' value='{{.CSRFToken}}'>
    {{with .Form}}
        <div>
            <label>Name:</label>
            {{with .Errors.Get "name"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='text' name='name' value='{{.Get "name"}}'>
        </div>
        <div>
            <label>Email:</label>
            {{with .Errors.Get "email"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='email' name='email' value='{{.Get "email"}}'>
        </div>
        <div>
            <label>Password:</label>
            {{with .Errors.Get "password"}}
                <label class='error'>{{.}}</label>
            {{end}}
            <input type='password' name='password'>
        </div>
        <div>
            <input type='submit' value='Signup'>
        </div>
    {{end}}
</form>
{{end}}
```

Try firing up the application and *view source* of one of the forms. You should see that it now has a CSRF token included in a hidden field, like so.


https://localhost:4000/user/login - Mozilla Firefox
Login - Snippetbox
https://localhost:4000/
view-source:https://localhost:4000/user/login
120%
Search

```

<!doctype html>
<html lang='en'>
    <head>
        <meta charset='utf-8'>
        <title>Login - 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>
            <div>
                <a href='/'>Home</a>

            </div>
            <div>

                    <a href='/user/signup'>Signup</a>
                    <a href='/user/login'>Login</a>

            </div>
        </nav>
        <section>


<form action='/user/login' method='POST' novalidate>

    <input type='hidden' name='csrf_token' value='Rsygu9tci43Q0g2Lgm866dVyEA0QOdGrtw6K5W6at6pvVQQohFXq7IjG0feg2wZgzM87vAKmUuz62N22b9oiPg=='>


        <div>
            <label>Email:</label>
            <input type='email' name='email' value=''>
        </div>
        <div>
            <label>Password:</label>
            <input type='password' name='password'>
        </div>
        <div>
            <input type='submit' value='Login'>
        </div>

```

And if you try submitting the forms it should now work correctly again.

Chapter 12.

Using Request Context

At the moment our logic for authenticating a user consists of simply checking whether a `"userID"` value exists in their session data, like so:

```
func (app *application) authenticatedUser(r *http.Request) int {
    return app.session.GetInt(r, "userID")
}
```

We could make this authentication more robust by also checking that this `"userID"` value relates to a real, active record in our `users` table — and ensure that we haven't decided to delete the user (for some reason or another) since they last logged in.

But there is a slight problem with doing this additional check.

Our `authenticatedUser()` helper is called sometimes multiple times in each request cycle. Currently we use it twice — once in the `requireAuthenticatedUser()` middleware and again in the `addDefaultData()` helper. So, if we check the database from the `authenticatedUser()` helper directly, we would end up making duplicated round-trips to the database during every request. 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-and-valid* user or not, and then pass this 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.

Chapter 12.1.

How Request Context Works

In essence every `http.Request` that our 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 *authenticate the user* once in some middleware, and then make the information about the user 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 use it in our application.

The Request Context Syntax

The basic code for adding information to a request's context looks like this:

```
ctx := r.Context()
ctx = context.WithValue(ctx, "user", &models.User{Name: "Bob Jones"})
r = r.WithContext(ctx)
```

```
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 `context.WithValue()` method to create a *new copy* of the existing context, with a `*models.User` struct added to it. In this case, we’ve used the string `"user"` as the *key* for this data in the context.
- Then finally we use the `r.WithContext()` method to create a *copy* of the request containing our new context.

It’s important to be clear that we don’t actually update the context for a request directly. What we’re do is *create 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 a bit more verbose than it needs to be. It’s typical to shorten it a little bit like so:

```
ctx = context.WithValue(r.Context(), "user", &models.User{Name: "Bob Jones"})
r = r.WithContext(ctx)
```

So, that’s how you add data to request context. But what about retrieving it again?

The important thing to explain here is that, behind the scenes, request context values are stored with the type `interface{}`. 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 use the `r.Context().Value()` method, like so:

```
user, ok := r.Context().Value("user").(*models.User)
if !ok {
    return fmt.Errorf("could not convert %T to *models.User", user)
}

fmt.Println(user.Name)
```

Avoiding Key Collisions

In the code samples above, I've used the string `"user"` as the key for storing and retrieving the data from a request 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 `"user"`. And that would cause a naming collision and bugginess.

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 so far, it's much better to do something like this:

```
type contextKey string

var contextKeyUser = contextKey("user")

...

ctx := r.Context()
ctx = context.WithValue(ctx, contextKeyUser, &models.User{Name: "Bob Jones"})
r = r.WithContext(ctx)

...

user, ok := r.Context().Value(contextKeyUser).(*models.User)
if !ok {
    return fmt.Errorf("could not convert %T to *models.User", user)
}



fmt.Println(user.Name)
```

Chapter 12.2.

Request Context for Authentication/Authorization

So, with those explanations out of the way let's use the request context functionality in our application.

We'll begin by heading back to our `pkg/models/mysql/users.go` file and updating the `UserModel.Get()` method, so that it retrieves the details of a specific user from the database like so:

```
File: pkg/models/mysql/users.go
```

```go
package mysql

import (
    "database/sql"
    "strings"

    "alexedwards.net/snippetbox/pkg/models"

    "github.com/go-sql-driver/mysql"
    "golang.org/x/crypto/bcrypt"
)

type UserModel struct {
    DB *sql.DB
}
```

```
...

func (m *UserModel) Get(id int) (*models.User, error) {
    s := &models.User{}

    stmt := `SELECT id, name, email, created FROM users WHERE id = ?`
    err := m.DB.QueryRow(stmt, id).Scan(&s.ID, &s.Name, &s.Email, &s.Created)
    if err == sql.ErrNoRows {
        return nil, models.ErrNoRecord
    } else if err != nil {
        return nil, err
    }

    return s, nil
}
```

Then open the `cmd/web/main.go` file and define your own custom `contextKey` type and `contextKeyUser` variable, so that we have a unique key we can use to store and retrieve the user details from request context.

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

```
package main

import (
    "crypto/tls"
    "database/sql"
    "flag"
    "html/template"
    "log"
    "net/http"
    "os"
    "time"
    "alexedwards.net/snippetbox/pkg/models/mysql"

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

type contextKey string

var contextKeyUser = contextKey("user")

type application struct {
    errorLog      *log.Logger
    infoLog       *log.Logger
    session       *sessions.Session
    snippets      *mysql.SnippetModel
    templateCache map[string]*template.Template
    users         *mysql.UserModel
}

...
```

And now for the exciting part. Let's create a new `authenticate()` middleware function which fetches the details for the current user from the database (based on the `"userID"` in their session), and then adds their details to the request context.

Here's the code:

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

```
package main

import (
    "context" // New import
    "fmt"
    "net/http"

    "alexedwards.net/snippetbox/pkg/models" // New import

    "github.com/justinas/nosurf"
)

...

func (app *application) authenticate(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Check if a userID value exists in the session. If this *isn't
        // present* then call the next handler in the chain as normal.
        exists := app.session.Exists(r, "userID")
        if !exists {
            next.ServeHTTP(w, r)
            return
        }

        // Fetch the details of the current user from the database. If
        // no matching record is found, remove the (invalid) userID from
        // their session and call the next handler in the chain as normal.
        user, err := app.users.Get(app.session.GetInt(r, "userID"))
        if err == models.ErrNoRecord {
            app.session.Remove(r, "userID")
            next.ServeHTTP(w, r)
            return
        } else if err != nil {
            app.serverError(w, err)
            return
        }

        // Otherwise, we know that the request is coming from a valid,
        // authenticated (logged in) user. We create a new copy of the
        // request with the user information added to the request context, and
        // call the next handler in the chain *using this new copy of the
        // request*.
        ctx := context.WithValue(r.Context(), contextKeyUser, user)
        next.ServeHTTP(w, r.WithContext(ctx))
    })
    })
}
```

The important thing to emphasize here is the following difference:

- When we *don't* have an authenticated-and-valid user, we pass the original and unchanged `*http.Request` to the next handler in the chain.
- When we *do* have an authenticated-and-valid user, we create a copy of the request with the user's details stored in the request context. We then pass this copy of the `*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

import (
    "net/http"

    "github.com/bmizerany/pat"
    "github.com/justinas/alice"
)

func (app *application) routes() http.Handler {
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea
    // Add the authenticate() middleware to the chain.
    dynamicMiddleware := alice.New(app.session.Enable, noSurf, app.authenticate

    mux := pat.New()
    mux.Get("/", dynamicMiddleware.ThenFunc(app.home))
    mux.Get("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticati
    mux.Post("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticat
    mux.Get("/snippet/:id", dynamicMiddleware.ThenFunc(app.showSnippet))

    mux.Get("/user/signup", dynamicMiddleware.ThenFunc(app.signupUserForm))
    mux.Post("/user/signup", dynamicMiddleware.ThenFunc(app.signupUser))
    mux.Get("/user/login", dynamicMiddleware.ThenFunc(app.loginUserForm))
    mux.Post("/user/login", dynamicMiddleware.ThenFunc(app.loginUser))
    mux.Post("/user/logout", dynamicMiddleware.Append(app.requireAuthentication

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

    return standardMiddleware.Then(mux)
}
```

So, the next thing we need to do is add an easy way to retrieve the user details from the request context.

Let’s go back to the `authenticatedUser()` helper that we made previously and update it to look like this:

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

```
package main

import (
    "bytes"
    "fmt"
    "net/http"
    "runtime/debug"
    "time"

    "alexedwards.net/snippetbox/pkg/models" // New import
    "github.com/justinas/nosurf"
)

...

func (app *application) authenticatedUser(r *http.Request) *models.User {
    user, ok := r.Context().Value(contextKeyUser).(*models.User)
    if !ok {
        return nil
    }
    return user
}
```

This helper is quite a bit different now. If there is a `*models.User` struct in the request context with the key `contextKeyUser`, then we know that the request is coming from an authenticated-and-valid user — and we return their corresponding user details.

Otherwise, we know that the request isn't coming from an authenticated-and-valid user and we return `nil`.

If you try to run the application at this point, the compiler will fail. We need to update our code in a couple of places to expect a `*models.User` return value from this method — instead of an `int` value like we were returning before.

The first place to update is the `requireAuthenticatedUser()` middleware, which you'll need to change to check for a `nil` return value instead of `0`, like so:

```
File: cmd/web/middleware
```

```
func (app *application) requireAuthenticatedUser(next http.Handler) http.Handle
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Check that the authenticatedUser helper doesn't return nil.
        if app.authenticatedUser(r) == nil {
            http.Redirect(w, r, "/user/login", 302)
            return
        }

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

And the other thing is the `templateData` struct containing the data to pass to our templates:

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

```
package main

import (
    "html/template"
    "path/filepath"
    "time"

    "alexedwards.net/snippetbox/pkg/forms"
    "alexedwards.net/snippetbox/pkg/models"
)

type templateData struct {
    AuthenticatedUser *models.User // Update this to have the type *models.User
    CSRFToken         string
    CurrentYear       int
    Flash              string
    Form              *forms.Form
    Snippet          *models.Snippet
    Snippets         []*models.Snippet
}

...
```

And if you run the application again, everything should now compile correctly.

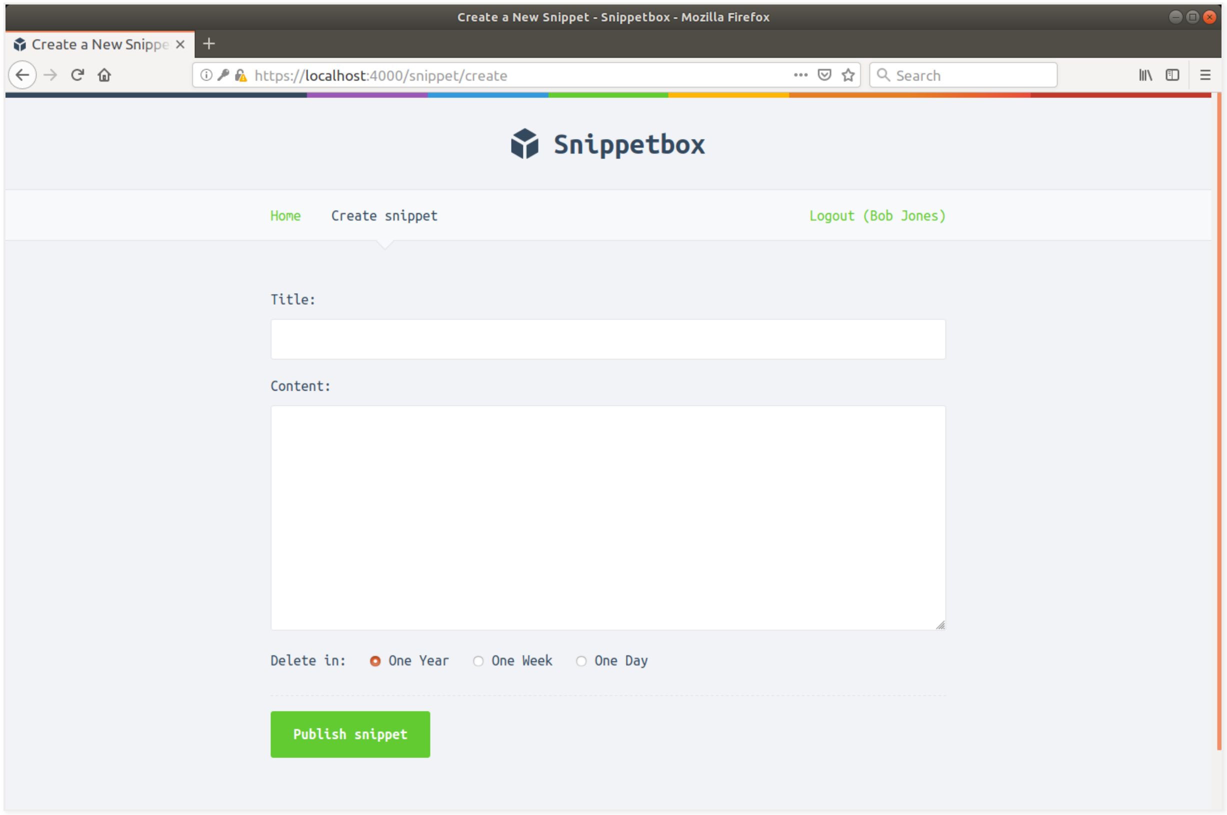
There’s one more little tweak we can make to the application. Because our templates now have access to the `*models.User` struct containing information for the current authenticated-and-valid user, we can display information about them to help make it clear who they are logged in as.

Update the `ui/html/base.layout.tmpl` file so that the user’s name is displayed next to the ‘Logout’ link, like so:

```
File: ui/html/base.layout.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-
        <link rel='stylesheet' href='https://fonts.googleapis.com/css?family=Ub
    </head>
    <body>
        <header>
            <h1><a href='/'>Snippetbox</a></h1>
        </header>
        <nav>
            <div>
                <a href='/'>Home</a>
                {{if .AuthenticatedUser}}
                    <a href='/snippet/create'>Create snippet</a>
                {{end}}
            </div>
            <div>
                {{if .AuthenticatedUser}}
                    <form action='/user/logout' method='POST'>
                        <input type='hidden' name='csrf_token' value='{{.CSRFTo
                        <!-- Include the user's name next to the logout link --
                        <button>Logout ({{.AuthenticatedUser.Name}})</button>
                    </form>
                {{else}}
                    <a href='/user/signup'>Signup</a>
                    <a href='/user/login'>Login</a>
                {{end}}
            </div>
        </nav>
        <section>
            {{with .Flash}}
            <div class='flash '>{{.}}</div>
            {{end}}
            {{template "body" .}}
        </section>
        {{template "footer" .}}
        <script src="/static/js/main.js" type="text/javascript"></script>
    </body>
</html>
{{end}}
```

If you restart the application now it should now look something like this, with the logged in user’s name on the right of the navigation bar.



If you like, you could also open MySQL and delete the user that you're currently logged in as from the `users` table. 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 no longer exists 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).

Chapter 13.

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 were 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.

Chapter 13.1.

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
```

```go
package main

...

func humanDate(t time.Time) string {
    return t.UTC().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, its standard practice to create your tests in `*_test.go` files which live directly alongside 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
```

```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(2020, 12, 17, 10, 0, 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 Dec 2020 at 10:00" {
        t.Errorf("want %q; got %q", "17 Dec 2020 at 10:00", hd)
    }
}
```

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.

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      alexedwards.net/snippetbox/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      alexedwards.net/snippetbox/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 `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
```

```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(2020, 12, 17, 10, 0, 0, 0, time.UTC),
            want: "17 Dec 2020 at 10:00",
        },
        {
            name: "Empty",
            tm:   time.Time{},
            want: "",
        },
        {
            name: "CET",
            tm:   time.Date(2020, 12, 17, 10, 0, 0, 0, time.FixedZone("CET", 1*
            want: "17 Dec 2020 at 09:00",
        },
    }

    // 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
        // and 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("want %q; got %q", tt.want, hd)
            }
        })
    }
}
```

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()` (in UTC) to be `17 Dec 2020 at 09:00`, not `17 Dec 2020 at 10:00`.

OK, let's run this and see what happens:

```
$ go test -v ./cmd/web
=== RUN   TestHumanDate
=== RUN   TestHumanDate/UTC
=== RUN   TestHumanDate/Empty
=== RUN   TestHumanDate/CET
--- FAIL: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- FAIL: TestHumanDate/Empty (0.00s)
        templates_test.go:44: want ""; got "01 Jan 0001 at 00:00"
    --- FAIL: TestHumanDate/CET (0.00s)
        templates_test.go:44: want "17 Dec 2020 at 09:00"; got "17 Dec 2020 at 
FAIL
FAIL    alexedwards.net/snippetbox/cmd/web    0.005s
```

We can see that we get 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?

It also worth pointing out that when we use the `t.Errorf()` function to mark a test as failed, it doesn't cause `go test` to immediately exit. All the other tests and sub-tests will continue to be run after a failure.

As a side note, you can use the `-failfast` flag to stop the tests running after the first failure, if you want, like so:

```
$ go test -failfast -v ./cmd/web
=== RUN   TestHumanDate
=== RUN   TestHumanDate/UTC
=== RUN   TestHumanDate/Empty
--- FAIL: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- FAIL: TestHumanDate/Empty (0.00s)
        templates_test.go:44: want ""; got "01 Jan 0001 at 00:00"
FAIL
FAIL    alexedwards.net/snippetbox/cmd/web    0.007s
```

Anyway… 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      alexedwards.net/snippetbox/cmd/web    0.009s
```

Running All Tests

To run *all* the tests for a project — instead of just those in a specific package — you can use the `./...` wildcard pattern like so:

```
$ go test ./...
```

```
ok      alexedwards.net/snippetbox/cmd/web    0.006s
?       alexedwards.net/snippetbox/pkg/forms    [no test files]
?       alexedwards.net/snippetbox/pkg/models    [no test files]
?       alexedwards.net/snippetbox/pkg/models/mysql    [no test files]
```

Running the Web Application

One problem with the convention of putting your tests in a `*_test.go` file is that is makes it harder to use the `go run` command to start our web application.

If you try to run our web application using the same `go run cmd/web/*` command that we've used for most of this book, you'll now get an error message like this:

```
$ go run cmd/web/*
go run: cannot run *_test.go files (cmd/web/templates_test.go)
```

To get around this, you can use the following bash command to exclude the `*_test.go` files when executing the `go run` command:

```
$ go run cmd/web/!(*_test).go
INFO    2018/12/17 13:37:32 Starting server on :4000
```

Note: If you get an `event not found` error when running this command, you probably need to enable extended globbing in your bash terminal first. You can do this by running `shopt -s extglob`.

Chapter 13.2.

Testing HTTP Handlers

Let's move on and discuss some specific techniques for unit testing your HTTP handlers.

All the handlers that we've written for our Snippetbox project so far are a bit complex to test, and to introduce things I'd prefer to start off with something a bit simpler.

If you're following along, head over to your `handlers.go` file and create a new `ping` handler function which simply returns a `200 OK` response. 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
```

```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

To assist in testing your HTTP handlers Go provides the `net/http/httptest` package, which contains a suite of useful tools.

One of these tools is the `httptest.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 `httptest.ResponseRecorder` object, 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
```

```go
package main

import (
    "io/ioutil"

    "net/http"
    "net/http/httptest"
    "testing"
)

func TestPing(t *testing.T) {
    // Initialize a new httptest.ResponseRecorder.
    rr := httptest.NewRecorder()

    // Initialize a new dummy http.Request.
    r, err := http.NewRequest("GET", "/", nil)
    if err != nil {
        t.Fatal(err)
    }

    // Call the ping handler function, passing in the
    // httptest.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()

    // We can then examine the http.Response to check that the status code
    // written by the ping handler was 200.
    if rs.StatusCode != http.StatusOK {
        t.Errorf("want %d; got %d", http.StatusOK, rs.StatusCode)
    }

    // And we can check that the response body written by the ping handler
    // equals "OK".
    defer rs.Body.Close()
    body, err := ioutil.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }

    if string(body) != "OK" {
        t.Errorf("want body to equal %q", "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 any further tests.

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      alexedwards.net/snippetbox/cmd/web    0.007s
```

So this is looking good. We can see that all the tests we've written under our `cmd/web` package are being run and passing without any problems.

Testing Middleware

It’s also possible to use the same general technique to unit test your middleware.

We’ll demonstrate by creating a `TestSecureHeaders` test for the `secureHeaders` middleware that we made earlier in the book. As part of this test we want to check that:

- The middleware sets the `X-Frame-Options: deny` header.
- The middleware sets the `X-XSS-Protection: 1; mode=block` header.
- The 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 (
    "io/ioutil"
    "net/http"
    "net/http/httptest"
    "testing"
)
```

```
func TestSecureHeaders(t *testing.T) {
    // Initialize a new httptest.ResponseRecorder and dummy http.Request.
    rr := httptest.NewRecorder()

    r, err := http.NewRequest("GET", "/", nil)
    if err != nil {
        t.Fatal(err)
    }

    // Create a mock HTTP handler that we can pass to our secureHeaders
    // middleware, which writes a 200 status code and "OK" response body.
    next := http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("OK"))
    })

    // Pass the mock HTTP handler to our secureHeaders middleware. Because
    // secureHeaders *returns* a http.Handler we can call its ServeHTTP()
    // method, passing in the http.ResponseRecorder and dummy http.Request to
    // execute it.
    secureHeaders(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 X-Frame-Options header
    // on the response.
    frameOptions := rs.Header.Get("X-Frame-Options")
    if frameOptions != "deny" {
        t.Errorf("want %q; got %q", "deny", frameOptions)
    }

    // Check that the middleware has correctly set the X-XSS-Protection header
    // on the response.
    xssProtection := rs.Header.Get("X-XSS-Protection")
    if xssProtection != "1; mode=block" {
        t.Errorf("want %q; got %q", "1; mode=block", xssProtection)
    }
    // Check that the middleware has correctly called the next handler in line
    // and the response status code and body are as expected.
    if rs.StatusCode != http.StatusOK {
        t.Errorf("want %d; got %d", http.StatusOK, rs.StatusCode)
    }

    defer rs.Body.Close()
    body, err := ioutil.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }

    if string(body) != "OK" {
        t.Errorf("want body to equal %q", "OK")
    }
}
```

If you run the tests now, everything should pass without any issues.

```
$ go test -v ./cmd/web/
=== RUN   TestPing
--- PASS: TestPing (0.00s)
=== RUN   TestSecureHeaders
--- PASS: TestSecureHeaders (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      alexedwards.net/snippetbox/cmd/web    0.009s
```

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.

Running Specific Tests

It's possible to only run specific tests by using the `-run` flag. This allows you to pass in a regular expression — and only tests with a name that matches the regular expression will be run.

In our case, we could opt to run only the `TestPing` test like so:

```
$ go test -v -run="^TestPing$" ./cmd/web/
=== RUN   TestPing
--- PASS: TestPing (0.01s)
PASS
ok      alexedwards.net/snippetbox/cmd/web    0.012s
```

And you can even use the `-run` flag to limit testing to some specific sub-tests. For example:

```
$ go test -v -run="^TestHumanDate$/^UTC|CET$" ./cmd/web
=== RUN   TestHumanDate
=== RUN   TestHumanDate/UTC
=== RUN   TestHumanDate/CET
--- PASS: TestHumanDate (0.00s)
    --- PASS: TestHumanDate/UTC (0.00s)
    --- PASS: TestHumanDate/CET (0.00s)
```

```
PASS
ok      alexedwards.net/snippetbox/cmd/web    0.007s
```

Note how, when it comes to running specific sub-tests, the value of the `-run` flag contains multiple regular expressions separated by a `/` character? The first part needs to match the name of the test, and the second part needs to match the name of the sub-test.

Additional Information

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 your tests in parallel.

You can indicate that it's OK for a test to be run in concurrently alongside other tests by calling the `t.Parallel()` function at the start of the test. For example:

```go
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 ./...
  ```

Chapter 13.3.

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. Arguably, 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:

Method	Pattern	Handler	Action
…	…	…	…
GET	/ping	ping	Return a 200 OK 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.

Head back to your `handlers_test.go` file and update the `TestPing` test like so:

File: cmd/web/handlers_test.go

```
package main

import (
    "io/ioutil"
    "log"       // New import

    "net/http"
    "net/http/httptest"
    "testing"
)

func TestPing(t *testing.T) {
    // Create a new instance of our application struct. For now, this just
    // contains a couple of mock loggers (which discard anything written to
    // them).
    app := &application{
        errorLog: log.New(ioutil.Discard, "", 0),
        infoLog:  log.New(ioutil.Discard, "", 0),
    }
```

```go
    // We then use the httptest.NewTLSServer() 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() to shutdown the server when
    // the test finishes.
    ts := httptest.NewTLSServer(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 code as before.
    if rs.StatusCode != http.StatusOK {
        t.Errorf("want %d; got %d", http.StatusOK, rs.StatusCode)
    }

    defer rs.Body.Close()
    body, err := ioutil.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }

    if string(body) != "OK" {
        t.Errorf("want body to equal %q", "OK")
    }
}
```

There are a few things about this code to point out and discuss.

- The `httptest.NewTLSServer()` function accepts a `http.Handler` as the parameter, and this handler gets 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 as the handler. Doing this gives us a test server that exactly mimics our application routes, middleware and handlers, and 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 `httptest.NewServer()` function to create the test server instead.

- You might be wondering at this point why we have set the `errorLog` and `infoLog` fields of our `application` struct, but none of the other fields. The reason for this is that the loggers are 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: want 200; got 404
    handlers_test.go:51: want body to equal "OK"
FAIL
FAIL    alexedwards.net/snippetbox/cmd/web    0.004s
```

If you're following along, you should get a failure at this point.

We can see from the test output that our `GET /ping` request is currently receiving a 404 response, 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
```

```go
package main

import (
    "net/http"

    "github.com/bmizerany/pat"
    "github.com/justinas/alice"
)

func (app *application) routes() http.Handler {
    standardMiddleware := alice.New(app.recoverPanic, app.logRequest, secureHea
    dynamicMiddleware := alice.New(app.session.Enable, noSurf, app.authenticate

    mux := pat.New()
    mux.Get("/", dynamicMiddleware.ThenFunc(app.home))
    mux.Get("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticate
    mux.Post("/snippet/create", dynamicMiddleware.Append(app.requireAuthenticat
    mux.Get("/snippet/:id", dynamicMiddleware.ThenFunc(app.showSnippet))
    mux.Get("/user/signup", dynamicMiddleware.ThenFunc(app.signupUserForm))
    mux.Post("/user/signup", dynamicMiddleware.ThenFunc(app.signupUser))
    mux.Get("/user/login", dynamicMiddleware.ThenFunc(app.loginUserForm))
    mux.Post("/user/login", dynamicMiddleware.ThenFunc(app.loginUser))
    mux.Post("/user/logout", dynamicMiddleware.Append(app.requireAuthenticatedU

    // Register the ping handler function as the handler for the GET /ping
    // route.
    mux.Get("/ping", http.HandlerFunc(ping))
    fileServer := http.FileServer(http.Dir("./ui/static/"))
    mux.Get("/static/", http.StripPrefix("/static", fileServer))

    return standardMiddleware.Then(mux)
}
```

And if you run the tests again everything should now pass.

```
$ go test ./cmd/web/
ok      alexedwards.net/snippetbox/cmd/web    0.008s
```

Using Test Helpers

Our `TestPing` function is now working nicely. But there's a good opportunity to break out some of this code into some 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 `pkg/testutils` (or similar) which can be imported by your test files.

In our case the helpers will only be used in our `cmd/web` package and we'll put them in a new `cmd/web/testutils_test.go` file.

Go ahead and create it…

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

And then add the following code:

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

```go
package main

import (
    "io/ioutil"
    "log"
    "net/http"
    "net/http/httptest"
    "testing"
)

// Create a newTestApplication helper which returns an instance of our
// application struct containing mocked dependencies.
func newTestApplication(t *testing.T) *application {
    return &application{
        errorLog: log.New(ioutil.Discard, "", 0),
        infoLog:  log.New(ioutil.Discard, "", 0),
    }
}

// Define a custom testServer type which anonymously embeds a httptest.Server
// instance.
type testServer struct {
    *httptest.Server
}

// Create a newTestServer helper which initalizes and returns a new instance
// of our custom testServer type.
```

```
func newTestServer(t *testing.T, h http.Handler) *testServer {
    ts := httptest.NewTLSServer(h)
    return &testServer{ts}
}

// Implement a get method on our custom testServer type. This makes a GET
// request to a given url path on the test server, and returns the response
// status code, headers and body.
func (ts *testServer) get(t *testing.T, urlPath string) (int, http.Header, []by
    rs, err := ts.Client().Get(ts.URL + urlPath)
    if err != nil {
        t.Fatal(err)
    }

    defer rs.Body.Close()
    body, err := ioutil.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }

    return rs.StatusCode, rs.Header, body
}
```

Hopefully it's clear that this is just a generalization of the code we've already written in this chapter to spin up a test server and make a `GET` request against it.

Then let's head back to our `TestPing` handler and update it to use these new helpers:

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

```
package main
```

```go
import (
    "net/http"
    "testing"
)

func TestPing(t *testing.T) {
    app := newTestApplication(t)
    ts := newTestServer(t, app.routes())
    defer ts.Close()

    code, _, body := ts.get(t, "/ping")

    if code != http.StatusOK {
        t.Errorf("want %d; got %d", http.StatusOK, code)
    }

    if string(body) != "OK" {
        t.Errorf("want body to equal %q", "OK")
    }
}
```

And, again, if you run the tests again everything should still pass.

```
$ go test ./cmd/web/
ok      alexedwards.net/snippetbox/cmd/web    0.013s
```

This is shaping up nicely. We now 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

In this chapter we've been using the `ts.Client().Get()` method to make requests against our test server. The `ts.Client()` part of this code returns a configurable `http.Client`, which we can change to make these requests work slightly differently.

There's a couple of changes I'd like to make to the default `http.Client` so that it's better suited to testing our web application. Specifically:

- 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*.
- 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.

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
```

```go
package main

import (
    "io/ioutil"
    "log"
    "net/http"
    "net/http/cookiejar" // New import
    "net/http/httptest"
    "testing"
)

...

func newTestServer(t *testing.T, h http.Handler) *testServer {
    ts := httptest.NewTLSServer(h)

    // Initialize a new cookie jar.
    jar, err := cookiejar.New(nil)
    if err != nil {
        t.Fatal(err)
    }
    // Add the cookie jar to the client, so that response cookies are stored
    // and then sent with subsequent requests.
    ts.Client().Jar = jar

    // Disable redirect-following for the client. Essentially this function
    // is called after a 3xx response is received by the client, and returning
    // the http.ErrUseLastResponse error forces it to immediately return the
    // received response.
    ts.Client().CheckRedirect = func(req *http.Request, via []*http.Request) er
        return http.ErrUseLastResponse
    }

    return &testServer{ts}
}

...
```

Chapter 13.4.

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 `showSnippet` handler and `GET /snippet/: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 {
    errorLog      *log.Logger
    infoLog       *log.Logger
    session       *sessions.Session
    snippets      *mysql.SnippetModel
    templateCache map[string]*template.Template
    users         *mysql.UserModel
}
```

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 `errorLog` and `infoLog` dependencies with loggers that write messages to ioutil.Discard, instead of the os.Stdout and os.Stderr streams like we do in our production application:

```
func newTestApplication(t *testing.T) *application {
    return &application{
        errorLog: log.New(ioutil.Discard, "", 0),
        infoLog:  log.New(ioutil.Discard, "", 0),
    }
}
```

The reason for mocking these and writing to ioutil.Discard is to avoid clogging up our test output with unnecessary log messages when we run go test -v.

Note: Depending on your background and programming experiences, you might not consider these loggers to be *mocks*. You might call them *fakes*, *stubs* 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 mysql.SnippetModel and mysql.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 `pkg/models/mock` package and two new `snippets.go` and `users.go` files to hold the database model mocks, like so:

```
$ mkdir pkg/models/mock
$ touch pkg/models/mock/snippets.go
$ touch pkg/models/mock/users.go
```

Let's begin by creating a mock of our `mysql.SnippetModel`. To do this, we're going to create a simple struct which implements the same methods as our production `mysql.SnippetModel`, but have the methods return some fixed dummy data instead.

```
File: pkg/models/mock/snippets.go
```

```go
package mock

import (
    "time"

    "alexedwards.net/snippetbox/pkg/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, content, expires string) (int, error) {
    return 2, nil
}

func (m *SnippetModel) Get(id int) (*models.Snippet, error) {
    switch id {
    case 1:
        return mockSnippet, nil
    default:
        return nil, models.ErrNoRecord
    }
}

func (m *SnippetModel) Latest() ([]*models.Snippet, error) {
    return []*models.Snippet{mockSnippet}, nil
}
```

And let’s do the same for our `mysql.UserModel`, like so:

```
File: pkg/models/mock/users.go
```

```
package mock

import (
    "time"

    "alexedwards.net/snippetbox/pkg/models"
)

var mockUser = &models.User{
    ID:      1,
    Name:    "Alice",
    Email:   "alice@example.com",
    Created: time.Now(),
}
```

```
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) {
    switch email {
    case "alice@example.com":
        return 1, nil
    default:
        return 0, models.ErrInvalidCredentials
    }
}

func (m *UserModel) Get(id int) (*models.User, error) {
    switch id {
    case 1:
        return mockUser, nil
    default:
        return nil, models.ErrNoRecord
    }
}
```

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
```

```go
package main

import (
    "io/ioutil"
    "log"
    "net/http"
    "net/http/cookiejar"
    "net/http/httptest"
    "testing"
    "time" // New import

    "alexedwards.net/snippetbox/pkg/models/mock" // New import

    "github.com/golangcollege/sessions" // New import
)

func newTestApplication(t *testing.T) *application {
    // Create an instance of the template cache.
    templateCache, err := newTemplateCache("./../../ui/html/")
    if err != nil {
        t.Fatal(err)
    }

    // Create a session manager instance, with the same settings as production.
    session := sessions.New([]byte("3dSm5MnygFHh7XidAtbskXrjbwfoJcbJ"))
    session.Lifetime = 12 * time.Hour
    session.Secure = true

    // Initialize the dependencies, using the mocks for the loggers and
    // database models.
    return &application{
        errorLog:      log.New(ioutil.Discard, "", 0),
        infoLog:       log.New(ioutil.Discard, "", 0),
        session:       session,
        snippets:      &mock.SnippetModel{},
        templateCache: templateCache,
        users:         &mock.UserModel{},
    }
}

...
```

Important: When you run your tests with `go test` the current working directory will be set to the package directory for the *currently executing test*. So, if you're reading in any files as part of your tests, you need to make sure that the file path is relative to that particular file.

If you go ahead and try to run the tests now, it will fail with the following messages:

```
$ go test ./cmd/web
# alexedwards.net/snippetbox/cmd/web [alexedwards.net/snippetbox/cmd/web.test]
cmd/web/testutils_test.go:28:3: cannot use mock.SnippetModel literal (type *mock
cmd/web/testutils_test.go:30:3: cannot use mock.UserModel literal (type *mock.Us
FAIL    alexedwards.net/snippetbox/cmd/web [build failed]
```

This is happening because our `application` struct is expecting pointers to `mysql.SnippetModel` and `mysql.UserModel` objects, but we are trying to use pointers to `mock.SnippetModel` and `mock.UserModel` objects instead.

The idiomatic fix for this is to change our `application` struct so that it uses *interfaces* instead, which are satisfied by both our mock and production database models.

Hint: If you're not familiar with the idea of *interfaces* in Go, then there is a good introduction in this blog post which I recommend reading.

A nice pattern, which helps to keep your code simple, is to define the interfaces *inline* in the `application` struct, like so:

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

```
package main

import (
    "crypto/tls"
    "database/sql"
    "flag"
    "html/template"
    "log"
    "net/http"
    "os"
    "time"

    "alexedwards.net/snippetbox/pkg/models" // New import
    "alexedwards.net/snippetbox/pkg/models/mysql"

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

type contextKey string

var contextKeyUser = contextKey("user")


type application struct {
    errorLog *log.Logger
    infoLog  *log.Logger
    session  *sessions.Session
    snippets interface {
        Insert(string, string, string) (int, error)
        Get(int) (*models.Snippet, error)
        Latest() ([]*models.Snippet, error)
    }
    templateCache map[string]*template.Template
    users         interface {
        Insert(string, string, string) error
        Authenticate(string, string) (int, error)
        Get(int) (*models.User, error)
    }
}

...
```

And if you try running the tests again now, everything should work correctly.

```
$ go test ./cmd/web/
ok      alexedwards.net/snippetbox/cmd/web    0.027s
```

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 `*mysql.SnippetModel` and `*mysql.UserModel` they are *interfaces* instead.

So long as an object has the necessary methods to satisfy the interface, we can use them in our `application` struct. Both our 'production' database models (under the `pkg/models/mysql` package) and mock database models (under the `pkg/models/mock` package) satisfy the interfaces, so we can now use them interchangeably.

Testing the showSnippet Handler

With that all now set up, let's get stuck into writing an end-to-end test for our `GET /snippet/:id` route which uses these mocked dependencies.

As part of this test, the code in our `showSnippet` 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:

- For the request `GET /snippet/1` we receive a `200 OK` response with the relevant mocked snippet in the HTML response body.
- For all other requests to `GET /snippet/*` we should receive a `404 Not Found` response.

Open up the `cmd/web/handlers_test.go` file and create a new `TestShowSnippet` test like so:

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

```
package main

import (
    "bytes" // New import
    "net/http"
    "testing"
)

...

func TestShowSnippet(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 []byte
    }{
        {"Valid ID", "/snippet/1", http.StatusOK, []byte("An old silent pond...
        {"Non-existent ID", "/snippet/2", http.StatusNotFound, nil},
        {"Negative ID", "/snippet/-1", http.StatusNotFound, nil},
        {"Decimal ID", "/snippet/1.23", http.StatusNotFound, nil},
        {"String ID", "/snippet/foo", http.StatusNotFound, nil},
        {"Empty ID", "/snippet/", http.StatusNotFound, nil},
        {"Trailing slash", "/snippet/1/", http.StatusNotFound, nil},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            code, _, body := ts.get(t, tt.urlPath)

            if code != tt.wantCode {
                t.Errorf("want %d; got %d", tt.wantCode, code)
            }

            if !bytes.Contains(body, tt.wantBody) {
                t.Errorf("want body to contain %q", tt.wantBody)
            }
        })
    }
}
```

When you run the tests again, everything should pass and you'll see some output including the new `TestShowSnippet` tests and which looks a bit like this:

```
$ go test -v ./cmd/web/
=== RUN   TestPing
--- PASS: TestPing (0.01s)
=== RUN   TestShowSnippet
=== RUN   TestShowSnippet/Valid_ID
=== RUN   TestShowSnippet/Non-existent_ID
=== RUN   TestShowSnippet/Negative_ID
=== RUN   TestShowSnippet/Decimal_ID
=== RUN   TestShowSnippet/String_ID
=== RUN   TestShowSnippet/Empty_ID
=== RUN   TestShowSnippet/Trailing_slash
--- PASS: TestShowSnippet (0.01s)
    --- PASS: TestShowSnippet/Valid_ID (0.00s)
    --- PASS: TestShowSnippet/Non-existent_ID (0.00s)
    --- PASS: TestShowSnippet/Negative_ID (0.00s)
    --- PASS: TestShowSnippet/Decimal_ID (0.00s)
    --- PASS: TestShowSnippet/String_ID (0.00s)
    --- PASS: TestShowSnippet/Empty_ID (0.00s)
    --- PASS: TestShowSnippet/Trailing_slash (0.00s)
=== RUN   TestSecureHeaders
--- PASS: TestSecureHeaders (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      alexedwards.net/snippetbox/cmd/web    0.029s
```

There's one new thing to point out here: notice how the names of the sub-tests have been canonicalized? Any spaces in the sub-test name have been replaced 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 `signupUser` 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 emulate 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 a CSRF token in the HTML 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 a HTML response body:

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

```go
package main

import (
    "html" // New import
    "io/ioutil"
    "log"
    "net/http"
    "net/http/cookiejar"
    "net/http/httptest"
    "regexp" // New import
    "testing"
    "time"

    "alexedwards.net/snippetbox/pkg/models/mock"

    "github.com/golangcollege/sessions"
)

// 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' va

func extractCSRFToken(t *testing.T, body []byte) string {
    // Use the FindSubmatch 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.FindSubmatch(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 potentially includes the `+` character, and this will be escaped to `&#43;`. So after extracting the token from the HTML we need to run it through `html.UnescapeString()` to get the original token value.

Once that's done, let's go back to our `cmd/web/handlers_test.go` file and create a new `TestSignupUser` function.

To start with, we'll make this perform a `GET /user/signup` request and then extract and print the CSRF token from the HTML response body, like so:

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

```
package main

...

func TestSignupUser(t *testing.T) {
    // Create the application struct containing our mocked dependencies and set
    // up the test server for running and 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.
    t.Log(csrfToken)
}
```

And if you run the test you should see output similar to this, with the CSRF token from the HTML response logged in the output:

```
$ go test -v -run="TestSignupUser" ./cmd/web/
=== RUN   TestSignupUser
--- PASS: TestSignupUser (0.01s)
    handlers_test.go:79: pgSoIO8oeTTv5i1XpFHcZpVUU/oEjgUg6etZeZnoHPq+zi8aokP8OPs
PASS
ok      alexedwards.net/snippetbox/cmd/web    0.015s
```

Note: To see the output from the `t.Log()` command you need to run `go test` with the `-v` (verbose) flag enabled.

Another note: If you run this test for a second time immediately afterwards, you'll likely get the same CSRF token *because the test results have been cached*. This is because Go is smart enough to know that nothing has changed since you previously ran the test, so it will return a cached test result instead of running it again. If you want, you can force the test to run again by using the `-count=1` flag like so:

```
$ go test -v -run="TestSignupUser" -count=1 ./cmd/web/
```

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` object, which we can use to send `POST` requests and a given request body to our test server.

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
```

```go
package main

import (
    "html"
    "io/ioutil"
    "log"
    "net/http"
    "net/http/cookiejar"
    "net/http/httptest"
    "net/url" // New import
    "regexp"
    "testing"
    "time"

    "alexedwards.net/snippetbox/pkg/models/mock"

    "github.com/golangcollege/sessions"
)

...

type testServer struct {
    *httptest.Server
}
```

```
...

// 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 data that you want to send in the request body.
func (ts *testServer) postForm(t *testing.T, urlPath string, form url.Values) (
    rs, err := ts.Client().PostForm(ts.URL+urlPath, form)
    if err != nil {
        t.Fatal(err)
    }

    // Read the response body.
    defer rs.Body.Close()
    body, err := ioutil.ReadAll(rs.Body)
    if err != nil {
        t.Fatal(err)
    }

    // Return the response status, headers and body.
    return rs.StatusCode, rs.Header, 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:

- Requests with empty name, email or password fields result in the signup form being redisplayed with the message "This field cannot be blank".
- Requests with an invalid email field result in the signup form being redisplayed with the message "This field is invalid".

- Requests with a password field shorter than 10 characters result in the signup form being redisplayed with the message “This field is too short (minimum is 10 characters)”.
- Requests to signup with an email which is already in use result in the signup form being redisplayed with the message “Address is already in use”.
- 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.

Go ahead and update the `TestSignupUser` function like so:

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

```go
package main

import (
    "bytes"
    "net/http"
    "net/url" // New import
    "testing"
)

...

func TestSignupUser(t *testing.T) {
    app := newTestApplication(t)
    ts := newTestServer(t, app.routes())
    defer ts.Close()

    _, _, body := ts.get(t, "/user/signup")
    csrfToken := extractCSRFToken(t, body)
    tests := []struct {
        name         string
        userName     string
        userEmail    string
        userPassword string
        csrfToken    string
        wantCode     int
        wantBody     []byte
    }{
        {"Valid submission", "Bob", "bob@example.com", "validPa$$word", csrfTok
        {"Empty name", "", "bob@example.com", "validPa$$word", csrfToken, http.
        {"Empty email", "Bob", "", "validPa$$word", csrfToken, http.StatusOK, [
        {"Empty password", "Bob", "bob@example.com", "", csrfToken, http.Status
        {"Invalid email (incomplete domain)", "Bob", "bob@example.", "validPa$$
        {"Invalid email (missing @)", "Bob", "bobexample.com", "validPa$$word",
        {"Invalid email (missing local part)", "Bob", "@example.com", "validPa$
        {"Short password", "Bob", "bob@example.com", "pa$$word", csrfToken, htt
        {"Duplicate email", "Bob", "dupe@example.com", "validPa$$word", csrfTok
        {"Invalid CSRF Token", "", "", "", "wrongToken", http.StatusBadRequest,
    }

    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)

            if code != tt.wantCode {
                t.Errorf("want %d; got %d", tt.wantCode, code)
            }

            if !bytes.Contains(body, tt.wantBody) {
                t.Errorf("want body %s to contain %q", body, tt.wantBody)
            }
        })
    }
}
```

If you run the test, you should get passing output like this:

```
$ go test -v -run="TestSignupUser" ./cmd/web/
=== RUN   TestSignupUser
=== RUN   TestSignupUser/Valid_submission
=== RUN   TestSignupUser/Empty_name
=== RUN   TestSignupUser/Empty_email
=== RUN   TestSignupUser/Empty_password
=== RUN   TestSignupUser/Invalid_email_(incomplete_domain)
=== RUN   TestSignupUser/Invalid_email_(missing_@)
=== RUN   TestSignupUser/Invalid_email_(missing_local_part)
=== RUN   TestSignupUser/Short_password
=== RUN   TestSignupUser/Duplicate_email
=== RUN   TestSignupUser/Invalid_CSRF_Token
--- PASS: TestSignupUser (0.01s)
    --- PASS: TestSignupUser/Valid_submission (0.00s)
    --- PASS: TestSignupUser/Empty_name (0.00s)
    --- PASS: TestSignupUser/Empty_email (0.00s)
    --- PASS: TestSignupUser/Empty_password (0.00s)
    --- PASS: TestSignupUser/Invalid_email_(incomplete_host) (0.00s)
    --- PASS: TestSignupUser/Invalid_email_(missing_@) (0.00s)
    --- PASS: TestSignupUser/Invalid_email_(missing_local_part) (0.00s)
    --- PASS: TestSignupUser/Short_password (0.00s)
    --- PASS: TestSignupUser/Duplicate_email (0.00s)
    --- PASS: TestSignupUser/Invalid_CSRF_Token (0.00s)
PASS
ok      alexedwards.net/snippetbox/cmd/web    0.015s
```

Chapter 13.6.

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 production 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 `mysql.UserModel.Get()` 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_c
CREATE USER 'test_web'@'localhost';
GRANT CREATE, DROP, ALTER, INDEX, SELECT, INSERT, UPDATE, DELETE ON test_snippe
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 any 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 `pkg/models/mysql/testdata` directory like so:

```
$ mkdir pkg/models/mysql/testdata
$ touch pkg/models/mysql/testdata/setup.sql
$ touch pkg/models/mysql/testdata/teardown.sql
```

```
File: pkg/models/mysql/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.eS206TNfkGfr6HzGJSWG',
    '2018-12-23 17:25:22'
);
```

```
File: pkg/models/mysql/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. Fun fact: The Go tool also ignores any directories or files which have names that begin with an `_` or `.` character.

Alright, now that we’ve got the scripts in place let’s create a new file to hold some helper functions for our integration tests:

```
$ touch pkg/models/mysql/testutils_test.go
```

And add to it 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;
- Returns an anonymous function which executes the `teardown.sql` script and closes the connection pool.

```
File: pkg/models/mysql/testutils_test.sql
```

```
package mysql

import (
    "database/sql"
    "io/ioutil"
    "testing"
)

func newTestDB(t *testing.T) (*sql.DB, func()) {
    // 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
    if err != nil {
        t.Fatal(err)
    }

    // Read the setup SQL script from file and execute the statements.
    script, err := ioutil.ReadFile("./testdata/setup.sql")
    if err != nil {
        t.Fatal(err)
    }
    _, err = db.Exec(string(script))
    if err != nil {
        t.Fatal(err)
    }

    // Return the connection pool and an anonymous function which reads and
    // executes the teardown script, and closes the connection pool. We can
    // assign this anonymous function and call it later once our test has
    // completed.
    return db, func() {
        script, err := ioutil.ReadFile("./testdata/teardown.sql")
        if err != nil {
            t.Fatal(err)
        }
        _, err = db.Exec(string(script))
        if err != nil {
            t.Fatal(err)
        }

        db.Close()
    }
}
```

Testing the UserModel.Get Method

Now the preparatory work is done we're ready to actually write our integration test for the `mysql.UserModel.Get()` 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 `mysql.UserModel.Get(1)` returns a `models.User` struct containing the details of this user.
- Calling `mysql.UserModel.Get()` with any other user ID returns a `models.ErrNoRecord` error.

Let's follow the Go conventions and create a new file for this test directly alongside the code being tested like so:

```
$ touch pkg/models/mysql/users_test.go
```

And add a `TestUserModelGet` function containing the following code:

```
File: pkg/models/mysql/user_test.go
```

```go
package mysql

import (
    "reflect"
    "testing"
    "time"

    "alexedwards.net/snippetbox/pkg/models"
)

func TestUserModelGet(t *testing.T) {
    // Skip the test if the `-short` flag is provided when running the test.
    // We'll talk more about this in a moment.
    if testing.Short() {
        t.Skip("mysql: skipping integration test")
    }

    // Set up a suite of table-driven tests and expected results.
    tests := []struct {
        name      string
        userID    int
        wantUser  *models.User
        wantError error
    }{
        {
            name:   "Valid ID",
            userID: 1,
            wantUser: &models.User{
                ID:      1,
                Name:    "Alice Jones",
                Email:   "alice@example.com",
                Created: time.Date(2018, 12, 23, 17, 25, 22, 0, time.UTC),
            },
            wantError: nil,
        },
        {
            name:      "Zero ID",
            userID:    0,
            wantUser:  nil,
            wantError: models.ErrNoRecord,
        },
        {
            name:      "Non-existent ID",
            userID:    2,
            wantUser:  nil,
            wantError: models.ErrNoRecord,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            // Initialize a connection pool to our test database, and defer a
            // call to the teardown function, so it is always run immediately
            // before this sub-test returns.
            db, teardown := newTestDB(t)
            defer teardown()

            // Create a new instance of the UserModel.
            m := UserModel{db}

            // Call the UserModel.Get() method and check that the return value
            // and error match the expected values for the sub-test.
            user, err := m.Get(tt.userID)

            if err != tt.wantError {
                t.Errorf("want %v; got %s", tt.wantError, err)
            }

            if !reflect.DeepEqual(user, tt.wantUser) {
                t.Errorf("want %v; got %v", tt.wantUser, user)
            }
        })

    }
}
```

Hint: Using the `reflect.DeepEqual()` function, like we are in the code above, is an effective way to check for equality between arbitrarily complex custom types.

If you run this test, then everything should pass without any problems.

```
$ go test -v ./pkg/models/mysql
=== RUN   TestUserModelGet
=== RUN   TestUserModelGet/Valid_ID
=== RUN   TestUserModelGet/Zero_ID
=== RUN   TestUserModelGet/Non-existent_ID
--- PASS: TestUserModelGet (0.46s)
    --- PASS: TestUserModelGet/Valid_ID (0.16s)
    --- PASS: TestUserModelGet/Zero_ID (0.16s)
    --- PASS: TestUserModelGet/Non-existent_ID (0.15s)
PASS
ok      alexedwards.net/snippetbox/pkg/models/mysql    0.466s
```

The last line in the test output here is worth a mention. The total runtime for this test (0.466 seconds in my case) is much longer than for our previous tests — all of which took a few milliseconds to run. This big increase is runtime here is primarily due to the large number of round trips to the MySQL database that we made during the tests.

If you're running hundreds of integration tests against your database, you might end up 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, like we have in the code above.

When we run our tests with the `-short` flag, then our `TestUserModelGet` test will be skipped.

```
$ go test -v -short ./
$ go test -v -short ./...
=== RUN   TestPing
--- PASS: TestPing (0.01s)
=== RUN   TestShowSnippet

=== RUN   TestShowSnippet/Valid_ID
=== RUN   TestShowSnippet/Non-existent_ID
=== RUN   TestShowSnippet/Negative_ID
=== RUN   TestShowSnippet/Decimal_ID
=== RUN   TestShowSnippet/String_ID
=== RUN   TestShowSnippet/Empty_ID
=== RUN   TestShowSnippet/Trailing_slash
--- PASS: TestShowSnippet (0.01s)
    --- PASS: TestShowSnippet/Valid_ID (0.00s)
    --- PASS: TestShowSnippet/Non-existent_ID (0.00s)
    --- PASS: TestShowSnippet/Negative_ID (0.00s)
    --- PASS: TestShowSnippet/Decimal_ID (0.00s)
    --- PASS: TestShowSnippet/String_ID (0.00s)
    --- PASS: TestShowSnippet/Empty_ID (0.00s)
    --- PASS: TestShowSnippet/Trailing_slash (0.00s)
=== RUN   TestSignupUser
=== RUN   TestSignupUser/Valid_submission
=== RUN   TestSignupUser/Empty_name
=== RUN   TestSignupUser/Empty_email
=== RUN   TestSignupUser/Empty_password
=== RUN   TestSignupUser/Invalid_email_(incomplete_domain)
=== RUN   TestSignupUser/Invalid_email_(missing_@)
=== RUN   TestSignupUser/Invalid_email_(missing_local_part)
=== RUN   TestSignupUser/Short_password
=== RUN   TestSignupUser/Duplicate_email
=== RUN   TestSignupUser/Invalid_CSRF_Token
--- PASS: TestSignupUser (0.02s)
    --- PASS: TestSignupUser/Valid_submission (0.00s)
    --- PASS: TestSignupUser/Empty_name (0.00s)
    --- PASS: TestSignupUser/Empty_email (0.00s)
    --- PASS: TestSignupUser/Empty_password (0.00s)
    --- PASS: TestSignupUser/Invalid_email_(incomplete_domain) (0.00s)
    --- PASS: TestSignupUser/Invalid_email_(missing_@) (0.00s)
    --- PASS: TestSignupUser/Invalid_email_(missing_local_part) (0.00s)
    --- PASS: TestSignupUser/Short_password (0.00s)
        PASS: TestSignupUser/Duplicate_email (0.00s)
    --- PASS: TestSignupUser/Duplicate_email (0.00s)
    --- PASS: TestSignupUser/Invalid_CSRF_Token (0.00s)
=== RUN   TestSecureHeaders
--- PASS: TestSecureHeaders (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      alexedwards.net/snippetbox/cmd/web    0.053s
?       alexedwards.net/snippetbox/pkg/forms    [no test files]
?       alexedwards.net/snippetbox/pkg/models    [no test files]
?       alexedwards.net/snippetbox/pkg/models/mock    [no test files]
=== RUN   TestUserModelGet
--- SKIP: TestUserModelGet (0.00s)
    users_test.go:15: mysql: skipping integration test
PASS
ok      alexedwards.net/snippetbox/pkg/models/mysql    0.003s
```

Otherwise, if you don’t use the `-short` flag, then the test will be executed as normal.

Chapter 13.7.

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 ./...
ok      alexedwards.net/snippetbox/cmd/web    (cached)    coverage: 48.3% of sta
?       alexedwards.net/snippetbox/pkg/forms    [no test files]
?       alexedwards.net/snippetbox/pkg/models    [no test files]
?       alexedwards.net/snippetbox/pkg/models/mock    [no test files]
ok      alexedwards.net/snippetbox/pkg/models/mysql    (cached)    coverage: 10.
```

From the results here we can see that 48.3% of the statements in our `cmd/web` package are executed during our tests, and for our `pkg/models/mysql` the figure is 10.9%.

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 — write a *coverage profile* to a specific location (in our case `/tmp/profile.out`).

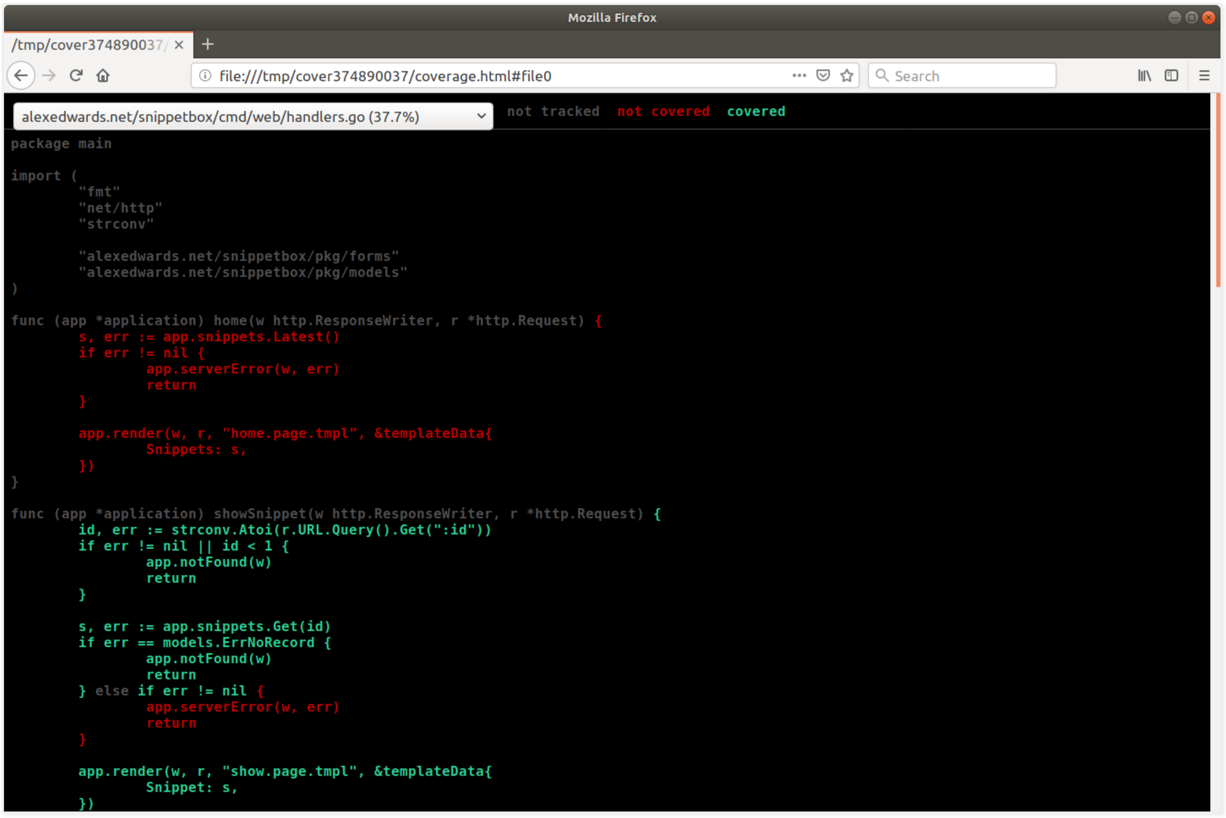
You can then view the coverage profile by using the `go tool cover` command like so:

```
$ go tool cover -func=/tmp/profile.out
alexedwards.net/snippetbox/cmd/web/handlers.go:12:            home
alexedwards.net/snippetbox/cmd/web/handlers.go:24:            showSnippet
alexedwards.net/snippetbox/cmd/web/handlers.go:45:            createSnippetForm
alexedwards.net/snippetbox/cmd/web/handlers.go:51:            createSnippet
alexedwards.net/snippetbox/cmd/web/handlers.go:79:            signupUserForm
alexedwards.net/snippetbox/cmd/web/handlers.go:85:            signupUser
alexedwards.net/snippetbox/cmd/web/handlers.go:117:            loginUserForm
alexedwards.net/snippetbox/cmd/web/handlers.go:123:            loginUser
alexedwards.net/snippetbox/cmd/web/handlers.go:146:            logoutUser
alexedwards.net/snippetbox/cmd/web/handlers.go:152:            ping
alexedwards.net/snippetbox/cmd/web/helpers.go:15:            serverError
alexedwards.net/snippetbox/cmd/web/helpers.go:22:            clientError
alexedwards.net/snippetbox/cmd/web/helpers.go:26:            notFound
alexedwards.net/snippetbox/cmd/web/helpers.go:30:            addDefaultData
alexedwards.net/snippetbox/cmd/web/helpers.go:42:            render
alexedwards.net/snippetbox/cmd/web/helpers.go:60:            authenticatedUser
alexedwards.net/snippetbox/cmd/web/main.go:41:                main
alexedwards.net/snippetbox/cmd/web/main.go:94:                openDB
alexedwards.net/snippetbox/cmd/web/middleware.go:13:        secureHeaders
alexedwards.net/snippetbox/cmd/web/middleware.go:22:        logRequest
alexedwards.net/snippetbox/cmd/web/middleware.go:30:        recoverPanic
alexedwards.net/snippetbox/cmd/web/middleware.go:43:        requireAuthenticate
alexedwards.net/snippetbox/cmd/web/middleware.go:54:        noSurf
alexedwards.net/snippetbox/cmd/web/middleware.go:65:        authenticate
alexedwards.net/snippetbox/cmd/web/routes.go:10:            routes
alexedwards.net/snippetbox/cmd/web/templates.go:22:            humanDate
alexedwards.net/snippetbox/cmd/web/templates.go:34:            newTemplateCache
alexedwards.net/snippetbox/pkg/models/mysql/snippets.go:13:    Insert
alexedwards.net/snippetbox/pkg/models/mysql/snippets.go:30:    Get
alexedwards.net/snippetbox/pkg/models/mysql/snippets.go:47:    Latest
alexedwards.net/snippetbox/pkg/models/mysql/users.go:17:     Insert
alexedwards.net/snippetbox/pkg/models/mysql/users.go:37:     Authenticate
alexedwards.net/snippetbox/pkg/models/mysql/users.go:58:     Get
total:                                                         (statements)
```

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:



Mozilla Firefox
/tmp/cover374890037/
file:///tmp/cover374890037/coverage.html#file0
Search
alexedwards.net/snippetbox/cmd/web/handlers.go (37.7%)
not tracked
not covered
covered

```
package main

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

	"alexedwards.net/snippetbox/pkg/forms"
	"alexedwards.net/snippetbox/pkg/models"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
	s, err := app.snippets.Latest()
	if err != nil {
		app.serverError(w, err)
		return
	}

	app.render(w, r, "home.page.tmpl", &templateData{
		Snippets: s,
	})
}

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
	id, err := strconv.Atoi(r.URL.Query().Get(":id"))
	if err != nil || id < 1 {
		app.notFound(w)
		return
	}

	s, err := app.snippets.Get(id)
	if err == models.ErrNoRecord {
		app.notFound(w)
		return
	} else if err != nil {
		app.serverError(w, err)
		return
	}

	app.render(w, r, "show.page.tmpl", &templateData{
		Snippet: s,
	})
```

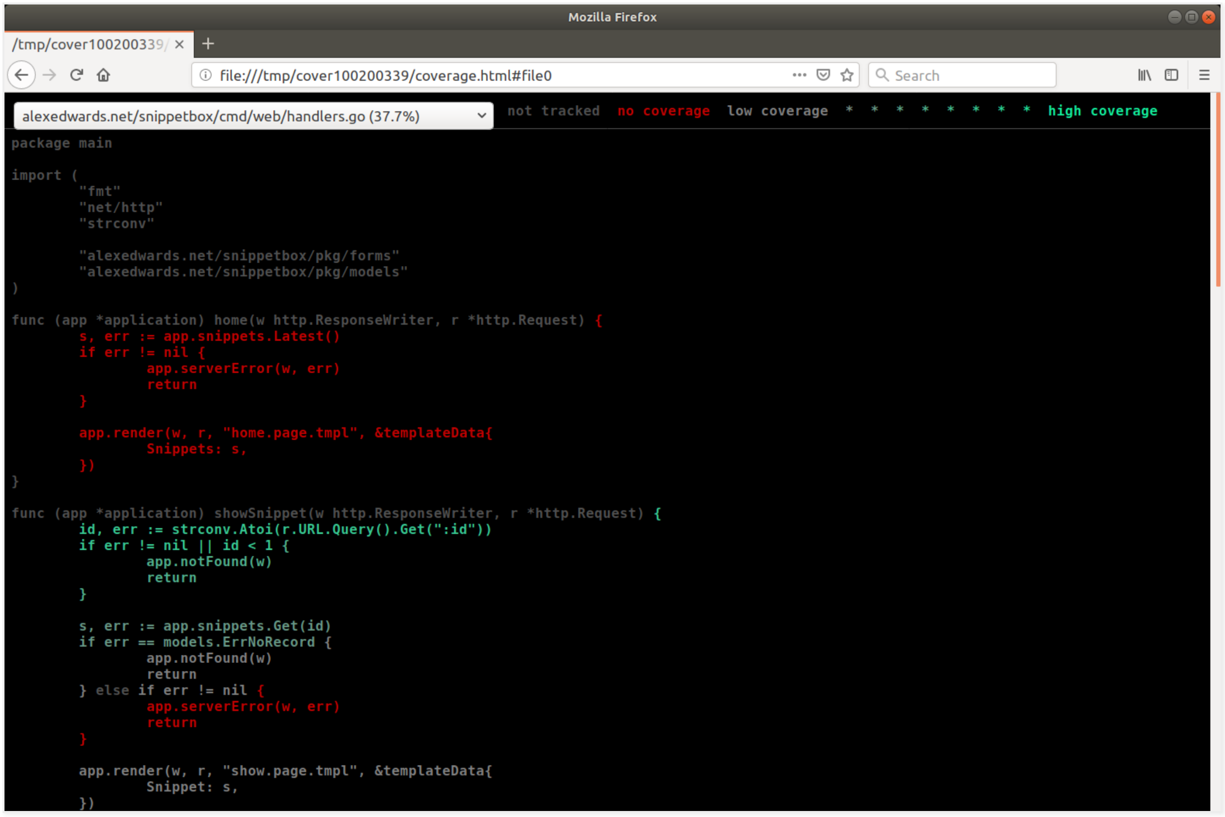
It’s easy to see exactly which statements get executed during your tests (colored green) and which are not (highlighted red).

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:


Mozilla Firefox
/tmp/cover100200339/
file:///tmp/cover100200339/coverage.html#file0
Search
alexedwards.net/snippetbox/cmd/web/handlers.go (37.7%)
not tracked no coverage low coverage * * * * * * * * high coverage

```
package main

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

        "alexedwards.net/snippetbox/pkg/forms"
        "alexedwards.net/snippetbox/pkg/models"
)

func (app *application) home(w http.ResponseWriter, r *http.Request) {
        s, err := app.snippets.Latest()
        if err != nil {
                app.serverError(w, err)
                return
        }

        app.render(w, r, "home.page.tmpl", &templateData{
                Snippets: s,
        })
}

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
        id, err := strconv.Atoi(r.URL.Query().Get(":id"))
        if err != nil || id < 1 {
                app.notFound(w)
                return
        }

        s, err := app.snippets.Get(id)
        if err == models.ErrNoRecord {
                app.notFound(w)
                return
        } else if err != nil {
                app.serverError(w, err)
                return
        }

        app.render(w, r, "show.page.tmpl", &templateData{
                Snippet: s,
        })
```

Note: If you're running some of your tests in parallel, you should use the `-covermode=atomic` flag instead to ensure an accurate count.

Chapter 14.

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 general way that our project 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, routing 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 about what each part of the codebase does, and how it fits in with the project as a whole.

I also recommend circling back to any parts of the book that you might have found difficult to understand 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.

Also, feel free to carry on playing around with the project and trying to extend it further. You might like to try:

- Adding a new 'About' page to the website.
- Increasing code coverage by adding more unit and integration tests.
- Creating an API endpoint which returns a JSON representation of a snippet.
- Adding functionality to confirm a user's email address on signup.
- Adding functionality for a user to reset their password.
- Creating a command line application under `cmd/cli` to carry out database admin tasks.

And finally, a small request. Sales and marketing really isn't my strong suit, so if you enjoyed this book and would recommend it to other people, I'd appreciate your help in spreading the word!

- Mention this book on Twitter
- Mention this book on Facebook

Chapter 15.

Appendices

Chapter 15.1.

How HTTPS Works

You can think of a TLS connection happening in two stages. The first stage is the *handshake*, in which the client verifies that the server is trusted and generates some *TLS session keys*. The second stage is the actual transmission of the data, in which the data is encrypted and signed using the TLS session keys generated during the handshake.

In summary, it works like this:

1. A TCP connection is established and the TLS handshake begins. The client sends the web server a list of the TLS versions and *cipher suites* that it supports (a cipher suite is essentially an identifier that describes the different cryptographic algorithms that the connection should use).

2. The web server sends the client confirmation of the TLS version and cipher suite it has chosen. From here on in, the precise process depends on which cipher suite is chosen. But for the sake of this explanation I'll assume that the cipher suite `TLS_RSA_WITH_AES_128_GCM_SHA256` is being used.

3. The first part of the cipher suite indicates the *key exchange algorithm* to be used (in this example `RSA`). The web server sends its TLS certificate (which contains its RSA public key). The client then verifies that the TLS certificate is not expired and is *trusted*. Web browsers

come installed with the public keys of all the major certificate authorities, which they can use to verify that the web server's certificate was indeed signed by the trusted certificate authority. The client also confirms that the host named in the TLS certificate is the one to which it has an open connection.

4. The client generates the secret *session keys*. It encrypts these using the server's RSA public key (from the TLS certificate) and sends them to the web server which decrypts them using their RSA private key. This is known as the *key exchange*. Both the client and web server now have access to the same session keys, and no other parties should know them. This is the end of the TLS handshake.

5. The transfer of the actual data is now ready to begin. The data is broken up into *records* (generally up to 16KB in size). The client calculates a HMAC of each record using one of the session keys and the *message authentication code* algorithm indicated by the cipher suite (in our example `SHA256`) and appends it to the record. The client then encrypts the record using another of the session keys and the *bulk encryption*algorithm indicated by the cipher suite (in our example `AES_128_GCM` — which is AES-128 in Galois/Counter mode).

6. The encrypted, signed record is sent to the server. Then, using the session keys, the server can unencrypt the record and verify that the signature is correct.

In this process two different types of encryption are used. The TLS handshake portion of the process uses asymmetric encryption (RSA) to securely share the session keys, whereas the actual transfer of the data uses symmetric encryption/decryption (AES) which is much faster.

You might also like to watch this excellent video by Johannes Bickel which illustrates the TLS handshake process visually and may help clarify how it works.

Forward Secrecy / ECDHE

A notable exception to the process above is where the chosen cipher suite uses Ephemeral Elliptic Curve Diffie-Hellman (*ECDHE*) key exchange during the handshake.

For instance, if the cipher suite `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` is chosen then ECDHE will be used. Notice the `ECDHE_RSA` at the start?

Using ECDHE in key exchange algorithm slightly changes steps 3 and 4 in the above summary. As well as sending the TLS certificate during the handshake, the server also dynamically generates an ECDHE key pair and sends the ECDHE pubic key to the client. It signs what it sends with its own RSA private key.

In step 4, the client then encrypts the session keys using the ECDHE public key, instead of the RSA public key contained in the web server's TLS certificate.

The advantage of using an ECDHE variant for the key exchange is that the secret session keys are encrypted using a different public key in every handshake, instead using the same one from the TLS certificate each time. It helps future-proof privacy in case someone is recording the traffic (e.g. government mass surveillance). Without it, anyone who gains access to the web server’s RSA private key in the future (e.g. by obtaining a warrant) would be able to decrypt the traffic.

By using dynamically generated ECDHE keys, breaking session keys can no longer be done by just obtaining the web server’s private key. It means that the session keys belonging to each individual connection must be compromised separately.

Generally, it’s good practice to prefer ECDHE cipher suite variants unless you have an explicit need for another device to read your traffic (e.g. a network monitor).

Chapter 15.2.

Further Reading and Useful Links

Coding Style Guidelines

- Effective Go — Tips for writing clear, idiomatic Go code.
- What's in a name? — Guidelines for naming things in Go.
- Go Code Review Comments — Style guidelines plus common mistakes to avoid.

Recommended Tutorials

- How to write benchmarks in Go — By Dave Cheney.
- Learning Go's Concurrency Through Illustrations — By Trevor Forrey.
- A Step-by-Step Guide to Go Internationalization & Localization — By Theo Despoudis.
- Generating Secure Random Numbers — By Matt Silverlock.
- Go 1.11 Modules — Various authors

Third-party Package Lists

- Go Projects — Large listing of third-party Go packages.

- Awesome Go — Another large listing of popular third-party Go packages.