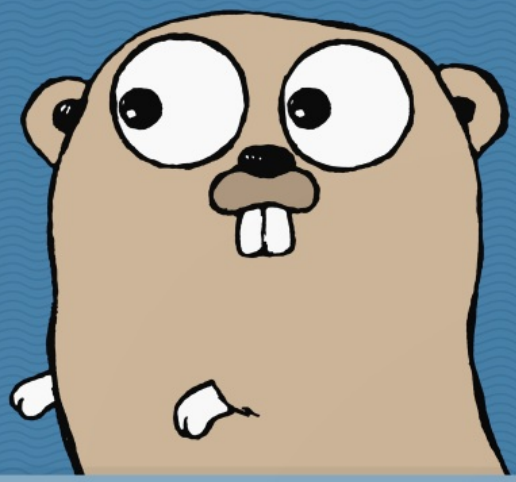
Alex Edwards

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 and securing your server.

Building a complete web application in this way has a number of 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-quality web applications with Go.

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

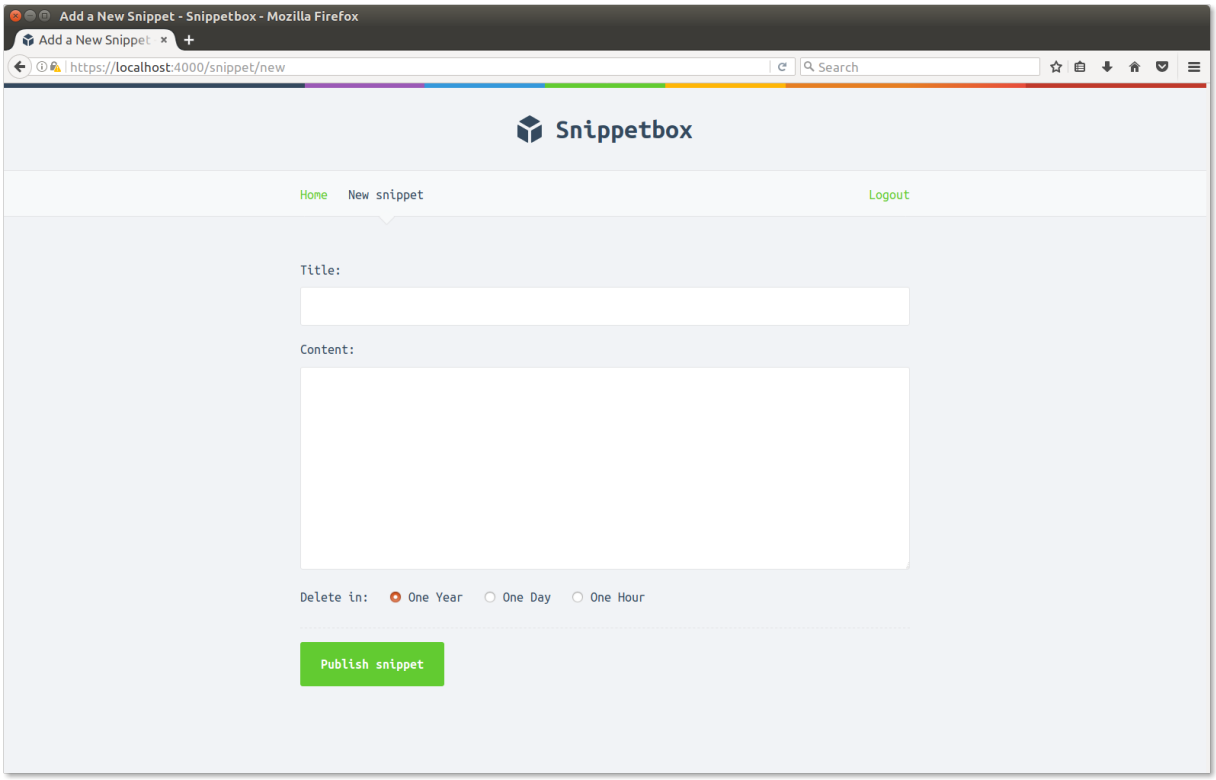
Break out your text editor, and happy coding!

— Alex

Contents

1. Introduction

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 their own text snippets and view other people's 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 save text 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 the Go 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 working at the 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 1.9 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

The code blocks throughout the book 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 of the code blocks also have a title bar at the top indicating the name of the file that we're working on.

hello.go

```
package main

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

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

In contrast, 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 a *notes* section, which contains 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 notes and circle back to them later.

About the Author

Hey, I'm Alex Edwards, a full-stack web developer. I began working with Go 5 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 work on a number of 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

Published by Brandberg Ltd.

2. Foundations

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

In particular you'll learn how to:

- Start a web server and listen for incoming requests.
- Send different responses and status codes to users.
- Structure your project in a sensible and scalable way.
- Route requests to different handlers based on the request path.
- Fetch and validate untrusted user input from URL query string parameters.
- Serve static files like images, CSS and Javascript from your application.

2.1. Workspaces and Repositories

Before writing any code we should setup a **workspace** that follows the Go conventions.

If you're not familiar with the term, a workspace is essentially a single directory on your computer where you want all your Go-related items to live. It's normal to use `$HOME/go` for your workspace, so if this directory doesn't already exist on your computer go ahead and create it now:

```
$ mkdir $HOME/go
```

By convention your workspace should contain three sub-directories: `bin`, `pkg` and `src`. Add them in if they don't already exist:

```
$ mkdir $HOME/go/bin $HOME/go/pkg $HOME/go/src
```

For now the most important of these is the `src` directory. This will contain the source-code repositories for your Go projects (and any third-party packages that you install).

Project Repository

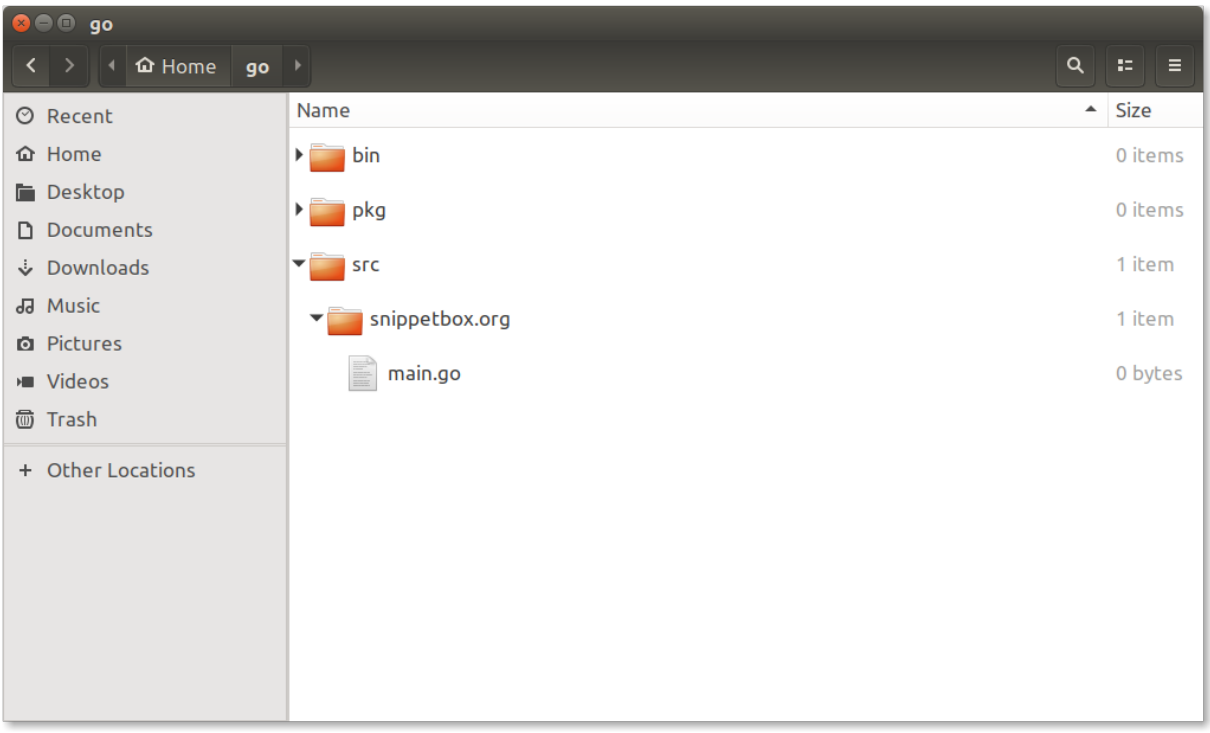
Next let's create a **repository** for our Snippetbox project under `src`. This will be the top-level home for our project – it will hold all the code we write as well as ancillary assets like static UI files and HTML templates.

It's important to note that the name of a repository determines its **import path**, so it's a good idea to pick a name that's globally unique to avoid import conflicts. An common convention is to base it on a URL that you own.

If you're following along, create a `snippetbox.org` repository and add a `main.go` file (which we'll use in the next chapter).

```
$ mkdir $HOME/go/src/snippetbox.org
$ touch $HOME/go/src/snippetbox.org/main.go
```

At this point the layout of your workspace should look a bit like this:



Notes

Using a Custom Workspace Location

If you don't want to use `$HOME/go` as your workspace that's OK. But you need to take an extra step and set a `GOPATH` environment variable which lets the Go tools know where your workspace is. For instance, if your workspace is at `$HOME/code/go` you should append the following line to your `$HOME/.profile` shell startup script...

$HOME/.profile

```
...
export GOPATH=$HOME/code/go
```

... and then restart your computer or log back in for the environment variable to take effect.

2.2. A Basic Web Application

In this chapter we'll kick off our Snippetbox project and create a super-simple web application with one route, which sends a plain-text `"Hello from Snippetbox"` response to the user.

There are three main components to making a web application in Go:

- 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 **serve mux** in Go terminology). This stores a mapping between the URL patterns for your application and the corresponding handlers. Usually you have one serve mux for your application containing all of your routes.

- The last thing we need is a running **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 a third-party server like Nginx or Apache.

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

main.go

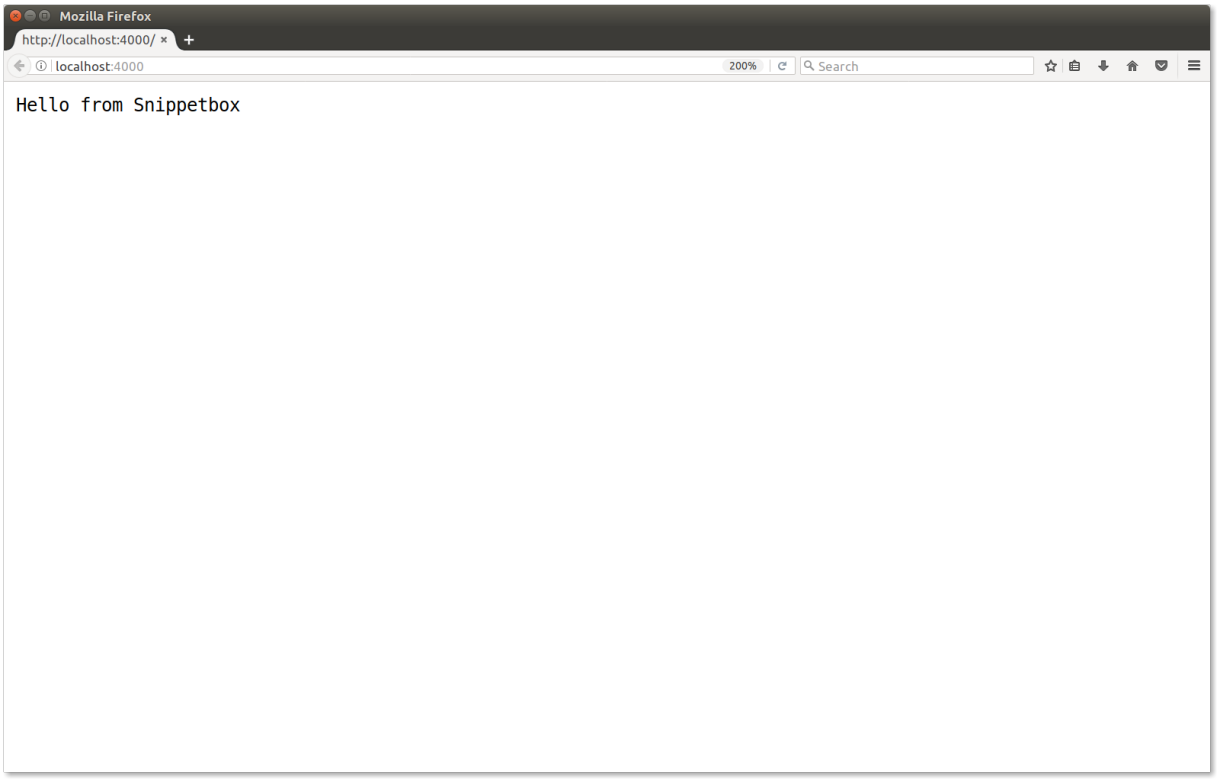
```
package main

import (
    "log"
    "net/http"
)

// Define a Home function which writes a plain-text "Hello from Snippetbox"
// message as the HTTP 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 serve mux. Then use
    // the mux.HandleFunc() method to register our 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 pass in
    // two parameters: the TCP network address to listen on (in this case ":4000")
    // and the serve mux we just created. If ListenAndServe() returns an error we
    // use the log.Fatal() function to record the error message and exit the program.
    log.Println("Starting server on :4000")
    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

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 request it will pass the request on to the serve mux and, in turn, the serve mux will check the URL path and dispatch the request to the matching handler – if one exists.

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

```
$ cd $HOME/go/src/snippetbox.org
$ go run main.go
2017/08/16 17:53:11 Starting server on :4000
```

While the application is running open up 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:



Before we continue, I should explain that Go's serve mux treats the pattern `"/"` like a catch-all. So when we run this application *all requests* will be dispatched to our `Home` handler. For instance, you can visit a different URL path like `http://localhost:4000/foo/bar` and you should receive exactly the same response. We'll talk more about this in the next chapter.

But before you stop the application, you might want to open a second terminal window and use curl to view the response headers:

```
$ curl -i http://localhost:4000/
HTTP/1.1 200 OK
Date: Wed, 16 Aug 2017 16:06:59 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8

Hello from Snippetbox
```

Notice how Go has defaulted to sending a `200 OK` status code, and has automatically set the correct `Date`, `Content-Length`, and `Content-Type` headers?

If you head back to your original terminal window, you can stop the web application by pressing `Ctrl+c` on your keyboard.

Notes

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 of 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.

Sometimes you might 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, and will return an error if a match can't be found.

2.3. Customizing Responses

There are a few different ways to make a handler in Go, but the most common is to write your handlers as a function – just like we have with the `Home` function in the previous chapter:

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

Notice how this takes exactly two mandatory parameters?

- The `http.ResponseWriter` parameter provides methods for assembling a HTTP response and sending it to the user. So far we've used its `w.Write()` method to send a byte slice containing `"Hello from Snippetbox"` as the response body.

- The `*http.Request` parameter holds information about the current request, such as the HTTP method and the URL being requested. I'll show how to use this as we progress through the book.

Restricting the Root Path

As I mentioned in the previous chapter, there's one problem with our application: All requests that our application receives are being sent the same response, regardless of their URL path.

Let's fix that and update our `Home` handler so it sends a `404 Not Found` response unless the URL path *exactly* matches `"/"`. To do this we'll need to use the `w.WriteHeader()` method to send the user a `404` HTTP status code, like so:

main.go

```
package main

import (
    "log"
    "net/http"
)

func Home(w http.ResponseWriter, r *http.Request) {
    // Use r.URL.Path to check whether the request URL path exactly matches "/".
    // If it doesn't, use the w.WriteHeader() method to send a 404 status code and
    // the w.Write() method to write a "Not Found" response body. Importantly, we
    // then return from the function so that the subsequent code is not executed.
    if r.URL.Path != "/" {
        w.WriteHeader(404)
        w.Write([]byte("Not Found"))
        return
    }

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

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", Home)

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

Although this change looks pretty straightforward there are a couple of `w.WriteHeader()` nuances I should explain:

- 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 you always need to call `w.WriteHeader()` *before* any call to `w.Write()`.

- It's only possible to call `w.WriteHeader()` once per request, and once the status code has been sent to the user it can't be changed. If you try to call it a second time then Go will log an error message.

Alright, save the file and restart the application. If you make a request to a URL like `http://localhost:4000/foo/bar` you should now get a `404 Not Found` response like so:

```
$ curl -i http://localhost:4000/foo/bar
HTTP/1.1 404 Not Found
Date: Wed, 16 Aug 2017 17:15:56 GMT
Content-Length: 9
Content-Type: text/plain; charset=utf-8

Not Found
```

In contrast the root path should continue to work just like before:

```
$ curl -i http://localhost:4000/
HTTP/1.1 200 OK
Date: Wed, 16 Aug 2017 17:15:58 GMT
Content-Length: 21
Content-Type: text/plain; charset=utf-8

Hello from Snippetbox
```

The NotFound Shortcut

We can shorten our `Home` function slightly by using the `http.NotFound()` helper function. This is essentially a lightweight wrapper around the `w.WriteHeader()` and `w.Write()` methods that we've just used.

main.go

```
package main

import (
    "log"
    "net/http"
)

func Home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        // Use the http.NotFound() function to send a 404 Not Found response.
        http.NotFound(w, r)
        return
    }

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

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", Home)

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

In terms of functionality this is almost exactly the same. The biggest difference is that we're now passing `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 actually quite rare to use the `w.Write()` and `w.WriteHeader()` methods directly like we have been so far. But I wanted to introduce them upfront because they underpin the more advanced (and interesting!) ways to send responses.

2.4. Project Structure

Before we add any more code to our `main.go` file it's a good time to think about the organization and structure for our project.

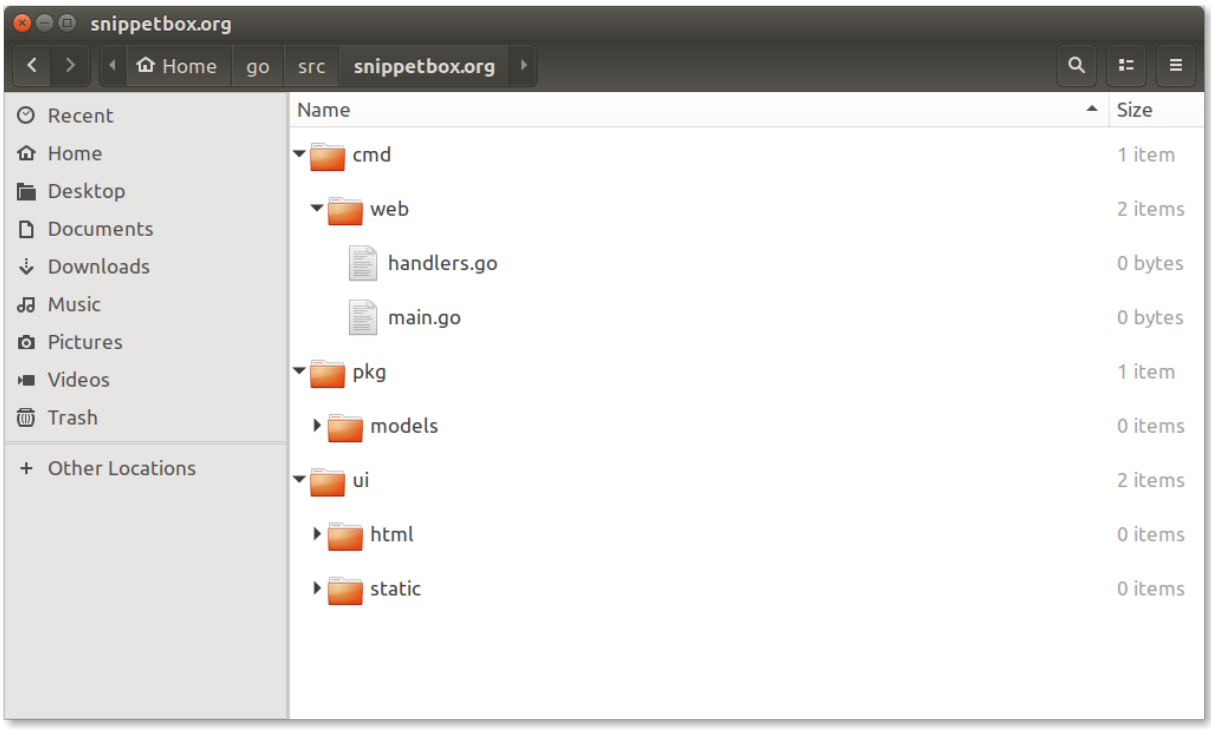
There's no single right (or even recommended) way to structure your project in Go. This can be both a blessing and a curse. It means that you have freedom and flexibility over how you organize your code, but you can also spend a lot of time trying to decide what the best structure should be.

For our project we'll use a popular and tried-and-tested approach which should be a good fit for a wide range of applications.

If you're following along, make sure that you're in the root of our project repository and run the following commands to generate an outline structure:

```
$ cd $HOME/go/src/snippetbox.org
$ rm main.go
$ mkdir -p cmd/web pkg/models ui/html ui/static
$ touch cmd/web/main.go
$ touch cmd/web/handlers.go
```

Once that's done your project repository should look just like this:



So what will each of these directories do?

- The `cmd` directory will contain the source code for our **executable applications**. For now we're only making one application – the Snippetbox web application – which will live in the `cmd/web` directory.

- The `pkg` directory will contain the reusable **packages** that our executable applications will import (a package is like a library in other languages). For now we've got one package directory, `models`, which will contain all

the database models for our project.

- There's also a `ui` directory in our repository root, which will contain our HTML templates and static files (like CSS and images).

A big benefit of this structure is that it scales nicely if your project grows to include multiple executable applications under the `cmd` directory. The `pkg` directory provides a sensible place to put shared, reusable, code (like database models) which can be easily imported by your different applications.

During our project build all the Go code we write will live exclusively under either the `cmd` or `pkg` directories. Sticking to this rule will help keep our Go and non-Go assets cleanly separated.

Porting the Code

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

cmd/web/main.go

```
package main

import (
    "log"
    "net/http"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", Home)

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

cmd/web/handlers.go

```
package main

import (
    "net/http"
)

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

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

To start our 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/*
2017/08/16 21:48:45 Starting server on :4000
```

As a side-note, it's important to emphasize that you can and should tailor your structure to fit to the project. The outline that we're using is a solid starting point, but as you gain experience with the language you'll get a feel for which patterns work well for you in different situations.

The only firm advice I can give is *try hard not to over-complicate things*; add structure and complexity only when it's demonstrably needed.

2.5. Routing Requests

Now let's concentrate on adding a couple of new routes to our Snippetbox web application, so that it looks like this:

URL Path	Handler	Action
/	Home	Display a "Hello from Snippetbox" message
/snippet	ShowSnippet	Display a specific snippet
/snippet/new	NewSnippet	Display the new snippet form

First, open up the `handlers.go` file and add a pair of placeholder functions for `ShowSnippet` and `NewSnippet`:

cmd/web/handlers.go

```
package main

import (
    "net/http"
)

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

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

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

// Add a placeholder NewSnippet handler function.
func NewSnippet(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Display the new snippet form..."))
}
```

And then register them with our serve mux, in exactly the same way as we did with the first route:

cmd/web/main.go

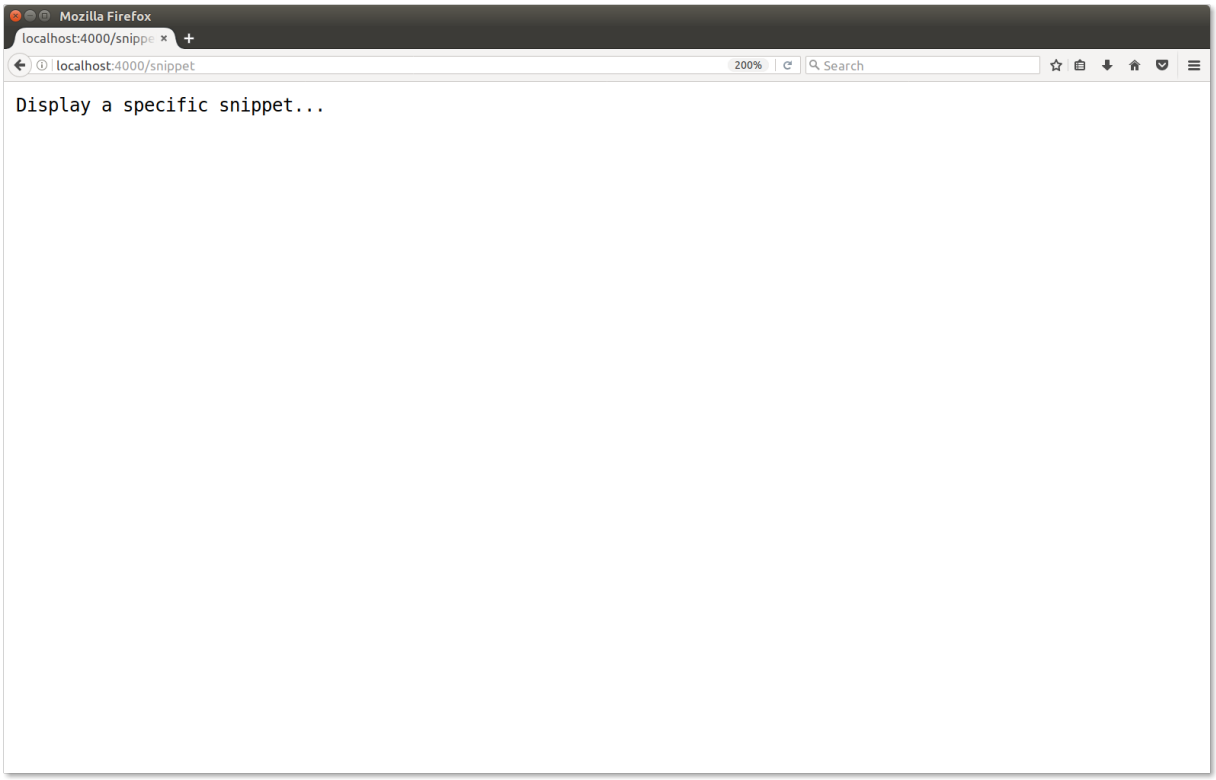
```
package main

import (
    "log"
    "net/http"
)

func main() {
    // Register the two new routes with the serve mux.
    mux := http.NewServeMux()
    mux.HandleFunc("/", Home)
    mux.HandleFunc("/snippet", ShowSnippet)
    mux.HandleFunc("/snippet/new", NewSnippet)

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

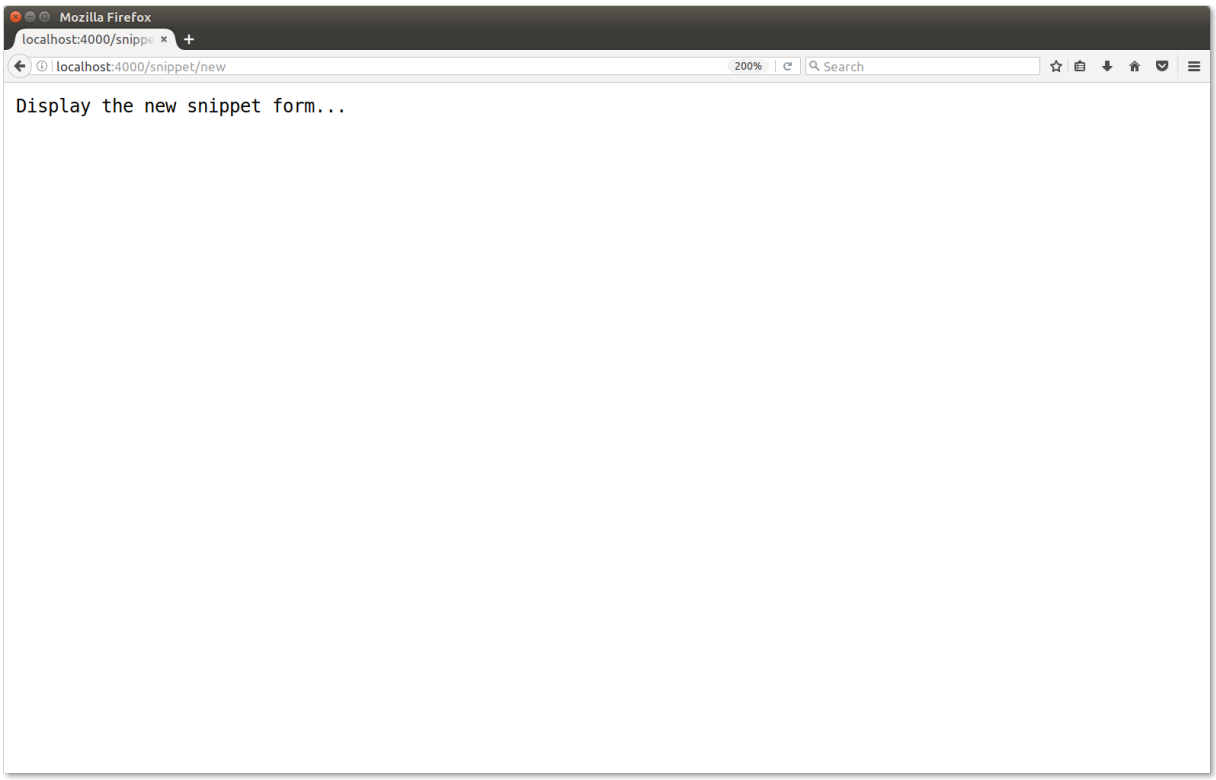
Make sure that both files are saved, and restart the web application:

```
$ $HOME/go/src/snippetbox.org
$ go run cmd/web/*
2017/08/17 18:32:04 Starting server on :4000
```

If you visit the following links in your browser you should now get the appropriate response for each route, looking a bit like the screenshots below:

- http://localhost:4000/snippet
- http://localhost:4000/snippet/new





Fixed Paths and Subtrees

Our two new routes – `"/snippet"` and `"/snippet/new"` – are both examples of **fixed paths** because they don't end in a trailing slash. In Go's serve mux, fixed path patterns like these are only matched (and the corresponding handler called) when the request URL path *exactly* matches the fixed path.

Go's serve mux also supports **subtree paths**, which do end with a trailing slash. An example of a subtree path is 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 `"/static/**"`.

This helps explain why our `"/"` pattern was acting like a catch-all. The pattern is a subtree path (because it ends in a trailing slash) so it essentially means *match a single slash, followed by anything (or nothing at all)*.

I should also mention that Go's serve mux gives longer patterns precedence over shorter ones. If a serve mux contains multiple patterns which match a request, it will always dispatch the request to the handler with the longest pattern.

The Default Serve Mux

As an aside, you might have read or watched other tutorials which use the `http.Handle()` and `http.HandleFunc()` functions to declare their routes.

Both these functions register routes with the **default serve mux**. There's nothing special about the default serve mux. It's just regular serve mux like we've already been using, 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()
```

I've avoided using this in our application because it poses a **security risk**.

Because the default serve mux is stored in a global variable, any package is able to 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 the default serve mux to expose a malicious handler to the web.

So as a rule of thumb it's a good idea to avoid the default serve mux. Use your own locally-scoped serve mux instead, like we have been so far.

Notes

Third-Party Routers

The functionality that Go's serve mux provides is pretty basic. It doesn't support routing based on the request method, it doesn't support regexp-based patterns, and it doesn't support semantic URLs with variables in them. If you're coming from a background of using frameworks like Rails, Django or Laravel you might find this a bit restrictive... and surprising!

But the reality is that Go's serve mux can still get you quite far, and for many applications it's perfectly sufficient. For the times that you need more, there's a huge choice of third-party routers that you can use. We'll take a look at these later in the book.

Domain Matching

It's possible to include host names in your route patterns. For example, the route below will only match if the request host and path is exactly `www.example.org/foo`:

```
mux := http.NewServeMux()
mux.HandleFunc("www.example.org/foo", FooHandler)
```

2.6. URL Query Strings

Let's add a bit of dynamic behavior to our application, so that the `"/snippet"` route accepts a query string parameter like so:

URL Path	Handler	Action
/	Home	Display a "Hello from Snippetbox" message
/snippet?id=1	ShowSnippet	Display a specific snippet
/snippet/new	NewSnippet	Display the new snippet form

Ultimately, the `ShowSnippet` function will use the `id` parameter to determine exactly which text snippet to show the user.

But for now, we'll just read the value of the `id` parameter and interpolate it with our placeholder response. To make this work we need our `ShowSnippet` handler function to do two things:

1. First 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.

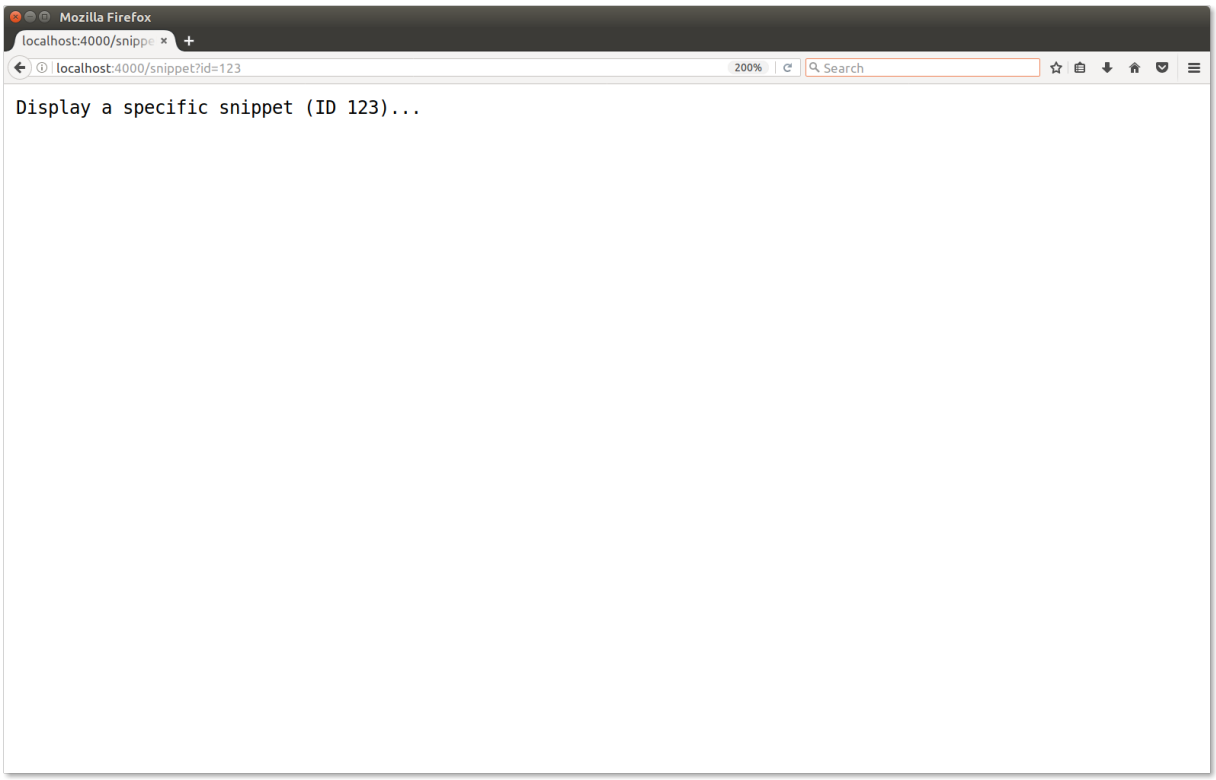
2. Because the `id` parameter is untrusted user input, we must validate the value to make sure it is sane and sensible. In this case we want to check that it contains a natural number.

cmd/web/handlers.go

```
package main

import (
    "fmt" // New import
    "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 couldn'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 response
    // and write it to the http.ResponseWriter.
    fmt.Fprintf(w, "Display a specific snippet (ID %d)...", id)
}

...
```

I should mention that the `r.URL.Query().Get()` method always returns a *string* value for the parameter, or the empty string `""` if no matching parameter exists. This is why we are able to pass it directly to the `strconv.Atoi()` function.

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



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 receive a `404 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)
```

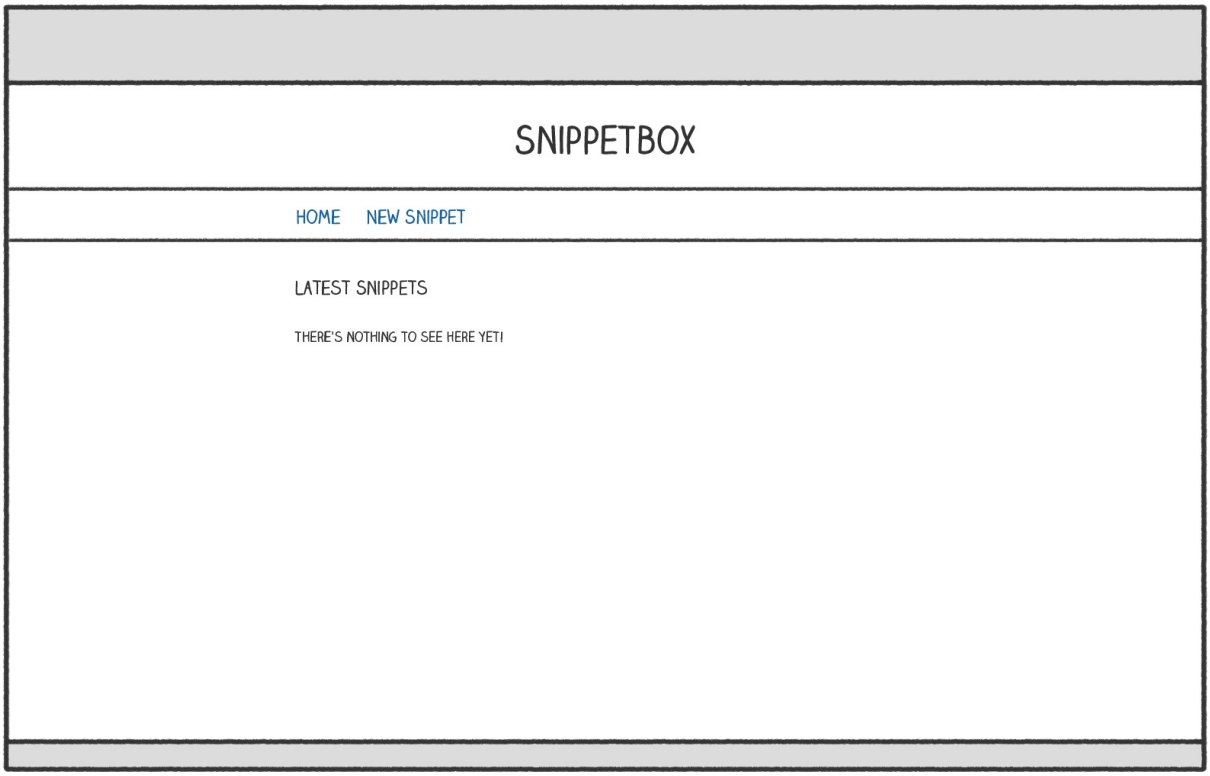
But we passed it our `http.ResponseWriter` object instead, and it worked fine.

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

If you're new to Go the concept of interfaces can be a bit confusing, but in practice, it simply means that anywhere you see an `io.Writer` parameter it's OK to use a `http.RespsonseWriter` too. Whatever is being written will subsequently be sent as the body of the HTTP response.

2.7. Basic HTML Templates

The next step in building our application is to make the `Home` function render a proper HTML homepage. Ultimately we're aiming to have it look a bit like this mockup:



To make this happen we'll use Go's `html/template` package, which provides a family of functions for safely parsing and rendering HTML templates.

Let's kick things off by adding two new files to the `ui/html` directory we made earlier:

- A `base.html` file with the outline HTML template for all our web pages .You might think of this as the 'layout' or 'master' template, if you're coming from other languages.
- A `home.page.html` file filled with our homepage-specific content.

```
$ cd $HOME/go/src/snippetbox.org
$ touch ui/html/base.html
$ touch ui/html/home.page.html
```

ui/html/base.html

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

ui/html/home.page.html

```
{{define "page-title"}}Home{{end}}

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

Hopefully this feels familiar if you've used templates in other languages before. The two files are simply regular HTML, with some extra **actions** in double curly braces.

We use the `{{define}}` and `{{end}}` actions to surround chunks of HTML and define distinct **named templates**. In our case we've defined three named templates: `base`, `page-title` and `page-body`. It's perfectly valid for a single file to contain multiple templates, like we have in the `home.page.html` file.

The `{{template}}` action is used to embed (or **nest**) one template in another. In our case the `base` template uses it to embed the `page-title` and `page-body` templates. If you're wondering, the dot at the end of the `{{template}}` action represents any dynamic data that you want to pass to the embedded template. We'll talk more about this later in the book.

Now for the exciting part. Let's update our `Home` function to combine these three templates into a **template set**, and then render the combined result as an HTTP response.

cmd/web/handlers.go

```
package main

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

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

    // Initialize a slice containing the paths to the two files.
    files := []string{
        "./ui/html/base.html",
        "./ui/html/home.page.html",
    }

    // 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 paths
    // as a variadic parameter). If there's an error, we log the detailed error
    // message and use the http.Error() function to send a generic 500 Internal
    // Server Error response.
    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }

    // Our template set contains three named templates: base, page-title and
    // page-body (note that every template in your template set must have a
    // unique name). We use the ExecuteTemplate() method to execute the "base"
    // template and write its content to our http.RespsonseWriter. The last
    // parameter to ExecuteTemplate() represents any dynamic data that we want to
    // pass in, which for now we'll leave as nil.
    err = ts.ExecuteTemplate(w, "base", nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
}

...
```

There's a couple of things to discuss here.

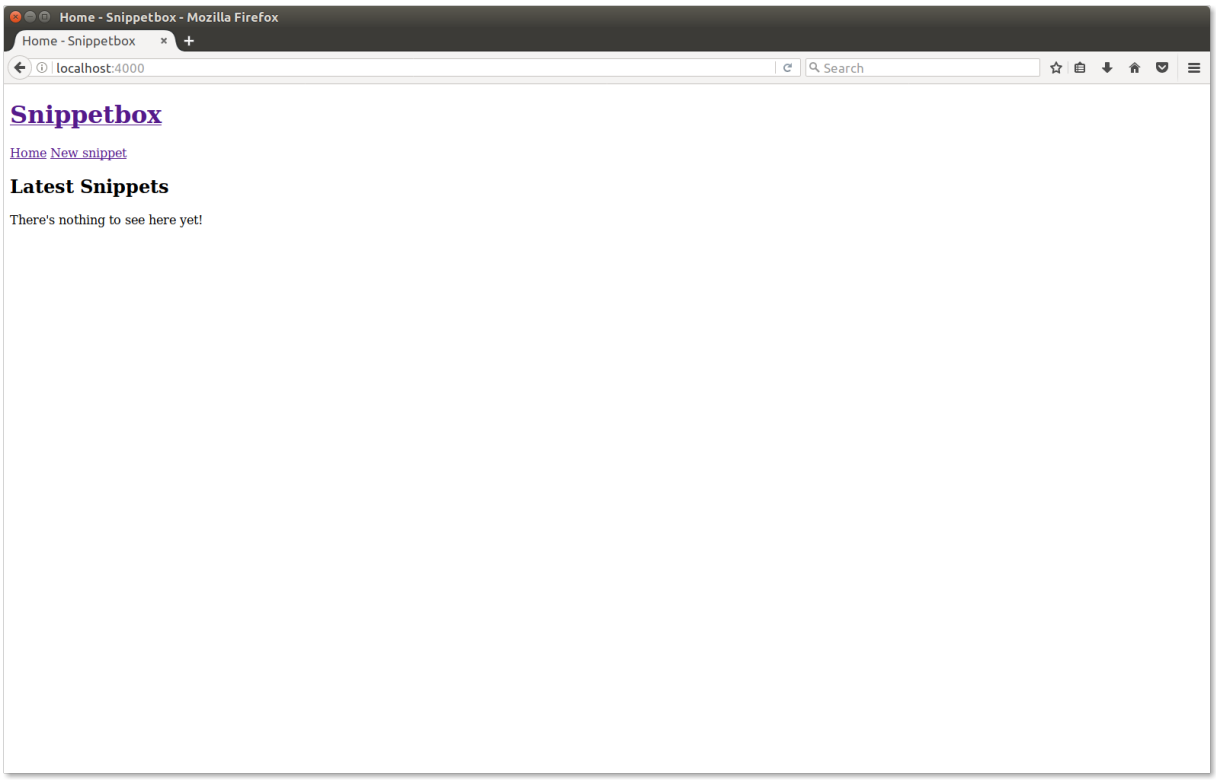
- We've started using the `http.Error()` function to send a response to the user if something goes wrong when rendering the templates. The `http.Error()` function is a shortcut – a bit like `http.NotFound()` – which sends a response to the user with a specific status code and plain-text message.

- The file paths that you pass to `template.ParseFiles()` must be relative to your current working directory, or absolute paths. In the code above I've made them relative to the root of the project repository.

So with that said, make sure you're in the root of our project repository and start the application:

```
$ cd $HOME/go/src/snippetbox.org
$ go run cmd/web/*
2017/08/18 18:02:08 Starting server on :4000
```

Open up `http://localhost:4000/` in your web browser and you should see that the HTML homepage is starting to take shape nicely.



Notes

The Block Action

Go's `html/template` package also provides a `{{block}}...{{end}}` action for embedding templates in each other. This is similar to the `{{template}}` action, except you can specify some default content if the template being embedded *doesn't exist in the current template set*.

In the context of a web application, this is particularly handy when you want to provide some default content (like a sidebar of footer) that 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}}
```

You can also, if you wish, not specify any default content between the `{{block}}` and `{{end}}` actions. In that case, if the embedded template doesn't exist in the template set then nothing will be rendered at all.

Removing Whitespace

If you *view source* of the homepage in your browser, you'll see that there is whitespace where the action tags are. In most cases this won't cause any problems and you shouldn't worry about it. But if it's important to you, you can remove the whitespace by pre/post-fixing your actions with the hyphen character.

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

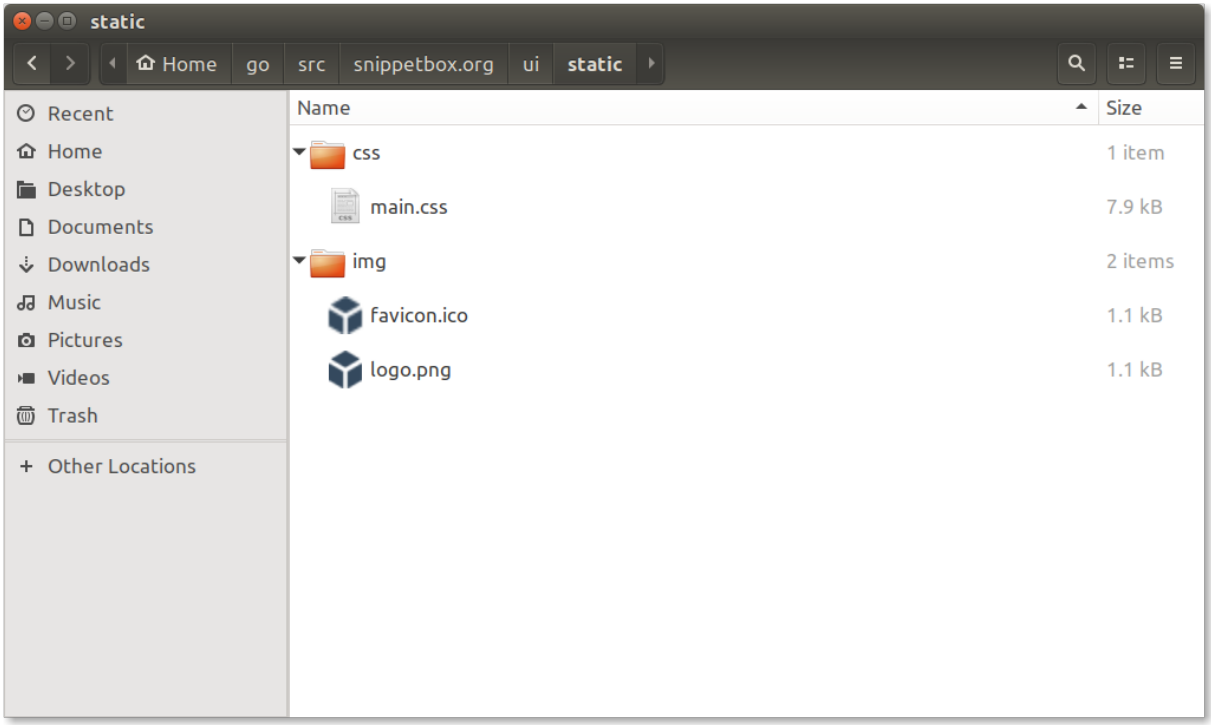
2.8. Serving Static Files

In this chapter we'll improve the look and feel of our homepage by adding to some static CSS and image files to our application.

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

```
$ cd $HOME/go/src/snippetbox.org
$ curl https://www.alexedwards.net/static/sb.v100.tar.gz | tar -xvz -C ./ui/static/
```

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



The http.FileServer Handler

The key to serving these static files from our web application is the `http.FileServer()` function. This lets us create a `http.FileServer` handler which serves files from a specific directory, like so:

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

Let's add a new route so that all requests which begin with `"/static/"` are handled by the file server (note how `"/static/"` is a subtree path pattern):

URL Path	Handler	Action
/	Home	Display a "Hello from Snippetbox" message
/snippet?id=1	ShowSnippet	Display a specific snippet

URL Path	Handler	Action
/snippet/new	NewSnippet	Display the new snippet form
/static/...	http.FileServer	Serve a specific static file

When the `http.FileServer` 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.

So for this to work correctly we have to 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 Not Found` response. Fortunately Go includes a `http.StripPrefix()` helper specifically for the task.

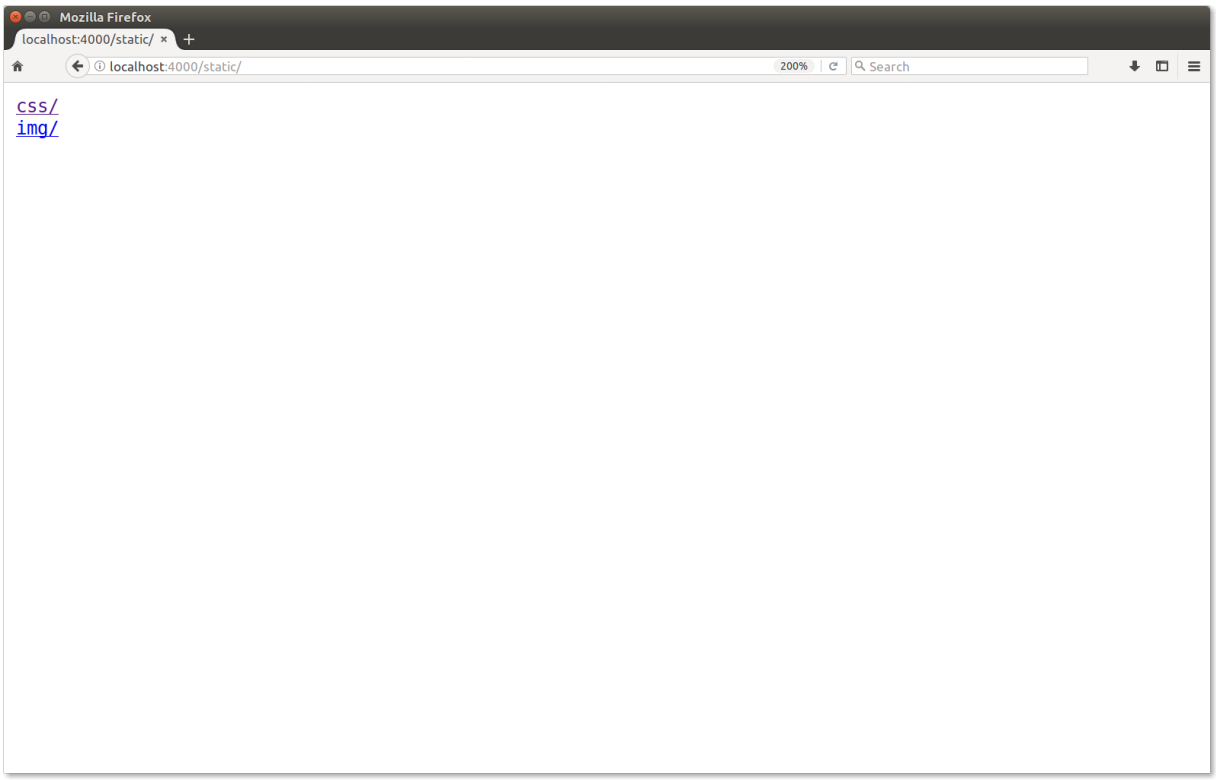
Open up your `main.go` file and add the following code, so that the file ends up looking like this:

cmd/web/main.go

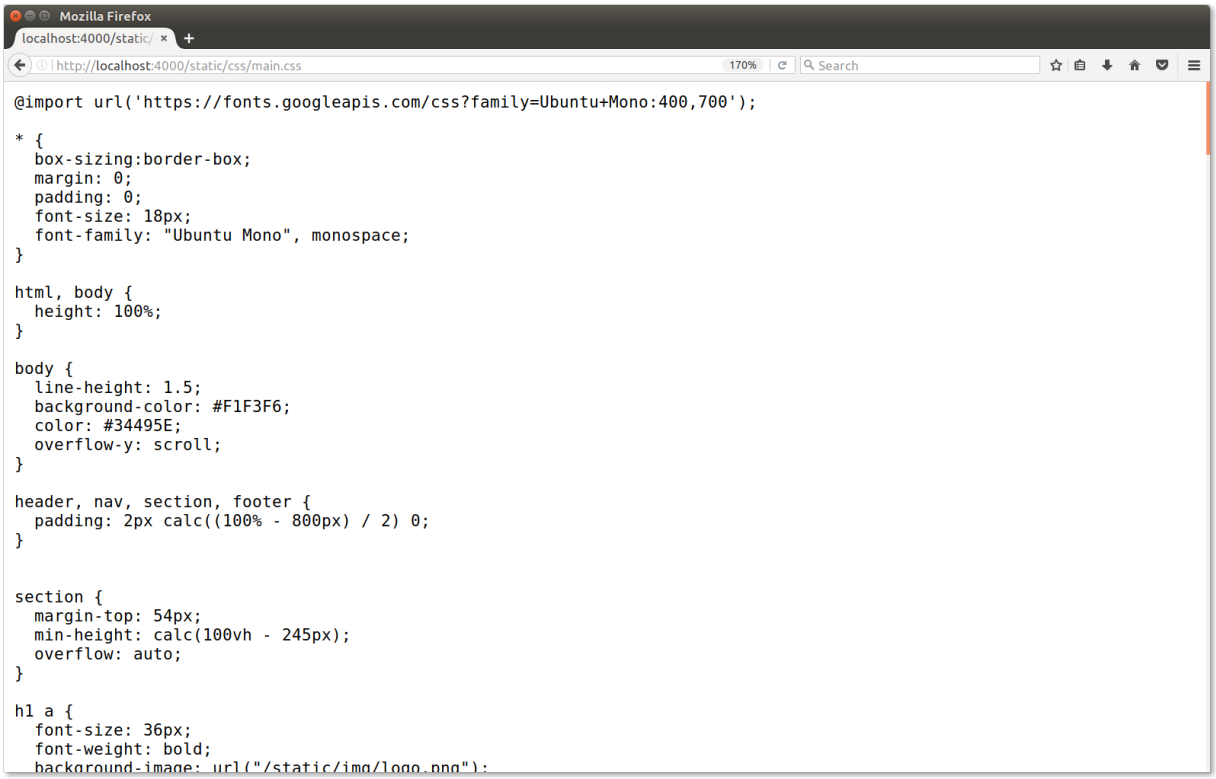
```
package main

import (
    "log"
    "net/http"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", Home)
    mux.HandleFunc("/snippet", ShowSnippet)
    mux.HandleFunc("/snippet/new", NewSnippet)

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

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

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

Once that's complete, save the file and restart the application.

Try taking a look at `http://localhost:4000/static/` in your browser and you should see a navigable directory listing of the `./ui/static` folder, a bit like this:



If you browse through, you should be able to view the individual files. For example, if you navigate to `http://localhost:4000/static/css/main.css` you should see the CSS file appear in your browser, like so:



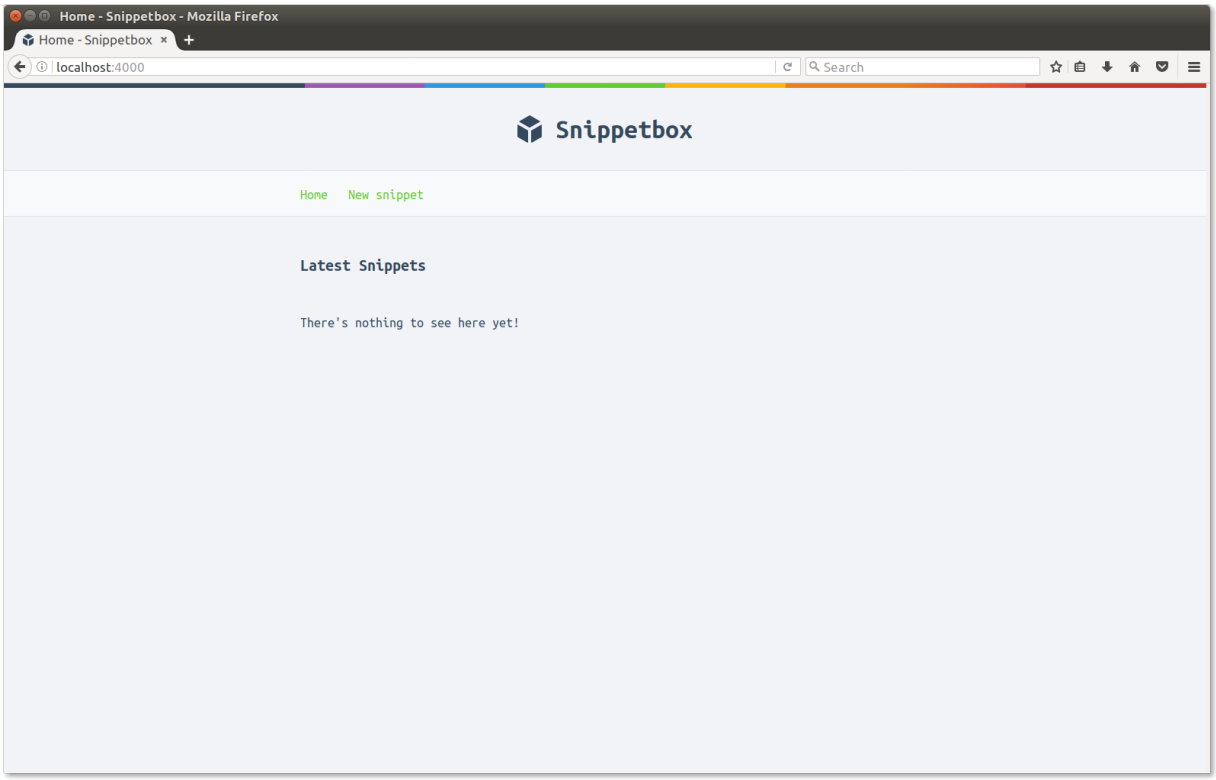
Using the Static Files

With that done, we can now update our `ui/html/base.html` file to make use of these static files:

ui/html/base.html

```
{{define "base"}}
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>{{template "page-title" .}} - Snippetbox</title>
        <!-- Link to the CSS stylesheet and favicon -->
        <link rel="stylesheet" href="/static/css/main.css">
        <link rel="shortcut icon" href="/static/img/favicon.ico" type="image/x-icon">
    </head>
    <body>
        <header>
            <h1><a href="/">Snippetbox</a></h1>
        </header>
        <nav>
            <a href="/">Home</a>
            <a href="/snippet/new">New snippet</a>
        </nav>
        <section>
            {{template "page-body" .}}
        </section>
    </body>
</html>
{{end}}
```

Make sure you save all your changes then visit `http://localhost:4000/`. If everything has worked correctly you should see a homepage which looks like this:



Notes

Features and Functions

Go's `http.FileServer` has a few really nice features that are worthy of a mention:

- 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.

- 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 our `logo.png` file, like so:

```
$ curl -i -H "Range: bytes=100-199" http://localhost:4000/static/img/logo.png
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Content-Length: 100
Content-Range: bytes 100-199/1075
Content-Type: image/png
Last-Modified: Thu, 04 May 2017 13:07:52 GMT
Date: Sat, 19 Aug 2017 09:12:49 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 Go 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:

```
func FooHandler(w http.ResponseWriter, r *http.Request) {
    file := filepath.Join("./ui/static", "foo.zip")
    http.ServeFile(w, r, file)
}
```

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

2.9. 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 it's 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 homepage"))
}
```

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.

We could then register this with our serve mux 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). For example:

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

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 to satisfy the `http.Handler` interface.

But so far throughout our application build we've been using the `mux.HandleFunc()` method to register our handler functions with the serve mux. 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)
```

The Request Cycle

The eagle-eyed of you might have noticed something interesting. The `http.ListenAndServe()` function takes a `http.Handler` object as the second parameter but we've been passing in our serve mux.

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

We were able to do this because the serve mux also has a `ServeHTTP()` method, meaning that it too satisfies the `http.Handler` interface.

For me it simplifies things to think of the serve mux 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 actually a very common idiom in Go, and something that we'll do a lot of later in our application build.

In fact, what exactly is happening is this: When our server receives a new HTTP request, it calls our serve mux's `ServeHTTP()` method. This looks up the relevant handler based on the request URL path, and in turn calls that handler's `ServeHTTP()` method.

Basically, 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 thing 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.

3. Configuration and Error Handling

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

In particular you'll learn how to:

- Set configuration settings for your application at runtime in an easy and idiomatic way using command-line flags.
- 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.
- Add helpers to your web application, so that your concerns are kept separate and you don't need to repeat yourself when writing code.

3.1. Command-line Flags

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 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" -static-dir="/var/www/static"
```

Defining Flags

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 configuration settings in favor of command-line flags instead:

cmd/web/main.go

```
package main

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

func main() {
    // Define command-line flags for the network address and location of the static
    // files directory.
    addr := flag.String("addr", ":4000", "HTTP network address")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")

    // Importantly, we use the flag.Parse() function to parse the command-line flags.
    // This reads in the command-line flag values and assigns them to the addr and
    // staticDir variables. You need to parse the flags *before* you use the addr
    // or staticDir variables, otherwise they will always contain the default value.
    // If any errors are encountered during parsing the application will be
    // terminated.
    flag.Parse()

    mux := http.NewServeMux()
    mux.HandleFunc("/", Home)
    mux.HandleFunc("/snippet", ShowSnippet)
    mux.HandleFunc("/snippet/new", NewSnippet)

    // The value returned from the flag.String() function is a pointer to the flag
    // value, not the value itself. So we need to dereference the pointer (i.e.
    // prefix it with the * symbol) before we use it as the path for our static file
    // server.
    fileServer := http.FileServer(http.Dir(*staticDir))
    mux.Handle("/static/", http.StripPrefix("/static", fileServer))

    // Again, we dereference the addr variable and use it as the network address
    // to listen on. Notice that we also use the log.Printf() function to interpolate
    // the correct address in the log message.
    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 listens on whatever address that you specify, like so:

```
$ go run cmd/web/* -addr=":9999"
2017/08/20 13:23:00 Starting server on :9999
```

Default Values

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

```
$ go run cmd/web/*
2017/08/20 13:23:46 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 our command-line flags. This has the benefit of converting whatever value the user provides at runtime to a `string` type.

Go also provides a range of other functions for defining command-line flags, 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 a different type.

For example, if you expect a specific command-line flag to contain an integer value it makes sense to define the flag using `flag.Int()` instead of `flag.String()`.

The Help Flag

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

```
$ go run cmd/web/* -help
Usage of /tmp/go-build554667279/command-line-arguments/_obj/exe/handlers:
  -addr string
        HTTP network address (default ":4000")
  -static-dir string
        Path to static assets (default "./ui/static")
exit status 2
```

So this is starting to look really good. We've now got an idiomatic way of managing the configuration settings at runtime, and an explicit and documented interface between our application and its operating configuration.

Using Environment Variables

If you've built and deployed web applications before you might be 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 when compared to command-line flags. We can't specify a default setting (the return value from `os.Getenv()` is the empty string if it doesn't exist), we don't get the `-help` functionality that we do with command-line flags, and the return value from `os.Getenv()` is *always* a string. We don't get automatic type conversions like we do with the `flag.Int()` and `flag.Bool()` 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
2017/08/20 14:53:46 Starting server on :9999
```

Notes

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 assets")
flag.Parse()
```

3.2. Dependency Injection

If you open up our `handlers.go` file you'll notice that the `Home` handler function still uses a hard-coded location for the HTML templates:

```
files := []string{
    "./ui/html/base.html",
    "./ui/html/home.page.html",
}
```

Now, we can easily add a new `-html-dir` command-line flag to our application so that we can configure it at runtime like our other settings. But this raises a good question: *how can we make the flag value available to our `Home` function from `main()`?*

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 loggers. 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 my advice is 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 code is in the same package, like ours, a neat way to inject dependencies is to put them into a custom `App` struct, and then define your handler functions as methods against `App`. I'll demonstrate.

Make sure you're in the root of our project repository and create a new `cmd/web/app.go` file containing the following code:

```
$ cd $HOME/go/src/snippetbox.org
$ touch cmd/web/app.go
```

cmd/web/app.go

```
package main

// Define an App struct to hold the application-wide dependencies and configuration
// settings for our web application. For now we'll only include a HTMLDir field
// for the path to the HTML templates directory, but we'll add more to it as our
// build progresses.
type App struct {
    HTMLDir string
}
```

Next update our handler functions so that they become *methods against the `App` struct*. Like so:

cmd/web/handlers.go

```
package main

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

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

    // Because the Home handler function is a now method against App it can access
    // its fields. So we can build the paths to the HTML template files using the
    // HTMLDir value in the App instance.
    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, "home.page.html"),
    }

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

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

// Change the signature of our ShowSnippet handler so it is defined as a method
// against App.
func (app *App) 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 (ID %d)...", id)
}

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

I should also quickly mention that in the code above we're using the `filepath.Join()` function to build the paths to the template files. This is better than manual concatenation because it automatically uses the correct file path separator for your OS (i.e. backslashes in Windows) and cleans the resulting path.

Let's now wire things up in our `main.go` file:

cmd/web/main.go

```
package main

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

func main() {
    // Define a new command-line flag for the path to the HTML template directory.
    addr := flag.String("addr", ":4000", "HTTP network address")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")

    flag.Parse()

    // Initialize a new instance of App containing the dependencies.
    app := &App{
        HTMLDir: *htmlDir,
    }

    // Swap our route declarations to use the App object's methods as the handler
    // functions.
    mux := http.NewServeMux()
    mux.HandleFunc("/", app.Home)
    mux.HandleFunc("/snippet", app.ShowSnippet)
    mux.HandleFunc("/snippet/new", app.NewSnippet)

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

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

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

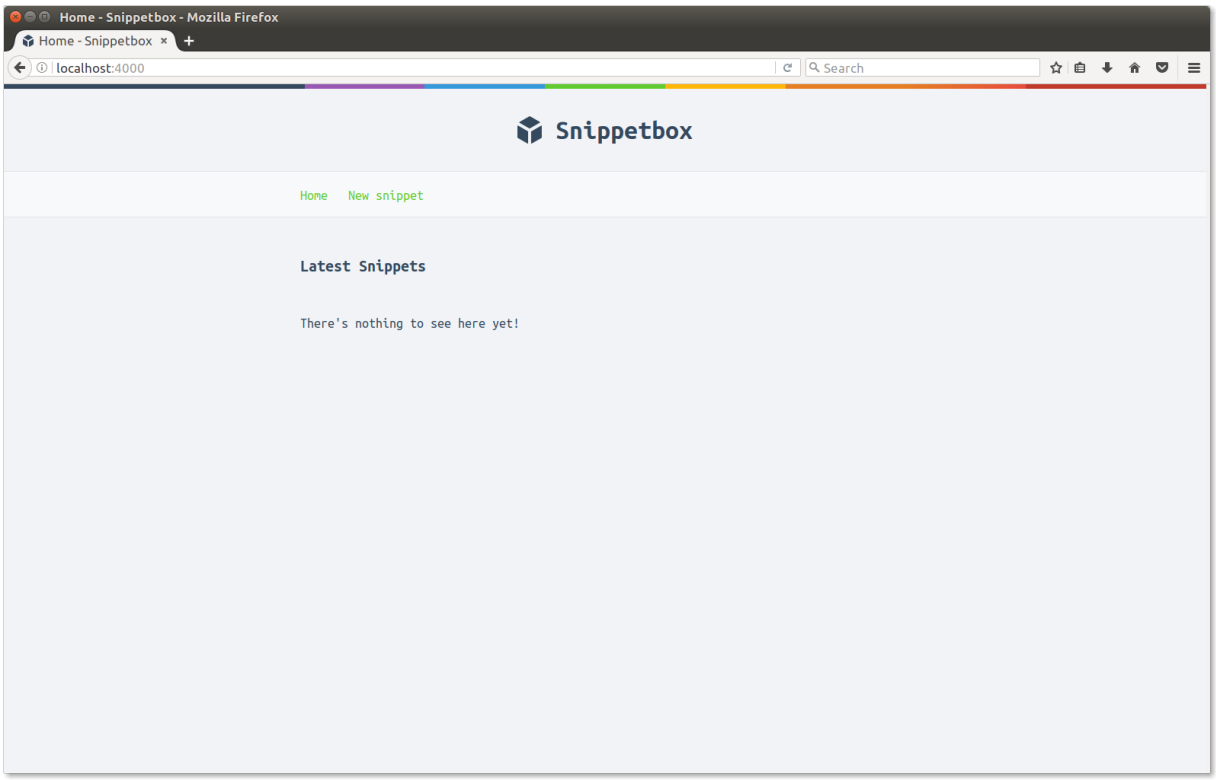
So where are we at?

Because we've removed all the hard-coded relative paths from our application, and swapped them for configurable command-line flags, our HTML template and static directories are now decoupled from our code. You should be able to correctly run the application from anywhere on your computer.

If you'd like to prove it to yourself, try changing into your root directory and then running the application using absolute file paths:

```
$ cd /
$ go run $HOME/go/src/snippetbox.org/cmd/web/* \
-html-dir="$HOME/go/src/snippetbox.org/ui/html" \
-static-dir="$HOME/go/src/snippetbox.org/ui/static"
2017/08/20 16:11:12 Starting server on :4000
```

If everything has worked correctly, you should still get the correctly rendered homepage when you visit `http://localhost:4000/` in your browser.



Notes

Using a Closure

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 `App` struct and have your handler functions close over this to form a **closure**. Very roughly:

```
func main() {
    app := &config.App{...}

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

```
func Home(app *config.App) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        if r.URL.Path != "/" {
            http.NotFound(w, r)
            return
        }

        files := []string{
            filepath.Join(app.HTMLDir, "base.html"),
            filepath.Join(app.HTMLDir, "home.page.html"),
        }

        ...
    }
}
```

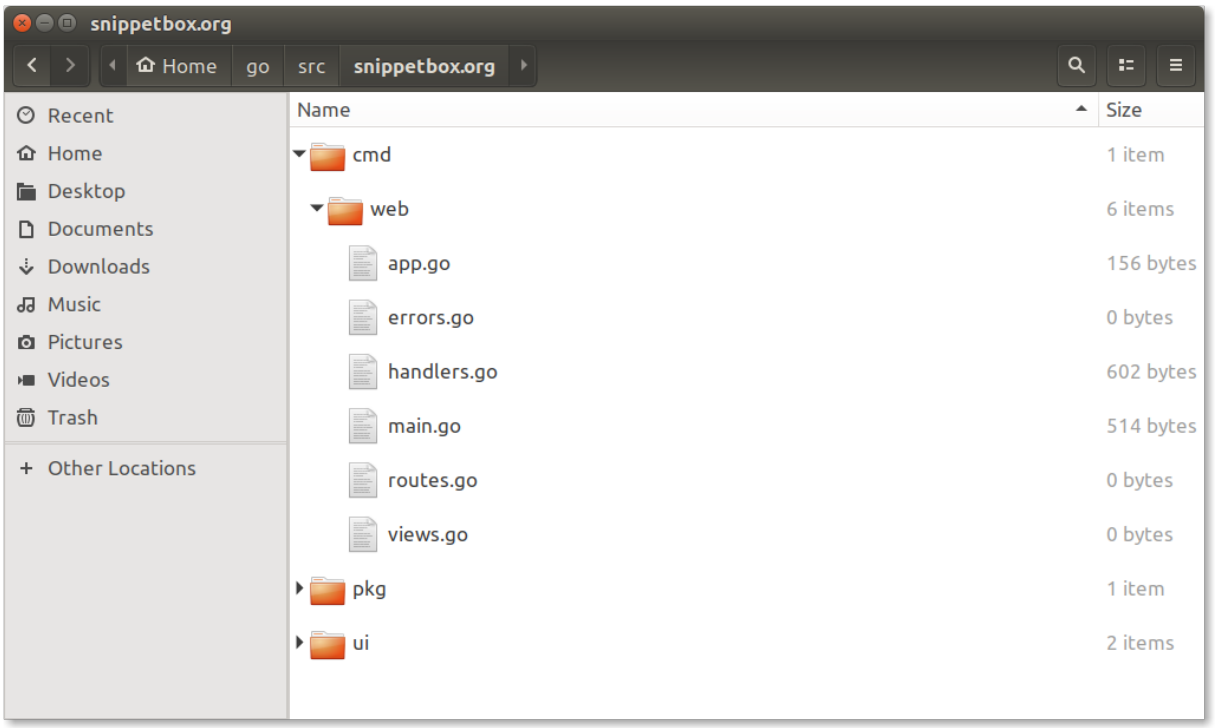
You can find a more complete and concrete example of how to use the closure pattern in this Gist.

3.3. Using Helpers

In this chapter we'll neaten up our application by moving some of the code into helper methods. This will help separate our concerns and stop us repeating code as we progress through the build.

Let's start by adding an `errors.go` file to provide some centralized error handling, a `routes.go` file to hold our application routes, and a `views.go` file to manage the rendering of HTML templates.

```
$ cd $HOME/go/src/snippetbox.org/
$ touch cmd/web/errors.go
$ touch cmd/web/routes.go
$ touch cmd/web/views.go
```



Centralized Error Handling

We'll begin with the error helpers. Open up the `errors.go` file and create the following three new methods:

cmd/web/errors.go

```
package main

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

// The ServerError helper writes an error message and stack trace to the log, then
// sends a generic 500 Internal Server Error response to the user.
func (app *App) ServerError(w http.ResponseWriter, err error) {
    log.Printf("%s\n%s", err.Error(), debug.Stack())
    http.Error(w, "Internal Server Error", http.StatusInternalServerError)
}

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

There's not much code here but it does introduce a few new features:

- In the `ServerError()` helper we use the `debug.Stack()` function to get stack trace for the *current goroutine* and append it to the log message. Being able to see the stack trace can be helpful when you're trying to debug errors.

- In the `ClientError()` helper we use the `http.StatusText()` function to generate a human-friendly text representation of a HTTP status code. For instance, `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. For example, in the `ServerError()` helper we've used the constant `http.StatusInternalServerError` instead of writing `500`. Using status constants is a nice touch which helps make your code clear and self-documenting – especially when dealing with more obscure statuses. You can find the complete list of status constants here.

You might also be wondering why the helpers are implemented as methods against `App`, instead of as regular functions. I've done this so the helpers will be able to access *exactly the same set of dependencies as our handler functions*, which will come in useful later in the build.

Centralized HTML Rendering

OK, let's move on to the `views.go` file and add a `RenderHTML()` helper which abstracts – and generalizes – our HTML template parsing code:

cmd/web/views.go

```
package main

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

func (app *App) RenderHTML(w http.ResponseWriter, page string) {
    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, page),
    }

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

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

The code here isn't introducing anything new – it's just a straight port of the code that we used to render the HTML templates earlier. The only difference is that the `RenderHTML()` function takes a `page` variable so we can dynamically specify the exact page we want to render.

Once that's done we can head back to the `handlers.go` file and update it to use the new helpers. In particular, the code for the `Home` method will now be much more compact.

cmd/web/handlers.go

```
package main

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

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

    app.RenderHTML(w, "home.page.html") // Use the app.RenderHTML() helper.
}

func (app *App) 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 app.NotFound() helper.
        return
    }

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

...
```

A Routes Method

And finally, let's move our route declarations into the `routes.go` file. This will help keep our `main()` function clean and focused as the application continues to grow.

There's a couple of steps to this. Moving the routes out of `main()` means that the file server will no longer have access to the location of the static file directory from the command-line flags. We can fix that by passing it as a

dependency just like we did with the HTML templates directory. Go ahead and add a `StaticDir` field to the `App` object:

cmd/web/app.go

```
package main

// Add a new StaticDir field to our application dependencies.
type App struct {
    HTMLDir   string
    StaticDir string
}
```

And then in the `routes.go` file create an `app.Routes()` method like so:

cmd/web/routes.go

```
package main

import (
    "net/http"
)

func (app *App) Routes() *http.ServeMux {
    // Declare a serve mux and define the routes in exactly the same as before.
    mux := http.NewServeMux()
    mux.HandleFunc("/", app.Home)
    mux.HandleFunc("/snippet", app.ShowSnippet)
    mux.HandleFunc("/snippet/new", app.NewSnippet)

    // Use the app.StaticDir field as the location of the static file directory.
    fileServer := http.FileServer(http.Dir(app.StaticDir))
    mux.Handle("/static/", http.StripPrefix("/static", fileServer))

    // Return the serve mux.
    return mux
}
```

Now we should be set to hook it all together in the `main.go` file:

cmd/web/main.go

```
package main

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

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")

    flag.Parse()

    // Add the *staticDir value to our application dependencies.
    app := &App{
        HTMLDir:   *htmlDir,
        StaticDir: *staticDir,
    }

    // Pass the app.Routes() method (which returns a serve mux) to the
    // http.ListenAndServe() function.
    log.Printf("Starting server on %s", *addr)
    err := http.ListenAndServe(*addr, app.Routes())
    log.Fatal(err)
}
```

4. Database-Driven Responses

For our Snippetbox web application to become truly useful we need somewhere to store (or *persist*) the text snippets, and the ability to query the 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).
- 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.
- Create a standalone `models` package, so that your database logic is reusable and decoupled from your web application.
- Use transactions, so that you can execute multiple SQL statements in one atomic action.

4.1. Setting Up MySQL

Installing 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.

When you are installing MySQL, remember to keep a mental note of the password you set for the `root` user; you'll need it in the next step.

Scaffolding the Database

Now MySQL is installed you should be able to connect to it from your terminal as the `root` user. When prompted type the password that you used during the installation.

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

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 make 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 our text snippets:

```
-- Create a `snippets` table.
CREATE TABLE snippets (
    id INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    created DATETIME NOT NULL,
    expires DATETIME NOT NULL
);

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

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

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

```
-- Add some dummy records (which we'll use in the next couple of chapters).
INSERT INTO snippets (title, content, created, expires) VALUES (
    'An old silent pond',
    'An old silent pond...\nA frog jumps into the pond,\nsplash! Silence again.\n\n– Matsuo Bashō',
    UTC_TIMESTAMP(),
    DATE_ADD(UTC_TIMESTAMP(), INTERVAL 1 YEAR)
);
```

```
INSERT INTO snippets (title, content, created, expires) VALUES (
    'Over the wintry forest',
    'Over the wintry\nforest, winds howl in  rage\nwith no leaves to blow.\n\n– Natsume Soseki',
    UTC_TIMESTAMP(),
    DATE_ADD(UTC_TIMESTAMP(), INTERVAL 1 YEAR)
);

INSERT INTO snippets (title, content, created, expires) VALUES (
    'First autumn morning',
    'First autumn morning\nthe mirror I stare into\nshows my father''s face.\n\n– Murakami Kijo',
    UTC_TIMESTAMP(),
    DATE_ADD(UTC_TIMESTAMP(), INTERVAL 1 MINUTE)
);
```

Notice that the first two text snippets expire in 1 year, the third expires in one minute?

Creating a User

From a security point of view it's not a good idea to connect to MySQL as the `root` user from our web application. Instead it's better to create a specific 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 `sb` user with `SELECT, INSERT, UPDATE, DELETE` privileges only on the database.

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

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

Connecting as the New User

You should now be able to re-connect to MySQL as the `sb` user. When prompted, enter the password that you just set.

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

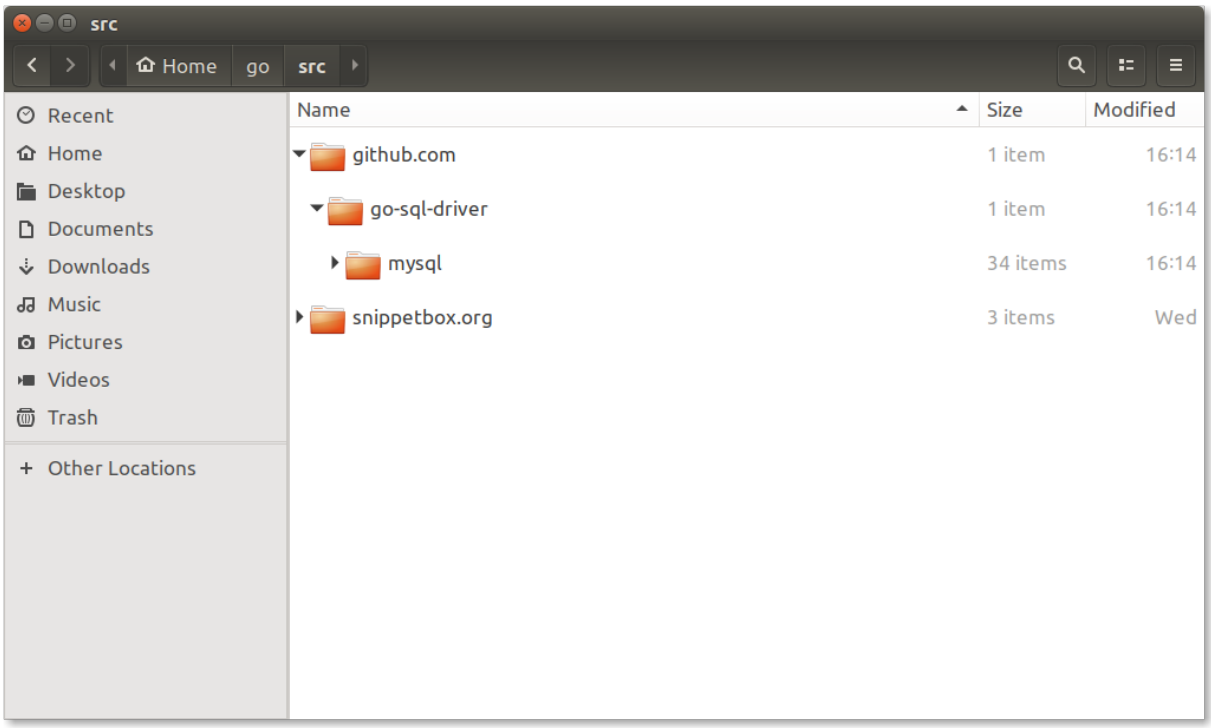
```
$ mysql -D snippetbox -u sb -p
Enter password:
mysql> DROP TABLE snippets;
ERROR 1142 (42000): DROP command denied to user 'sb'@'localhost' for table 'snippets'
mysql> SELECT * FROM snippets;
+----+-----------------------+-------------------------------------------------------------------------------------------+----------
| id | title                 | content                                                                                   | created
+----+-----------------------+-------------------------------------------------------------------------------------------+----------
|  1 | An old silent pond    | An old silent pond...
A frog jumps into the pond,
splash! Silence again.

– Matsuo Bashō    | 2017-08-21 13:38:01 | 2018-08-21 13:38:01 |
|  2 | Over the wintry forest | Over the wintry
forest, winds howl in  rage
with no leaves to blow.

– Natsume Soseki       | 2017-08-21 13:38:01 | 2018-08-21 13:38:01 |
|  3 | First autumn morning   | First autumn morning
the mirror I stare into
shows my father's face.

– Murakami Kijo       | 2017-08-21 13:38:01 | 2017-08-21 13:39:01 |
+----+-----------------------+-------------------------------------------------------------------------------------------+----------
3 rows in set (0.00 sec)
```

4.2. Installing a 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 actual database itself.

You can find a comprehensive list of available drivers on the Go wiki, but for our application we'll use the popular `github.com/go-sql-driver/mysql` driver. If you're following along you can download it using the `go get` command, which will add a copy of the package to your workspace's `src` directory:

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



4.3. Designing a Database Model

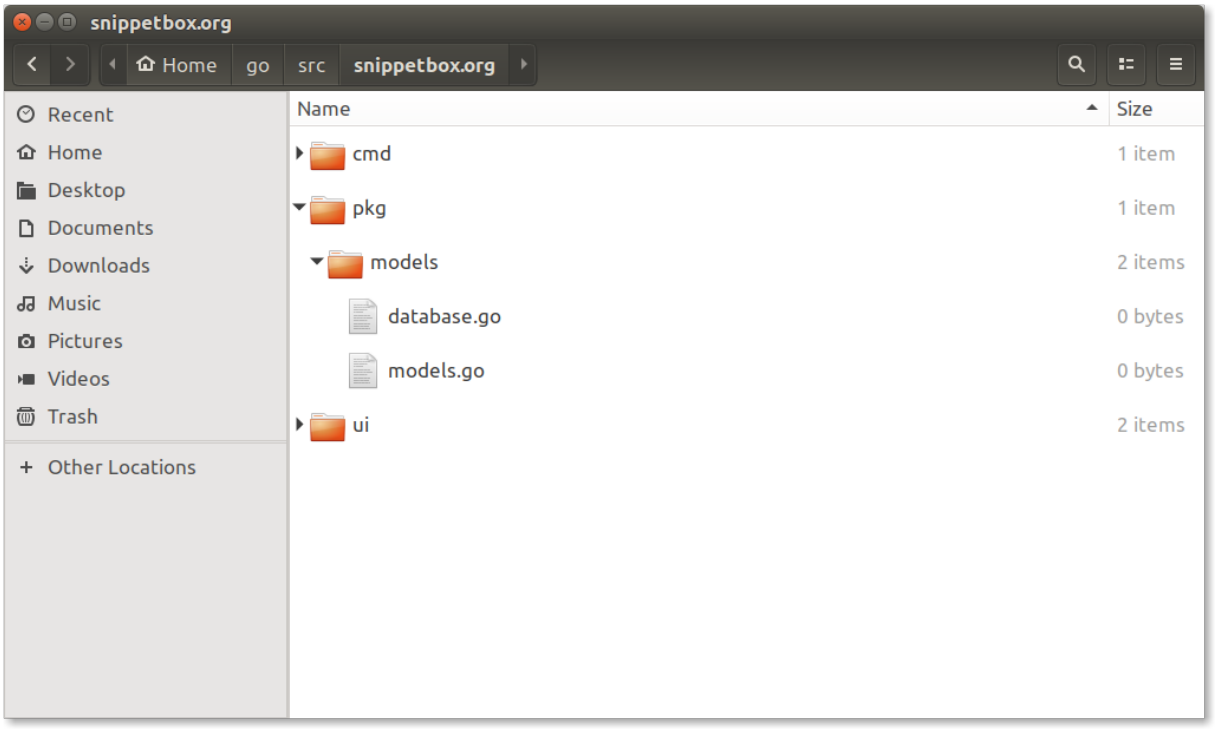
In this chapter we're going to sketch out a database model for our project.

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 distinct package separate 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 files in the `pkg/models` directory that we made at the start of this book:

```
$ cd $HOME/go/src/snippetbox.org
$ touch pkg/models/models.go
$ touch pkg/models/database.go
```



We'll use the `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:

pkg/models/models.go

```
package models

import (
    "time"
)

// Define a Snippet type to hold the information about an individual snippet.
type Snippet struct {
    ID      int
    Title   string
    Content string
    Created time.Time
    Expires time.Time
}

// For convenience we also define a Snippets type, which is a slice for holding
// multiple Snippet objects.
type Snippets []*Snippet
```

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

Now let's move on to the `database.go` file, which will contain the code specifically for working with our MySQL database. In this file we're going to define a new `Database` type and implement a `GetSnippet()` method on it. Like so:

pkg/models/database.go

```
package models

import (
    "time"
)

// Declare a Database type (for now it's just an empty struct).
type Database struct{}

// Implement a GetSnippet() method on the Database type. For now, this just returns
// some dummy data, but later we'll update it to query our MySQL database for a
// snippet with a specific ID. In particular, it returns a dummy snippet if the id
// passed to the method equals 123, or returns nil otherwise.
func (db *Database) GetSnippet(id int) (*Snippet, error) {
    if id == 123 {
        snippet := &Snippet{
            ID:      id,
            Title:   "Example title",
            Content: "Example content",
            Created: time.Now(),
            Expires: time.Now(),
        }
        return snippet, nil
    }

    return nil, nil
}
```

Using the Database Model

To use the `GetSnippet()` method in our handlers we need to establish a new `models.Database` object in `main()` and then inject it as a dependency via the `App` struct – just like we have with the other dependencies.

Open up your `main.go` file and update it like so:

cmd/web/main.go

```
package main

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

    "snippetbox.org/pkg/models" // New import
)

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")

    flag.Parse()

    // Initialize a models.Database instance and add it to our application
    // dependencies.
    app := &App{
        Database:  &models.Database{},
        HTMLDir:   *htmlDir,
        StaticDir: *staticDir,
    }

    log.Printf("Starting server on %s", *addr)
    err := http.ListenAndServe(*addr, app.Routes())
    log.Fatal(err)
}
```

We'll also need to update the `app.go` file and update it to hold a `*models.Database` object:

cmd/web/app.go

```
package main

import (
    "snippetbox.org/pkg/models" // New import
)

type App struct {
    Database  *models.Database
    HTMLDir   string
    StaticDir string
}
```

Now for the interesting part. Open up the `handlers.go` file and update the `ShowSnippet` function to use the `GetSnippet()` method from the database model:

cmd/web/handlers.go

```
package main

...

func (app *App) 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
    }

    snippet, err := app.Database.GetSnippet(id)
    if err != nil {
        app.ServerError(w, err)
        return
    }
    if snippet == nil {
        app.NotFound(w)
        return
    }

    fmt.Fprint(w, snippet)
}

...
```

Make sure to save all the files and restart the web application. If you visit `http://localhost:4000/snippet?id=123` in your browser you should see a response like this:

4.4. Creating a Connection Pool

Now that we've installed a driver the next step is to connect to MySQL from our web application.

At this point in the build our MySQL database has been setup, and we've got a driver installed. In this chapter we'll take the natural next step use the tools in Go's `database/sql` package to connect to MySQL from our web application .

The key to this is the `sql.Open()` function, which looks a bit like this:

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

There are a few things about `sql.Open()` that are worth explaining and emphasizing:

- 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*) which describes how to connect to your database.
- The format of the data source name (or DSN) 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 our driver you can find that documentation here. In the code above, the DSN is saying: *connect to the `snippetbox` database on the local machine as the `sb` user with the password `pass`*.
- The `sql.DB` object returned by `sql.Open()` isn't a database connection – it's a **pool of many connections**. 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's a waste of memory and network resources.

Usage in our Web Application

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

cmd/web/main.go

```
package main

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

    "snippetbox.org/pkg/models"

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

func main() {
    // Define a new command-line flag for the MySQL DSN string.
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "sb:pass@/snippetbox?parseTime=true", "MySQL DSN")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")

    flag.Parse()

    // To keep the main() function tidy I've put the code for creating a connection
    // pool into the separate connect() function below. We pass connect() the DSN
    // from the command-line flag.
    db := connect(*dsn)

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

    app := &App{
        Database:  &models.Database{},
        HTMLDir:   *htmlDir,
        StaticDir: *staticDir,
    }

    log.Printf("Starting server on %s", *addr)
    err := http.ListenAndServe(*addr, app.Routes())
    log.Fatal(err)
}

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

    if err := db.Ping(); err != nil {
        log.Fatal(err)
    }

    return db
}
```

There's a couple of 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 use the `db.Ping()` method to create a connection and check for any errors.

Testing a Connection

Make sure that all the files are 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 our normal *starting server...* message like so:

```
go run cmd/web/*
2017/08/21 17:36:04 Starting server on :4000
```

If the application fails to start and you get an `"Access denied..."` error message 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.

Passing the Connection Pool to our Models

Now that the connection pool is established we need to somehow make it available to our `models` package so that the `GetSnippet()` method can use it. A nice way to do this is to have our `model.Database` struct wrap (or in Go terminology, *anonymously embed*) a pointer to the `sql.DB` connection pool. I'll demonstrate:

pkg/models/database.go

```
package models

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

// Anonymously embed the sql.DB connection pool in our Database struct, so we can
// later access its methods from GetSnippet().
type Database struct {
    *sql.DB
}

func (db *Database) GetSnippet(id int) (*Snippet, error) {
    ...
}
```

cmd/web/main.go

```
package main

...

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "sb:pass@/snippetbox?parseTime=true", "MySQL DSN")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")

    flag.Parse()

    db := connect(*dsn)
    defer db.Close()

    app := &App{
        // Pass in the connection pool when initializing the models.Database object.
        Database:  &models.Database{db},
        HTMLDir:   *htmlDir,
        StaticDir: *staticDir,
    }

    log.Printf("Starting server on %s", *addr)
    err := http.ListenAndServe(*addr, app.Routes())
    log.Fatal(err)
}
```

Notes

The db.Close Method

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 `log.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'll be beneficial later in the future if you add a graceful shutdown to your application.

4.5. Single-record SQL Queries

In this chapter we'll update our `GetSnippet()` method to retrieve and return a single snippet from the database.

To do this we want to execute the following SQL query:

```
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). It also includes a check on the expiry time so that we don't return any text snippets that have expired.

We're also using the `?` character to indicate a **placeholder parameter** for the `id` value. Because the `id` value we want to query for is untrusted user input (it's originally coming from the request query string) it's good practice to use a placeholder parameter instead of interpolating it in the SQL query. We'll discuss placeholder parameters in more detail later in the chapter.

The final thing I'd like to mention is that we're specifying the column names we want to fetch explicitly, rather than using the `SELECT * FROM snippets` wildcard. This helps make our application more robust – if we add a new column to the `snippets` table in the future it won't suddenly break our Go code.

Executing the Query

Go provides three different methods for executing database queries. As a rule of thumb:

- `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`).

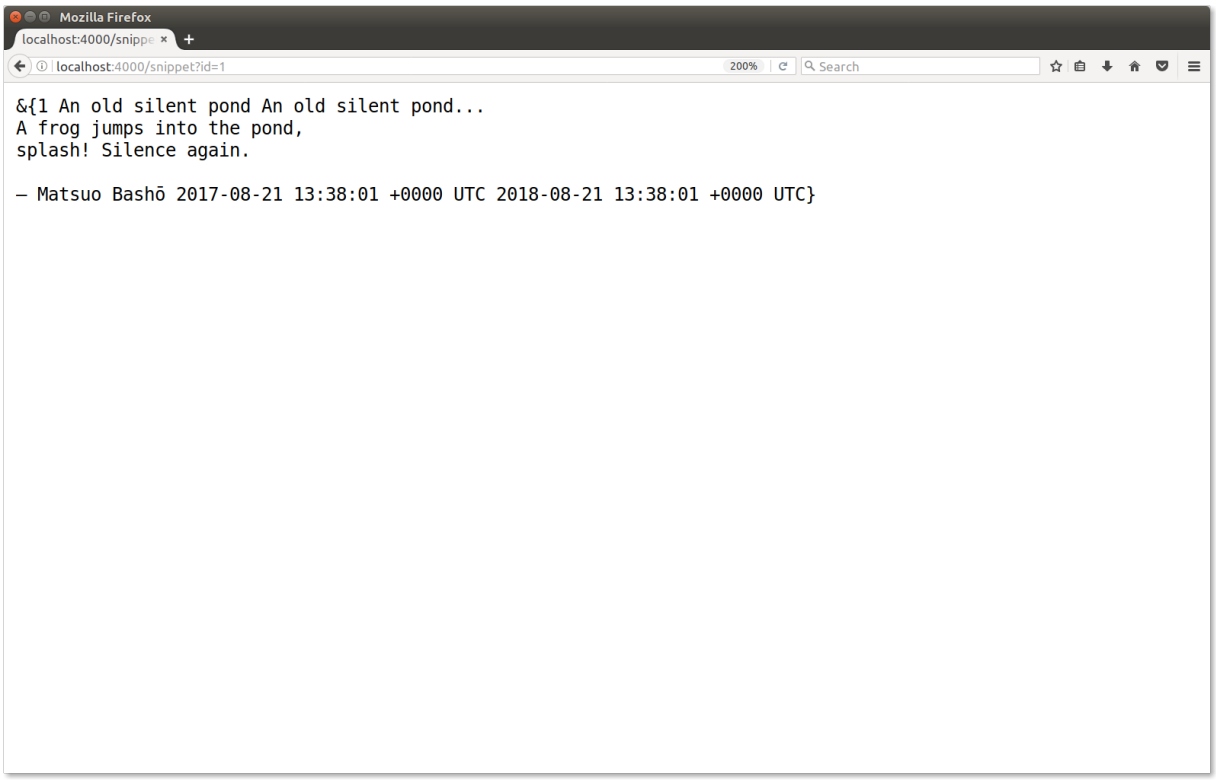
So in our case, the most appropriate tool for the job is `DB.QueryRow()`. Let's jump in the deep end and use this in our `GetSnippet()` method. We'll discuss the details afterwards.

If you're following along, open your `pkg/models/database.go` file and update it like so:

pkg/models/database.go

```
package models

import (
    "database/sql"
)

type Database struct {
    *sql.DB
}

func (db *Database) GetSnippet(id int) (*Snippet, 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 backticks instead
    // of normal double quotes).
    stmt := `SELECT id, title, content, created, expires FROM snippets
    WHERE expires > UTC_TIMESTAMP() AND id = ?`

    // Use the QueryRow() method on the embedded 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 returned by the database.
    row := db.QueryRow(stmt, id)

    // Initialize a pointer to a new zeroed Snippet struct.
    s := &Snippet{}

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

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

Save the file and restart the application. If you visit `http://localhost:4000/snippet?id=1` in your browser you should see a response like this:



You might also want to try making some requests for other snippets which are expired (`id=3`) or don't yet exist (`id=4`) to verify that they return a `404 Not Found` response:

```
$ curl http://localhost:4000/snippet?id=2
&{2 Over the wintry forest Over the wintry
forest, winds howl in  rage
with no leaves to blow.

– Natsume Soseki 2017-08-21 13:38:01 +0000 UTC 2018-08-21 13:38:01 +0000 UTC}
$ curl http://localhost:4000/snippet?id=3
Not Found
$ curl http://localhost:4000/snippet?id=4
Not Found
```

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.

Placeholder parameters

In the code above we constructed our SQL statement using placeholder parameters, where `?` acted as a placeholder for our `id` value.

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

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

1. It creates a new prepared statement on the database using the provided SQL statement. The database parses and compile the statement, then stores it ready for execution.

2. In a second separate step, `QueryRow()` 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.

There's a few things I should point out.

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:

```
app.DB.QueryRow("SELECT id, title... FROM snippets WHERE id = $1", id)
```

Also, `DB.QueryRow()` is a variadic function; it can be called with any number of trailing arguments. In practice this means that you can pass as many placeholder parameters as you need to. For instance:

```
env.DBPool.QueryRow("SELECT ... WHERE make = ? AND model = ?", make, model)
```

Notes

The database/sql package

As you're probably starting to realize, the `database/sql` package essentially provides a standard interface (or API) between your Go application and the world of SQL databases.

So long as you use the `database/sql` API, 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 tightly coupled to the particular database that you're currently using, and you can potentially swap databases in the future without re-writing all of your code.

It's important to note that while `database/sql` generally does a good job of providing a standard API 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 `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 like this because it can't convert `NULL` into a string:

```
sql: Scan error on column index 1: unsupported Scan, storing driver.Value type
<nil> into type *string
```

The fix for this is to change the 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 general 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 if necessary.

Benefits

Setting your models up in this 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 (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 `models.Database` type and implementing methods on it we've been able to make our MySQL database 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.

- Our models form an importable package *in their own right*. This makes it possible for other code to import and re-use the models too.

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

- *And finally*, because our model actions are defined as methods on an object (in our case `models.Database`), there's the opportunity to create an *interface* and mock it for unit testing purposes.

4.6. Multiple-record SQL Queries

The pattern for using `db.Query()` to return multiple rows is a little more complicated. I'll demonstrate by adding a `LatestSnippets()` method to our database model which will return the most recently created ten snippets (so long as they haven't expired) with 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/database.go` file and add the following code:

pkg/models/database.go

```
package models

...

func (db *Database) LatestSnippets() (Snippets, 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 QueryRow() method on the embedded connection pool to execute our
    // SQL statement. This results a sql.Rows resultset containing the result of
    // our query.
    rows, err := db.Query(stmt)
    if err != nil {
        return nil, err
    }

    // IMPORTANTLY we defer rows.Close() to ensure the sql.Rows resultset is
    // always properly closed before LatestSnippets() returns. Closing a
    // resultset is really important. 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. Another gotcha is that the
    // defer statement should come *after* you check for an error from
    // db.Query(). Otherwise, if db.Query() returns an error, you'll get a panic
    // trying to close a nil resultset.
    defer rows.Close()

    // Initialize an empty Snippets object (remember that this is just a slice of
    // the type []*Snippet).
    snippets := Snippets{}

    // 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 of 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 object.
        s := &Snippet{}
        // Use rows.Scan() to copy the values from each field in the row to the
        // new Snippet object that we created. Again, the arguments to row.Scan()
        // must be pointers to the place you want to copy the data into, and the
        // number of arguments must be exactly the same as the number of
        // columns returned by your statement.
        err := rows.Scan(&s.ID, &s.Title, &s.Content, &s.Created, &s.Expires)
        if err != nil {
            return nil, err
        }
        // Append it to the slice of snippets.
        snippets = append(snippets, s)
    }

    // When the rows.Next() loop has finished we call rows.Err() to retrieve any
    // error that was encountered during the iteration. It's important to
    // call this - don't assume that a successful iteration was completed
    // over the whole resultset.
    if err = rows.Err(); err != nil {
        return nil, err
    }

    // If everything went OK then return the Snippets slice.
    return snippets, nil
}
```

4.7. Executing SQL Statements

Finally let's take a look at the pattern for `db.Exec()`, which is used to execute SQL statements that don't return any rows. To demonstrate we'll add an `InsertSnippet()` method to our database model which should do two things:

1. Insert a new record into the `snippets` table, containing a given title, content and expiry time (in seconds).
2. Return the `id` for the new record.

To insert a new record the SQL statement we want to execute (with placeholder parameters) is:

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

So let's open up the `pkg/models/database.go` file and add the following code:

pkg/models/database.go

```
package models

...

func (db *Database) InsertSnippet(title, content, expires string) (int, error) {
    // Write the SQL statement we want to execute.
    stmt := `INSERT INTO snippets (title, content, created, expires)
    VALUES(?, ?, UTC_TIMESTAMP(), DATE_ADD(UTC_TIMESTAMP(), INTERVAL ? SECOND))`

    // Use the db.Exec() method to execute the statement snippet, passing in values
    // for our (untrusted) title, content and expiry placeholder parameters in
    // exactly the same way that we did with the QueryRow() method. This returns
    // a sql.Result object, which contains some basic information about what
    // happened when the statement was executed.
    result, err := 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 is of type int64, so we convert it to an int type for
    // returning from our Insert function.
    return int(id), nil
}
```

Before we continue, let's quickly discuss the `sql.Result` object 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.
- `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. 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 := s.DB.Exec("INSERT INTO ...", ...)
```

Notes

Working with Transactions

It's important realize that calls to `db.Exec()` can use *any connection from the pool*. Even if you have two calls to `db.Exec()` immediately next to each other in your code, there is no guarantee that they will use the same database connection. The same is true of `db.QueryRow()` and `db.Query()`.

Sometimes this isn't acceptable. For instance, if you lock a table with MySQL's `LOCK TABLES` command you have to 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:

```
func ExampleTransaction(db *sql.DB) error {
    // Calling the Begin() method on the connection pool creates a new sql.Tx
    // object, which represents the in-progress database transaction.
    tx, err := 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 we 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 transaction can be commited
    // to the database with the tx.Commit() method. It's really important to ALWAYS
    // call either Rollback() or Commit() before your function returns. If you
    // don't the connection will stay open and not be returned to the connection
    // pool. This can lead to hitting you maximum connection limit/running out of
    // 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 we 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 will
// wait until 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)
```

This comes with a caveat: there's a good chance that your database itself has a hard limit on the maximum number of connections. For example, the default limit for MySQL is 151. So leaving `SetMaxOpenConns` totally unlimited may result in your database returning "too many connections" errors under high load. To avoid that, you should set the sum total maximum of open and idle connections to comfortably below any hard database limit.

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.

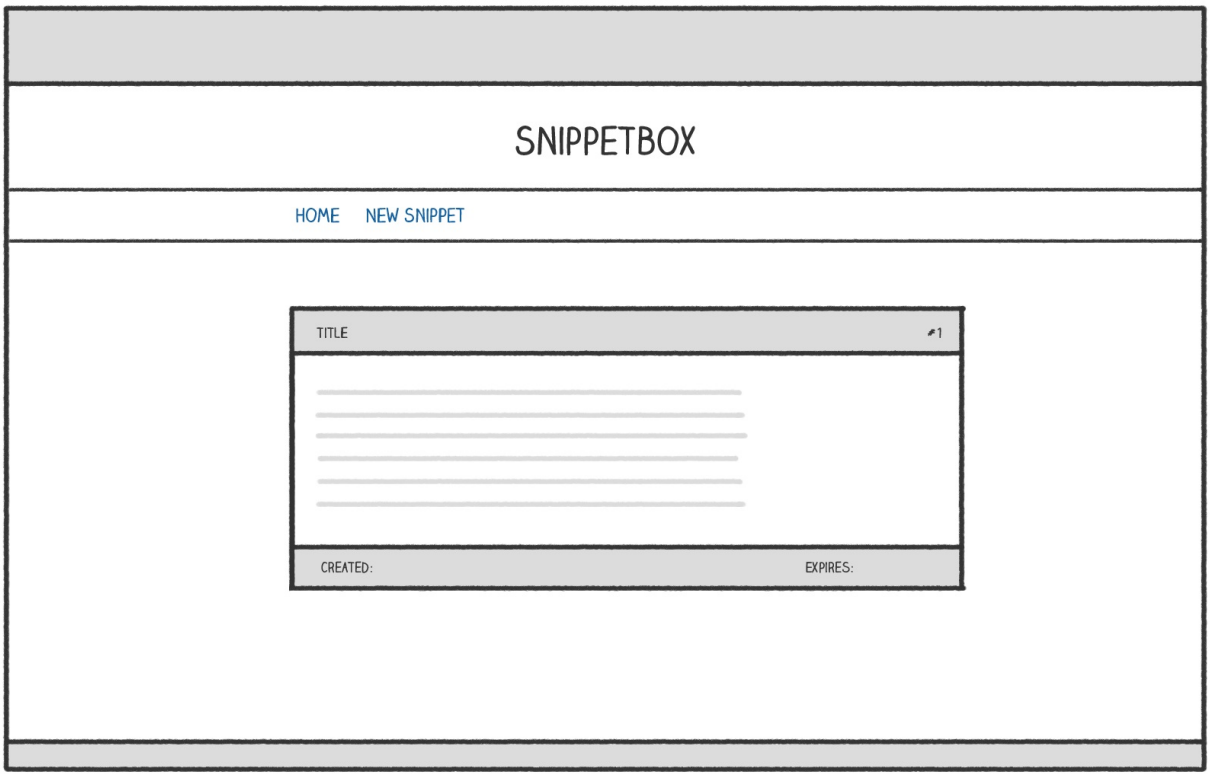
In particular you'll learn how to:

- 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 your own custom functions to format and display data in your HTML templates.
- Gracefully handle template rendering errors at runtime.

5.1. Rendering 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 chapter we'll update the application so that the data is displayed in a proper HTML webpage which looks a bit like this:



Passing the Snippet Data

The first thing to know is that Go's `html/template` package allows you to provide one – and only one – item of dynamic data when rendering a template. But in a real-world application there are often times that you want to display multiple pieces of dynamic data in the same template.

A lightweight and type-safe workaround for this to do this is to wrap your dynamic data in a struct, which acts like a single 'holding structure' for your data.

Let's update the `views.go` file so that:

- It defines a new `HTMLData` struct to hold the dynamic data we want to display in our web pages.
- The signature of the `app.RenderHTML()` helper (which we made earlier) is changed so that it accepts a `HTMLData` object as a parameter and passes it onward to the `template.ExecuteTemplate()` function.

Now, remember the `app.RenderHTML()` helper method that we made earlier? Open up the `views.go` file and update it to take an additional `data` parameter, which we'll use to pass our dynamic data to the HTML template.

cmd/web/views.go

```
package main

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

    "snippetbox.org/pkg/models" // New import
)

// Define a new HTMLData struct to act as a wrapper for the dynamic data we want
// to pass to our templates. For now this just contains the snippet data that we
// want to display, which has the underling type *models.Snippet.
type HTMLData struct {
    Snippet *models.Snippet
}

// Update the signature of RenderHTML() so that it accepts a new data parameter
// containing a pointer to a HTMLData struct.
func (app *App) RenderHTML(w http.ResponseWriter, page string, data *HTMLData) {
    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, page),
    }

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

    // Pass the data parameter straight on to the ExecuteTemplate() method. This
    // will make the dynamic data available to our templates when they are being
    // rendered.
    err = ts.ExecuteTemplate(w, "base", data)
    if err != nil {
        app.ServerError(w, err)
    }
}
```

Now let's head to the `handler.go` file and update our handler functions to use the new `app.RenderHTML()` method, like so

cmd/web/handlers.go

```
package main

import (
    "net/http"
    "strconv"
)

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

    // There's no dynamic data that we want to use in the homepage (yet!) so we
    // pass in nil.
    app.RenderHTML(w, "home.page.html", nil)
}

func (app *App) 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
    }

    snippet, err := app.Database.GetSnippet(id)
    if err != nil {
        app.ServerError(w, err)
        return
    }
    if snippet == nil {
        app.NotFound(w)
        return
    }

    // Render the show.page.html template, passing in the snippet data wrapped in
    // our HTMLData struct.
    app.RenderHTML(w, "show.page.html", &HTMLData{
        Snippet: snippet,
    })
}

...
```

Displaying the Snippet Data

Within your HTML templates, any dynamic data that you pass in is represented by the `.` character (referred to as **dot**).

In our case, the underlying type of dot is either a `HTMLData` struct or nil. 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 `HTMLData` struct has a `Snippet` field we could yield the snippet data by writing `{{.Snippet}}` in our templates.

In turn `{{.Snippet}}` will yield a struct with the underlying type `models.Snippet`. And we can yield its exported fields in exactly the same way. So if we want to yield the value of the `ID` field in our templates we can write `{{.Snippet.ID}}`.

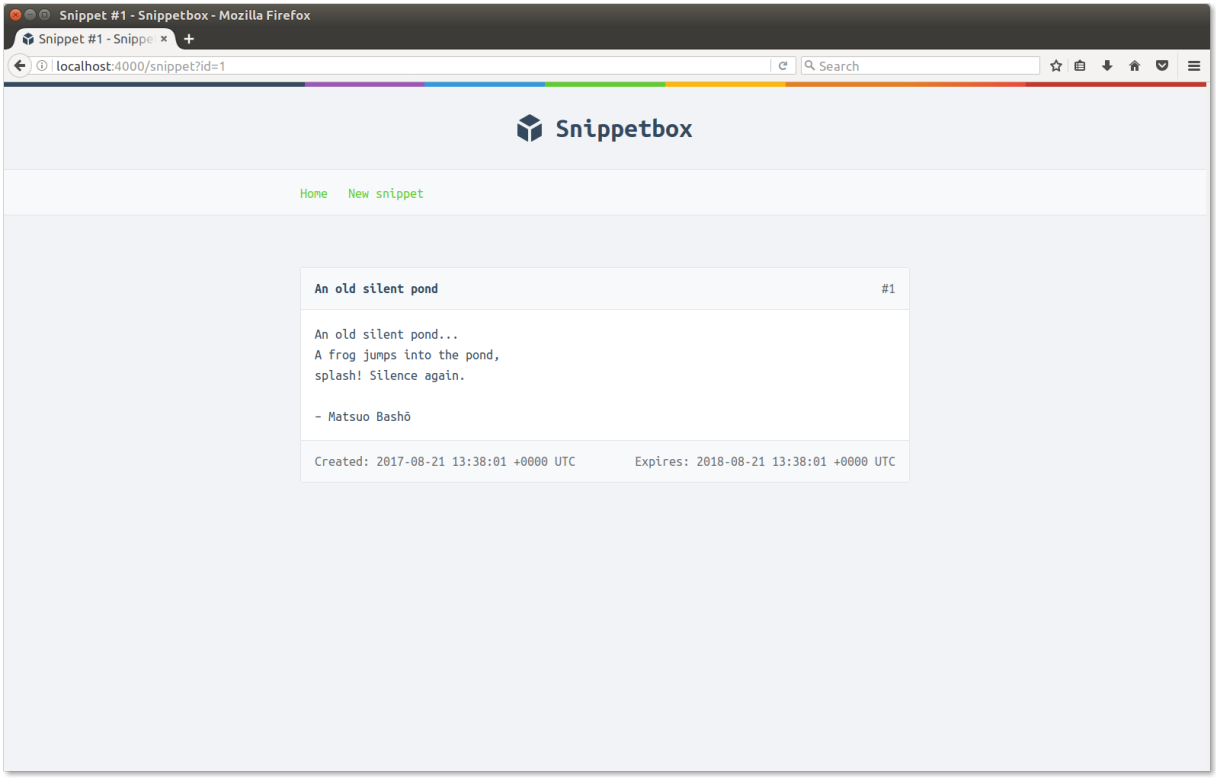
I'll demonstrate. Let's create an empty `ui/html/show.page.html` file and add the following markup:

```
$ cd $HOME/go/src/snippetbox.org
$ touch ui/html/show.page.html
```

ui/html/show.page.html

```
{{define "page-title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "page-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}}
```

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, added to the `HTMLData` struct, passed to our template, and the content is rendered correctly.



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 because the `page-title` and `page-body` templates are embedded (by the `base` template) they don't have access to dot by default. They only have access to dot because we explicitly passed or **pipelined** dot at the end of the `{{template}}` action in our `ui/html/base.html` file, like so:

```
...
<title>{{template "page-title" .}} - Snippetbox</title>
...
<section>
    {{template "page-body" .}}
</section>
...
```

Notes

HTML Comments

The `html/template` package always strips out any HTML comments you include in your templates, including 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.

5.2. Template Actions

In this chapter we're going to look at three **template actions** that Go provides 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 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's 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 these 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.*

Using the With Action

A good opportunity to use the `{{with}}` action is the `ui/html/show.page.html` file that we created in the previous chapter. Go ahead and update it like so:

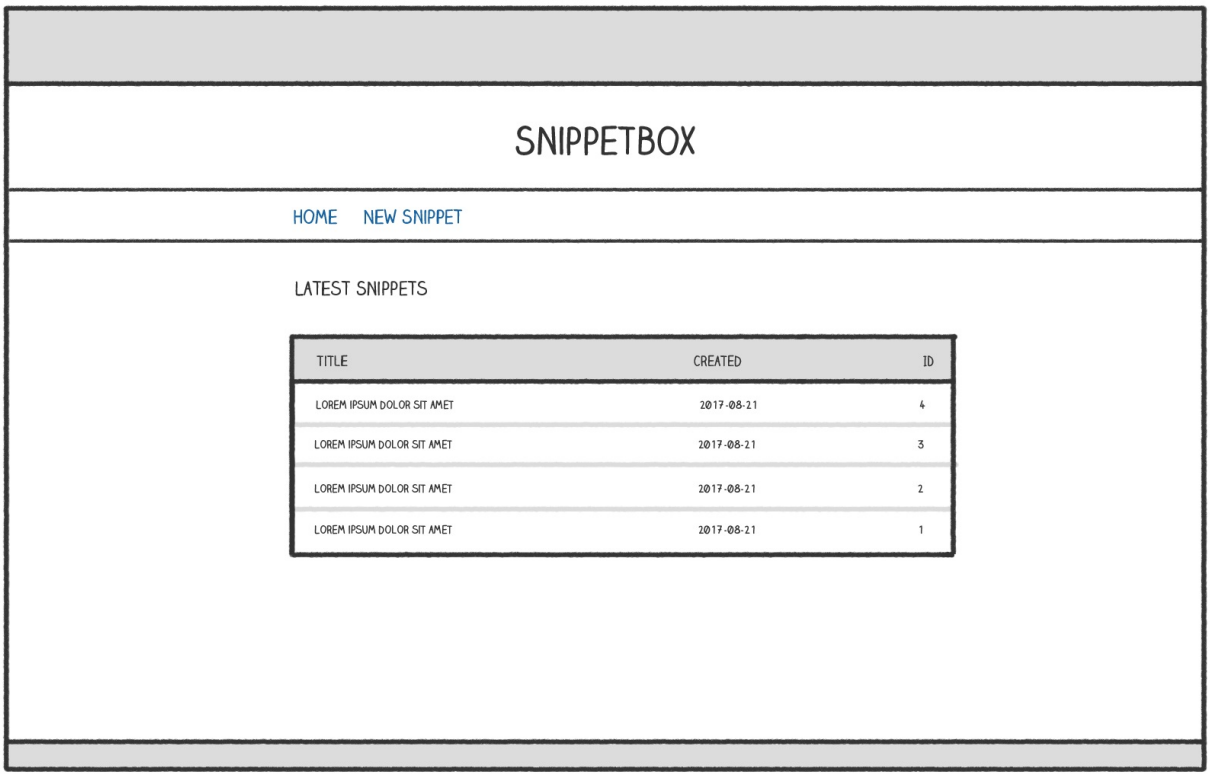
ui/html/show.page.html

```
{{define "page-title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "page-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}}
```

In this markup now, inside the `{{with .Snippet}}...{{end}}` tags the value of dot is set to `.Snippet`. Dot essentially becomes the `models.Snippet` struct instead of the parent `HTMLData` 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:



First update the `Home` handler function so that it fetches the latest snippets from our database model and passes them to our `home.page.html` templates:

cmd/web/handlers.go

```
package main

...

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

    // Fetch a slice of the latest snippets from the database.
    snippets, err := app.Database.LatestSnippets()
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // Pass the slice of snippets to the "home.page.html" templates.
    app.RenderHTML(w, "home.page.html", &HTMLData{
        Snippets: snippets,
    })
}

...
```

We'll then need to update the `HTMLData` struct so that it contains a `Snippets` field for holding a slice of snippets, like so:

cmd/web/views.go

```
package main

...

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

...
```

Lastly we'll update the `ui/html/home.page.html` file to display the 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.

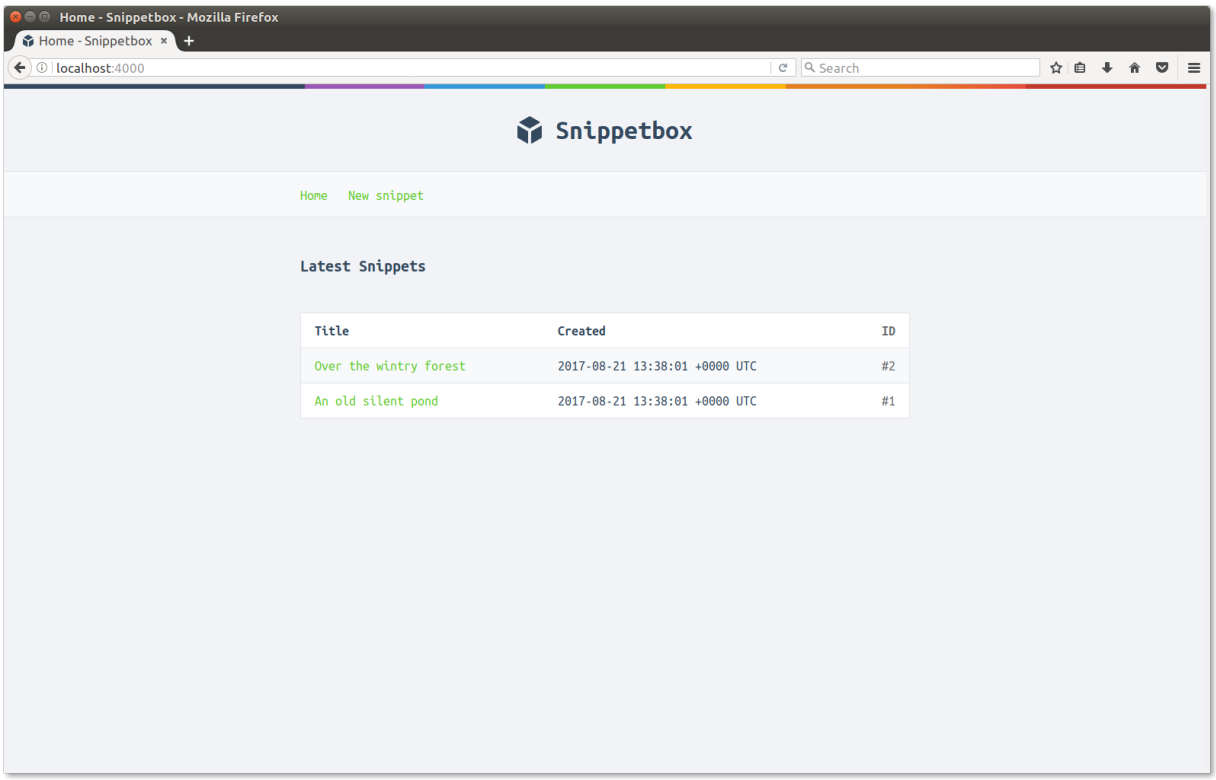
- We want to use the `{{range}}` action to iterate over all snippets in the slice, rendering the contents of each snippet in a table row.

Here's the markup:

ui/html/home.page.html

```
{{define "page-title"}}Home{{end}}

{{define "page-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:


Home - Snippetbox - Mozilla Firefox
Home - Snippetbox
localhost:4000
Search
Snippetbox
Home
New snippet
Latest Snippets
Title
Created
ID
Over the wintry forest
2017-08-21 13:38:01 +0000 UTC
#2
An old silent pond
2017-08-21 13:38:01 +0000 UTC
#1

5.3. Template Functions

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.
{{printf "%s-%s" .Foo .Bar}}	Yields a formated 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. Variables names must be prefixed by a dollar sign and can contain alphanumeric characters only.

Notice too the signature for calling template functions? Unlike the Go language itself parameters for each function are separated by a space and no parentheses are used.

Using Custom Functions

It's also possible to create your own custom template functions.

For example, in our application the `created` and `expires` dates look a bit ugly. It would be nice to display a 'humanized' representation of these `time.Time` objects, like "02 Jan 2006 at 15:04" instead of just outputting the default string representation.

Something like this is a perfect candidate for a custom template function. So in the rest of this chapter we'll focus on how to create a `humanDate` template function, which we can use in our templates like so:

```
<time>Created: {{humanDate .Created}}</time>
```

Theres two main steps to doing this:

1. We need to create a `template.FuncMap` object containing our 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 `views.go` file:

cmd/web/views.go

```
package main

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

    "snippetbox.org/pkg/models"
)

// Create a humanDate function which returns a nicely formated string
// representation of a time.Time object.
func humanDate(t time.Time) string {
    return t.Format("02 Jan 2006 at 15:04")
}

...

func (app *App) RenderHTML(w http.ResponseWriter, page string, data *HTMLData) {
    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, page),
    }

    // Initialize a template.FuncMap object. This is essentially a string-keyed map
    // which acts as a lookup between the names of our custom template functions and
    // the functions themselves.
    fm := template.FuncMap{
        "humanDate": humanDate,
    }

    // Our template.FuncMap must be registered with the template set before we call
    // the ParseFiles() method. This means we have to use template.New() to create
    // an empty, unnamed, template set, use the Funcs() method to register our
    // template.FuncMap, and then parse the files as normal.
    ts, err := template.New("").Funcs(fm).ParseFiles(files...)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    err = ts.ExecuteTemplate(w, "base", data)
    if err != nil {
        app.ServerError(w, err)
    }
}
```

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:

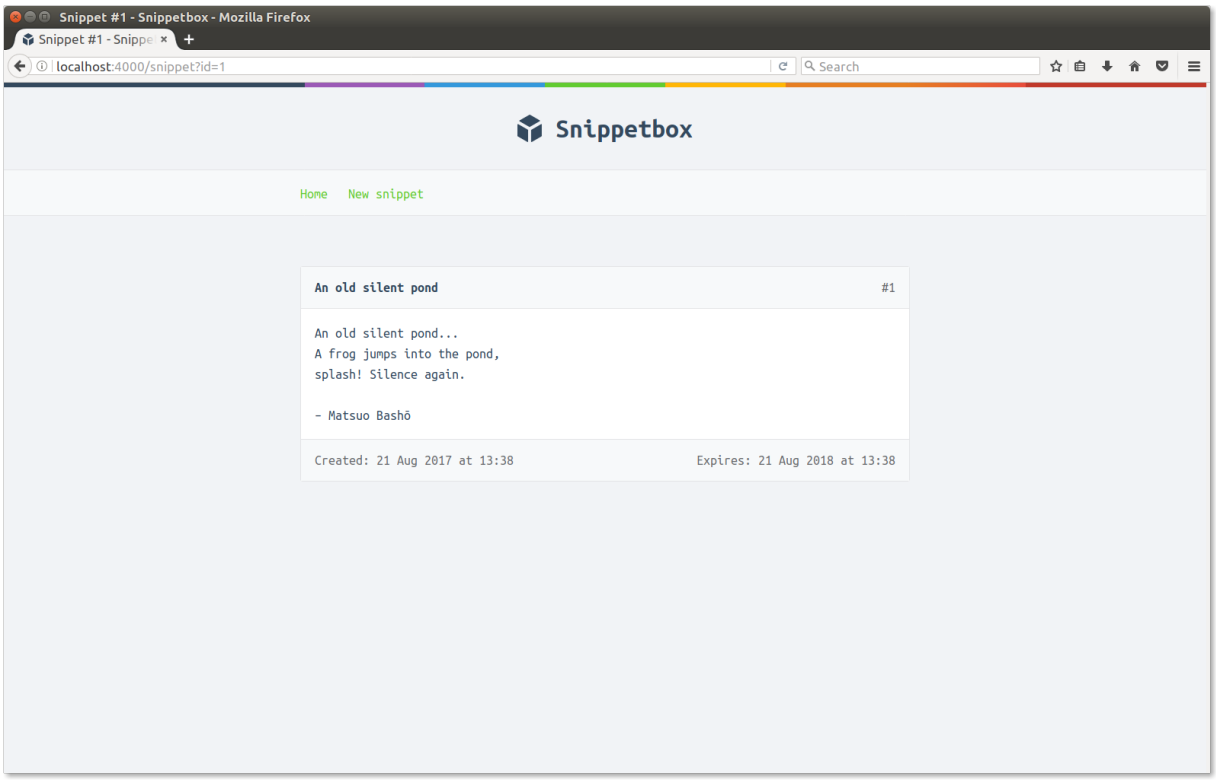
ui/html/home.page.html

```
{{define "page-title"}}Home{{end}}

{{define "page-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>{{humanDate .Created}}</td>
            <td>#{{.ID}}</td>
        </tr>
        {{end}}
    </table>
    {{else}}
        <p>There's nothing to see here yet!</p>
    {{end}}
{{end}}
```

ui/html/show.page.html

```
{{define "page-title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "page-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: {{humanDate .Created}}</time>
                <time>Expires: {{humanDate .Expires}}</time>
            </div>
        </div>
    {{end}}
{{end}}
```

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



Notes

Pipelining

In the code above we called our 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>
```

Calling Methods

Finally, if the object that you're yielding on has methods defined against it, you can call those (so long as they are exported and they return only a single value – or a single value and an error).

For example, if `.Created` has the underlying type `time.Time` you could render the name of the weekday by calling the `time.Weekday` method like so:

```
<time>{{.Created.Weekday}}</time>
```

You can also pass parameters to methods, in the same way that you can to functions. For example, you could use the `time.AddDate` method to add six months to a time like so:

```
<time>{{.Created.AddDate 0 6 0}}</time>
```

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 our `show.page.html` template and see what happens:

ui/html/show.page.html

```
{{define "page-title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "page-body"}}
    <!-- Start deliberate error. -->
    {{len nil}}
    <!-- End deliberate error -->
    {{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: {{humanDate .Created}}</time>
                <time>Expires: {{humanDate .Expires}}</time>
            </div>
        </div>
    {{end}}
{{end}}
```

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.

If you try to run the application now you should find that everything still compiles OK:

```
$ go run cmd/web/*
2017/08/24 18:28:19 Starting server on :4000
```

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

```
$ curl -i http://localhost:4000/snippet?id=1
HTTP/1.1 200 OK
Date: Thu, 24 Aug 2017 16:30:39 GMT
Content-Length: 551
Content-Type: text/html; charset=utf-8

<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Snippet #1 - Snippetbox</title>
        <link rel="stylesheet" href="/static/css/main.css">
        <link rel="shortcut icon" href="/static/img/favicon.ico" type="image/x-icon">
    </head>
    <body>
        <header>
            <h1><a href="/">Snippetbox</a></h1>
        </header>
        <nav>
            <a href="/">Home</a>
            <a href="/snippet/new">New snippet</a>
        </nav>
        <section>

    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 the trial renders 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 `views.go` file to use this approach instead:

cmd/web/views.go

```
package main

import (
    "bytes" // New import
    "html/template"
    "net/http"
    "path/filepath"
    "time"

    "snippetbox.org/pkg/models"
)

...

func (app *App) RenderHTML(w http.ResponseWriter, page string, data *HTMLData) {
    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, page),
    }

    funcs := template.FuncMap{
        "humanDate": humanDate,
    }

    ts, err := template.New("").Funcs(funcs).ParseFiles(files...)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // Initialize a new buffer.
    buf := new(bytes.Buffer)

    // Write the template to the buffer, instead of straight to the
    // http.ResponseWriter. If there's an error, call our error handler and then
    // return.
    err = ts.ExecuteTemplate(buf, "base", data)
    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)
}
```

Save this file, restart the application and try making the same request. 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: Thu, 24 Aug 2017 16:33:47 GMT
Content-Length: 22

Internal Server Error
```

Great stuff.

Before we move on to the next chapter, don't forget to head back to the `show.page.html` file and remove the

deliberate error!

ui/html/show.page.html

```
{{define "page-title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "page-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: {{humanDate .Created}}</time>
                <time>Expires: {{humanDate .Expires}}</time>
            </div>
        </div>
    {{end}}
{{end}}
```

5.5. Common Dynamic Data

In some web applications there may be common dynamic data that you want to pass to every webpage.

In our case, it would be nice to include the current URL path in our dynamic data each time we render a webpage. This would enable us to add a CSS class to highlight the currently 'live' or active item in our navigation bar, like so:

ui/html/base.html

```
{{define "base"}}
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>{{template "page-title" .}} - Snippetbox</title>
        <link rel="stylesheet" href="/static/css/main.css">
        <link rel="shortcut icon" href="/static/img/favicon.ico" type="image/x-icon">
    </head>
    <body>
        <header>
            <h1><a href="/">Snippetbox</a></h1>
        </header>
        <nav>
            <a href="/" {{if eq .Path "/"}}class="live"{{end}}>
                Home
            </a>
            <a href="/snippet/new" {{if eq .Path "/snippet/new"}}class="live"{{end}}>
                New snippet
            </a>
        </nav>
        <section>
            {{template "page-body" .}}
        </section>
    </body>
</html>
{{end}}
```

Let's update our `views.go` file to do exactly that:

cmd/web/views.go

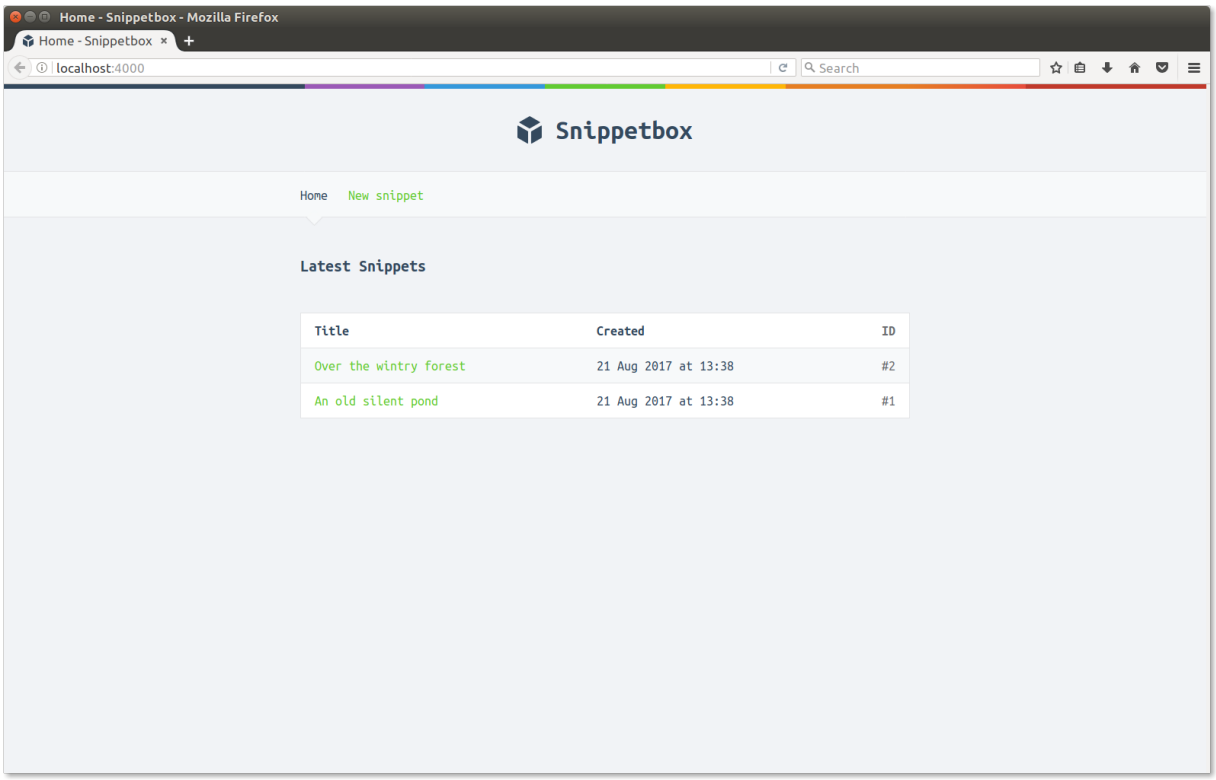
```
package main

...

// Add a Path field to the struct.
type HTMLData struct {
    Path     string
    Snippet  *models.Snippet
    Snippets []*models.Snippet
}

// Change the signature of the RenderHTML() method so that it accepts *http.Request
// as the second parameter.
func (app *App) RenderHTML(w http.ResponseWriter, r *http.Request, page string, data *HTMLData) {
    // If no data has been passed in, initialize a new empty HTMLData object.
    if data == nil {
        data = &HTMLData{}
    }

    // Add the current request URL path to the data.
    data.Path = r.URL.Path

    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, page),
    }

    funcs := template.FuncMap{
        "humanDate": humanDate,
    }

    ts, err := template.New("").Funcs(funcs).ParseFiles(files...)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    buf := new(bytes.Buffer)
    err = ts.ExecuteTemplate(buf, "base", data)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    buf.WriteTo(w)
}
```

Now we just need to update the `handlers.go` file so that our calls to `app.RenderHTML()` use the correct signature:

cmd/web/handlers.go

```
package main

...

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

    snippets, err := app.Database.LatestSnippets()
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // Include the *http.Request parameter.
    app.RenderHTML(w, r, "home.page.html", &HTMLData{
        Snippets: snippets,
    })
}

func (app *App) 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
    }

    snippet, err := app.Database.GetSnippet(id)
    if err != nil {
        app.ServerError(w, err)
        return
    }
    if snippet == nil {
        app.NotFound(w)
        return
    }

    // Include the *http.Request parameter.
    app.RenderHTML(w, r, "show.page.html", &HTMLData{
        Snippet: snippet,
    })
}

...
```

If you restart the application and visit the homepage at `http://localhost:4000/` you should now see the current navigation link highlighted like so:


Home - Snippetbox - Mozilla Firefox
Home - Snippetbox
localhost:4000
Search
Snippetbox
Home
New snippet
Latest Snippets
Title
Created
ID
Over the wintry forest
21 Aug 2017 at 13:38
#2
An old silent pond
21 Aug 2017 at 13:38
#1

6. RESTful Routing

In the next section of this book we're going to add an HTML form to our `"/snippet/new"` route so that users can add a snippet via the web application.

But before we do this, it's sensible follow HTTP good-practice and ensure that any state-changing, non-idempotent, HTTP requests (like adding a new snippet) use the `POST` HTTP method (rather than the `GET` method).

While we're at it, it makes sense to also restrict our other actions – which simply return information – to only support `GET` (and `HEAD`) requests. Essentially, we want our application routes to end up looking like this:

Method	URL Path	Handler	Action
GET	/	Home	Display the homepage
GET	/snippet?id=1	ShowSnippet	Display a specific snippet
GET	/snippet/new	NewSnippet	Display the new snippet form
POST	/snippet/new	NewSnippet	Create a new snippet
GET	/static/...	http.FileServer	Serve a specific static file

Another nice 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	URL Path	Handler	Action
GET	/	Home	Display the homepage
GET	/snippet/:id	ShowSnippet	Display a specific snippet
GET	/snippet/new	NewSnippet	Display the new snippet form
POST	/snippet/new	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 serve mux unfortunately doesn't support method based routing or semantic URLs with variables in them. There are some tricks you can use to get around this in simple cases, 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.

6.1. Choosing a Router

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

Out of all the third-party routers I've tried there are probably three that I'd recommend: Pat, Bone and Gorilla Mux. All have good documentation, decent test coverage, and work well with the `net/http` package.

- Pat is the most focused of the three routers. It provides method-based routing and support for semantic URLs, but not much else. But it's solid at what it does, and the code itself is very clear and well-written. A possible negative is that the package isn't really updated anymore.

- Bone provides similar functionality to Pat, but with extra convenience functions for registering handler functions and support for regular-expression based routes.

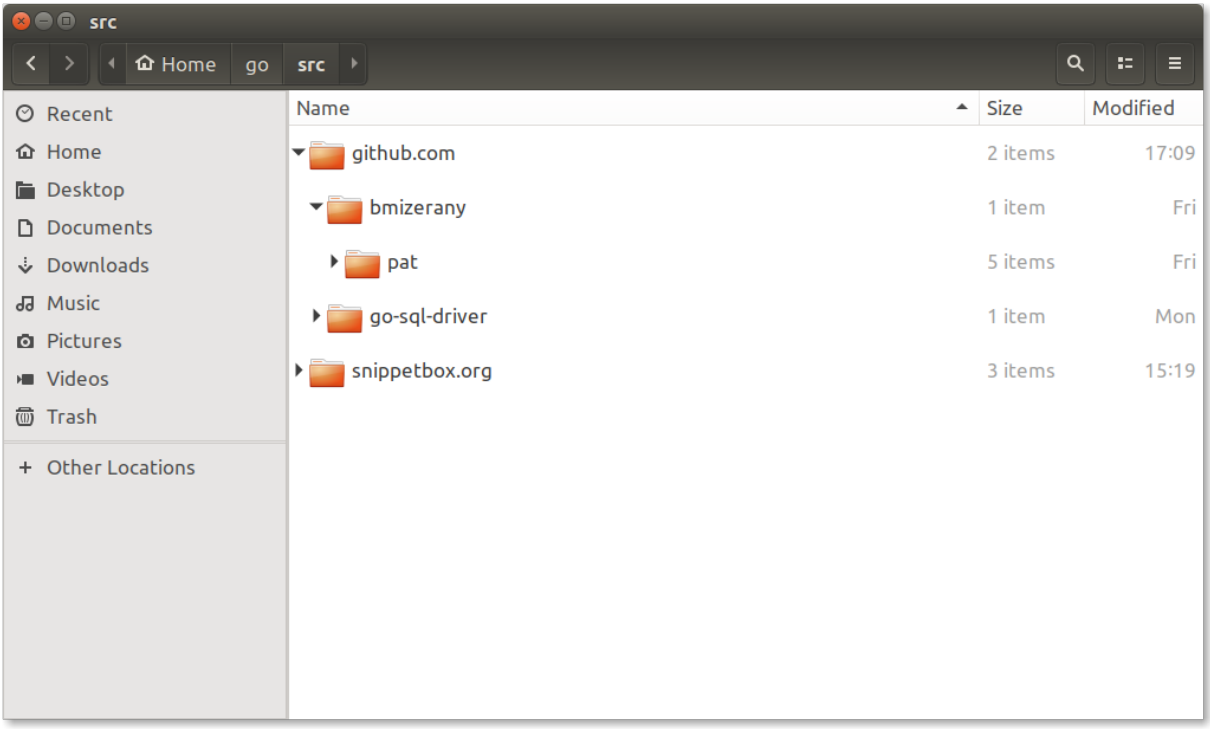
- Gorilla Mux is the most full-featured of the three routers and extremely popular. In addition to routing based on methods, semantic URLs and regular expressions, you can use it to route based on scheme, host and headers. The downside of the feature-completeness is that it's comparatively slow and memory hungry. Although for a database-driven web application like ours, the speed differential over the lifetime of a whole HTTP request is likely to be negligible.

Installing Pat

For the Snippetbox web application our needs are simple. We don't need the more advanced features that Bone or Gorilla Mux provide, so Pat is a good fit for our needs.

If you're following along, use `go get` to install the `pat` package on your computer:

```
$ go get github.com/bmizerany/pat
```


src
Home
go
src
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
Name
Size
Modified
github.com
2 items
17:09
bmizerany
1 item
Fri
pat
5 items
Fri
go-sql-driver
1 item
Mon
snippetbox.org
3 items
15:19

6.2. Implementing RESTful Routes

In this chapter we'll port our application from using Go's inbuilt serve mux to using Pat.

The basic syntax for creating a router and registering a route with Pat looks like this:

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

- 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 adds the contents of the named capture to the URL query string at runtime behind the scenes.
- We use the `mux.Get()` method 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 we need to convert them using the `http.HandlerFunc()` adapter – just like we were doing at the start of the book.

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

cmd/web/routes.go

```
package main

import (
    "net/http"

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

// Change the signature so we're returning a http.Handler instead of a
// *http.ServeMux.
func (app *App) Routes() http.Handler {
    mux := pat.New()
    mux.Get("/", http.HandlerFunc(app.Home))
    mux.Get("/snippet/new", http.HandlerFunc(app.NewSnippet))
    mux.Post("/snippet/new", http.HandlerFunc(app.CreateSnippet))
    mux.Get("/snippet/:id", http.HandlerFunc(app.ShowSnippet)) // Moved downwards

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

    return mux
}
```

There a few important things to point out here.

- Pat matches patterns *in the order that they were registered*. This means that we need to register the route for our `"/snippet/:id"` pattern *after* registering `"/snippet/new`. Otherwise a request for `GET "/snippet/new"` would be dispatched to the `app.ShowSnippet()` function instead of `app.NewSnippet()` like it should.
- URL patterns which end in a trailing slash (like `"/static/"` in our code above) work similar to Go's inbuilt serve mux. 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 `"/"`.

There's also a couple of changes we need to make to our `handlers.go` file:

cmd/web/handlers.go

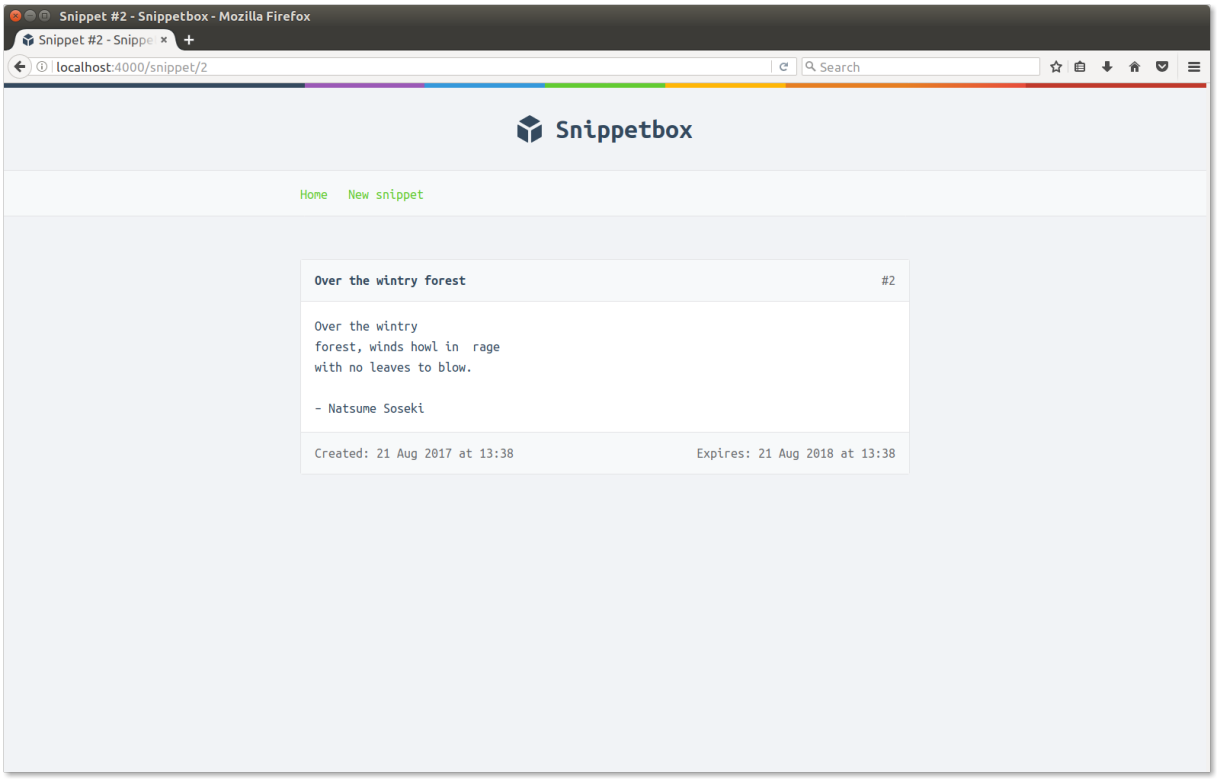
```
package main

...

// Because Pat matches the "/" path exactly, we can now remove the manual check
// of r.URL.Path != "/" from the Home function.
func (app *App) Home(w http.ResponseWriter, r *http.Request) {
    snippets, err := app.Database.LatestSnippets()
    if err != nil {
        app.ServerError(w, err)
        return
    }

    app.RenderHTML(w, r, "home.page.html", &HTMLData{
        Snippets: snippets,
    })
}

func (app *App) 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
    }

    snippet, err := app.Database.GetSnippet(id)
    if err != nil {
        app.ServerError(w, err)
        return
    }
    if snippet == nil {
        app.NotFound(w)
        return
    }

    app.RenderHTML(w, r, "show.page.html", &HTMLData{
        Snippet: snippet,
    })
}

...

// Add a new placeholder handler function for creating a snippet.
func (app *App) CreateSnippet(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Create a new snippet..."))
}
```

Finally, we need to update the table in our `home.page.html` file so that the links use the correct URL format:

ui/html/home.page.html

```
{{define "page-title"}}Home{{end}}

{{define "page-body"}}
    <h2>Latest Snippets</h2>
    {{if .Snippets}}
    <table >
        <tr>
            <th>Title</th>
            <th>Created</th>
            <th>ID</th>
        </tr>
        {{range .Snippets}}
        <tr>
            <!-- Interpolate the ID, instead of appending it in a query string. -->
            <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, you should be able to restart the application and now view the text snippets via the nice semantic URLs. For instance: `http://localhost:4000/snippet/2`.



You can also use curl to verify that requests are being dispatched to different methods based on their HTTP method:

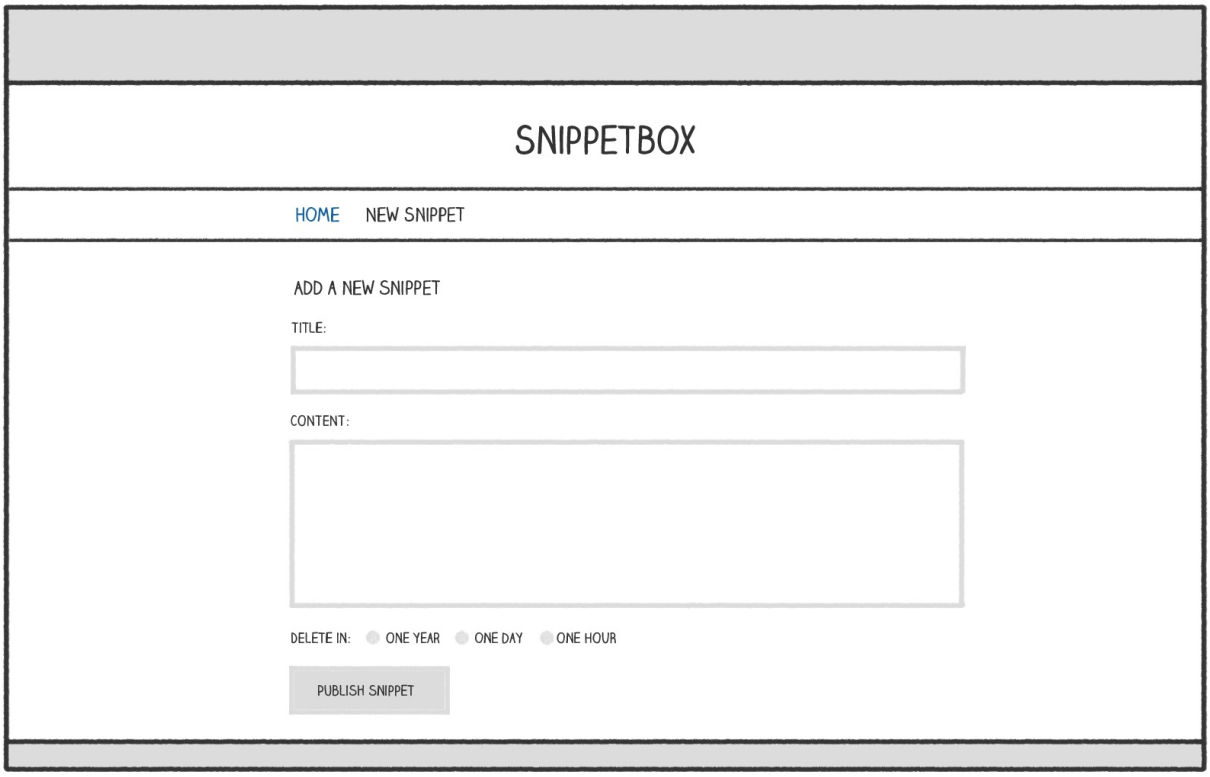
```
$ curl -i http://localhost:4000/snippet/new
HTTP/1.1 200 OK
Date: Fri, 25 Aug 2017 11:12:01 GMT
Content-Length: 31
Content-Type: text/plain; charset=utf-8

Display the new snippet form...
$ curl -i -X POST http://localhost:4000/snippet/new
HTTP/1.1 200 OK
Date: Fri, 25 Aug 2017 11:12:10 GMT
Content-Length: 23
Content-Type: text/plain; charset=utf-8

Create a new snippet...
```

Similarly, you can also see that requests using an unsupported HTTP method are met with a `405 Method Not Allowed` response. For example:

```
$ curl -i -X DELETE http://localhost:4000/
HTTP/1.1 405 Method Not Allowed
Allow: HEAD, GET
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Fri, 25 Aug 2017 11:13:01 GMT
Content-Length: 19

Method Not Allowed
```

7. Processing Forms

In this section of the book we've going to add the ability for a user to create and publish a new text snippet via the web application, starting off with an HTML form that looks a bit like this:



In this section you'll learn:

- How to parse and access form data sent in a `POST` request.
- Some techniques for performing common validation checks on the form data.
- A user-friendly pattern for alerting the user to validation failures and re-populating form fields with previously submitted data.
- How to keep your handlers clean by putting your form validation into a separate reusable package.

7.1. Parsing and Validation

Let's start by adding an HTML form to our `GET "/snippets/new"` route so that our users can create and save their own text snippets.

For processing this form we'll follow a standard `Post-Redirect-Get` pattern like so:

1. The user is shown the blank form when they visit the `"/snippet/new"` page.

2. The user completes the form and it's submitted to the server via a `POST` request to `"/snippet/new"`, which in turn will be passed to our `CreateSnippet` handler.

3. The form data is 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 we'll redirect the user to `"/snippet/:id"`.

If you're following along, let's being by making a new `ui/html/new.page.html` file containing the markup below.

```
$ cd $HOME/go/src/snippetbox.org
$ touch ui/html/new.page.html
```

ui/html/new.page.html

```
{{define "page-title"}}Add a New Snippet{{end}}

{{define "page-body"}}
<form action="/snippet/new" 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="31536000" checked> One Year
        <input type="radio" name="expires" value="86400"> One Day
        <input type="radio" name="expires" value="3600"> One Hour
    </div>
    <div>
        <input type="submit" value="Publish snippet">
    </div>
</form>
{{end}}
```

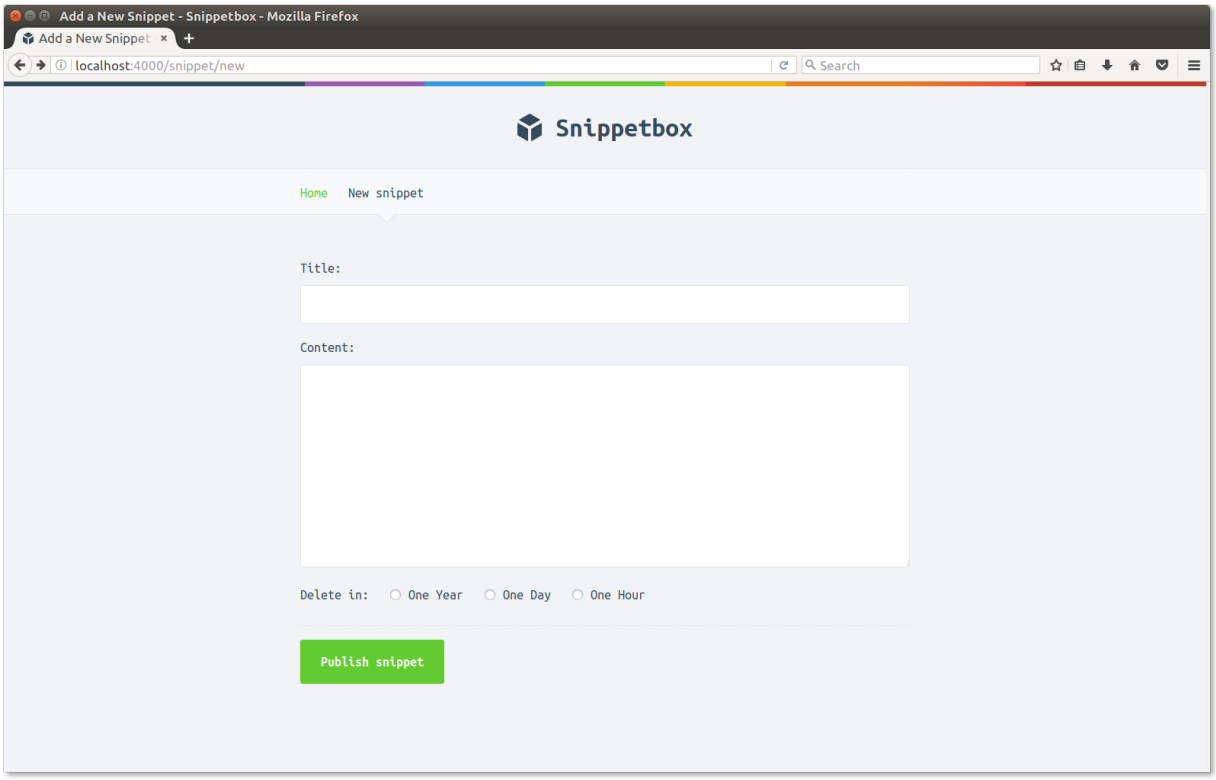
There's nothing special about this markup so far. It's just a standard HTML5 web form which sends three form values: `title`, `content` and `expires` (which is the number of seconds until the snippet should expire). The only thing to really point out is the form's `action` attribute – we've set it up so that the form will `POST` the data to the URL `/snippet/new` when it's submitted.

Let's update our `NewSnippet` handler to display this page, like so:

cmd/web/handlers.go

```
package main

...

func (app *App) NewSnippet(w http.ResponseWriter, r *http.Request) {
    app.RenderHTML(w, r, "new.page.html", nil)
}

...
```

Try firing up the application and visiting `http://localhost:4000/snippet/new`. You should see a form which looks like this:



Processing the Form Data

The `POST "/snippets/new"` route is handled by our `CreateSnippet` function, which we'll now update to validate the form data when it's submitted. At a high-level we can break this down into three distinct steps.

1. We first need to use the `r.ParseForm()` method to parse the request body. This checks that the request body is well-formed, and 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 `""`. This is similar to the way that query string parameters worked earlier in the book.

3. We can then validate the individual form values using the various functions in the `strings` and `unicode/utf8` packages.

We *could* do all of this inline in our `CreateSnippet` handler, but I find it easier and cleaner to break out the logic into a separate file or package. For our build, let's add a reusable `pkg/forms` package to our application containing the following code:

```
$ mkdir pkg/forms
$ touch pkg/forms/forms.go
```

pkg/forms/forms.go

```
package forms

// Declare a struct to hold the form values (and also a map to hold any validation
// failure messages).
type NewSnippet struct {
    Title    string
    Content  string
    Expires  string
    Failures map[string]string
}

// Implement an Valid() method which carries out validation checks on the form
// fields and returns true if there are no failures.
func (f *NewSnippet) Valid() bool {
    f.Failures = make(map[string]string)

    // We will validate the form fields here...

    return len(f.Failures) == 0
}
```

Let's hook this up to our `CreateSnippet` handler:

cmd/web/handlers.go

```
package main

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

    "snippetbox.org/pkg/forms" // New import
)

...

func (app *App) CreateSnippet(w http.ResponseWriter, r *http.Request) {
    // First we call r.ParseForm() which adds any POST (also PUT and PATCH) data
    // to the r.PostForm map. If there are any errors we use our
    // app.ClientError helper to send a 400 Bad Request response to the user.
    err := r.ParseForm()
    if err != nil {
        app.ClientError(w, http.StatusBadRequest)
        return
    }

    // We initialize a *forms.NewSnippet object and use the r.PostForm.Get() method
    // to assign the data to the relevant fields.
    form := &forms.NewSnippet{
        Title:   r.PostForm.Get("title"),
        Content: r.PostForm.Get("content"),
        Expires: r.PostForm.Get("expires"),
    }

    // Check if the form passes the validation checks. If not, then use the
    // fmt.Fprint function to dump the failure messages to the response body.
    if !form.Valid() {
        fmt.Fprint(w, form.Failures)
        return
    }

    // If the validation checks have been passed, call our database model's
    // InsertSnippet() method to create a new database record and return it's ID
    // value.
    id, err := app.Database.InsertSnippet(form.Title, form.Content, form.Expires)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // If successful, send a 303 See Other response redirecting the user to the
    // page with their new snippet.
    http.Redirect(w, r, fmt.Sprintf("/snippet/%d", id), http.StatusSeeOther)
}
```

Validation

So that's the basic pattern, but it's not much use without any actual validation rules. Let's head back to `pkg/forms/forms.go` and add some:

pkg/forms/forms.go

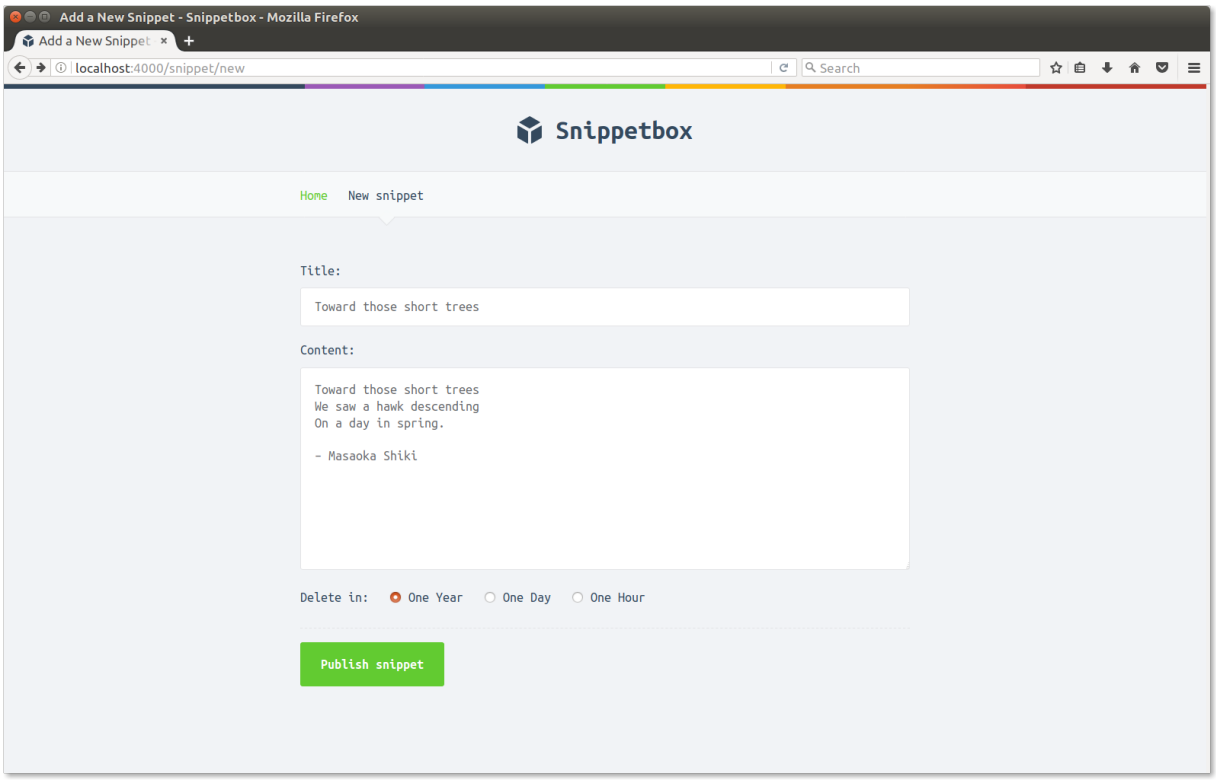
```
package forms

import (
    "strings"      // New import
    "unicode/utf8" // New import
)

type NewSnippet struct {
    Title    string
    Content  string
    Expires  string
    Failures map[string]string
}

func (f *NewSnippet) Valid() bool {
    f.Failures = make(map[string]string)

    // Check that the Title field is not blank and is not more than 100 characters
    // long. If it fails either of those checks, add a message to the f.Failures
    // map using the field name as the key.
    if strings.TrimSpace(f.Title) == "" {
        f.Failures["Title"] = "Title is required"
    } else if utf8.RuneCountInString(f.Title) > 100 {
        f.Failures["Title"] = "Title cannot be longer than 100 characters"
    }

    // Validate the Content and Expires fields aren't blank in a similar way.
    if strings.TrimSpace(f.Content) == "" {
        f.Failures["Content"] = "Content is required"
    }

    // Check that the Expires field isn't blank and is one of a fixed list. Using
    // a lookup on a map keyed with the permitted options and values of true is a
    // neat trick which saves you looping over the permitted values.
    permitted := map[string]bool{"3600": true, "86400": true, "31536000": true}
    if strings.TrimSpace(f.Expires) == "" {
        f.Failures["Expires"] = "Expiry time is required"
    } else if !permitted[f.Expires] {
        f.Failures["Expires"] = "Expiry time must be 3600, 86400 or 31536000 seconds"
    }

    // If there are no failure messages, return true.
    return len(f.Failures) == 0
}
```

A couple of things to point out:

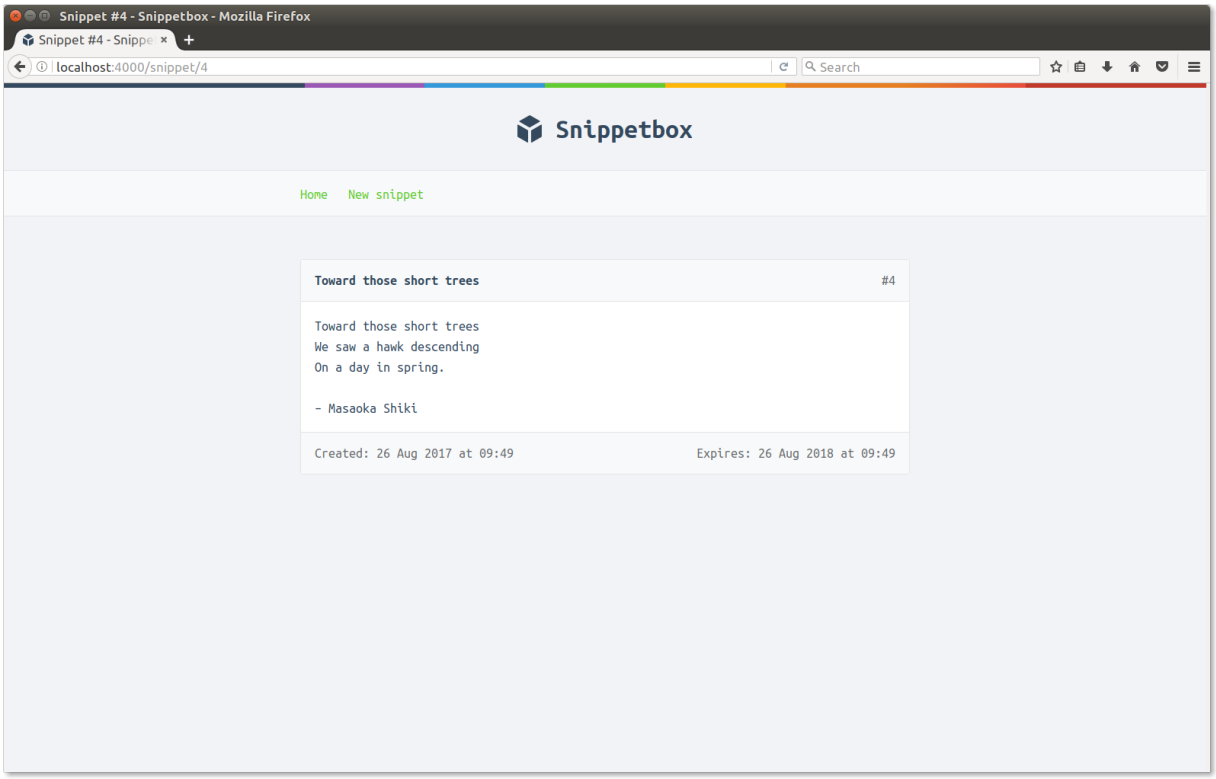
- I've used the `utf8.RuneCount()` function instead of the builtin `len` function because we want to count the number of *characters* in the title. To illustrate the difference, the string `"Zoë"` has 3 characters but a length of 4 bytes because of the umlaut.

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

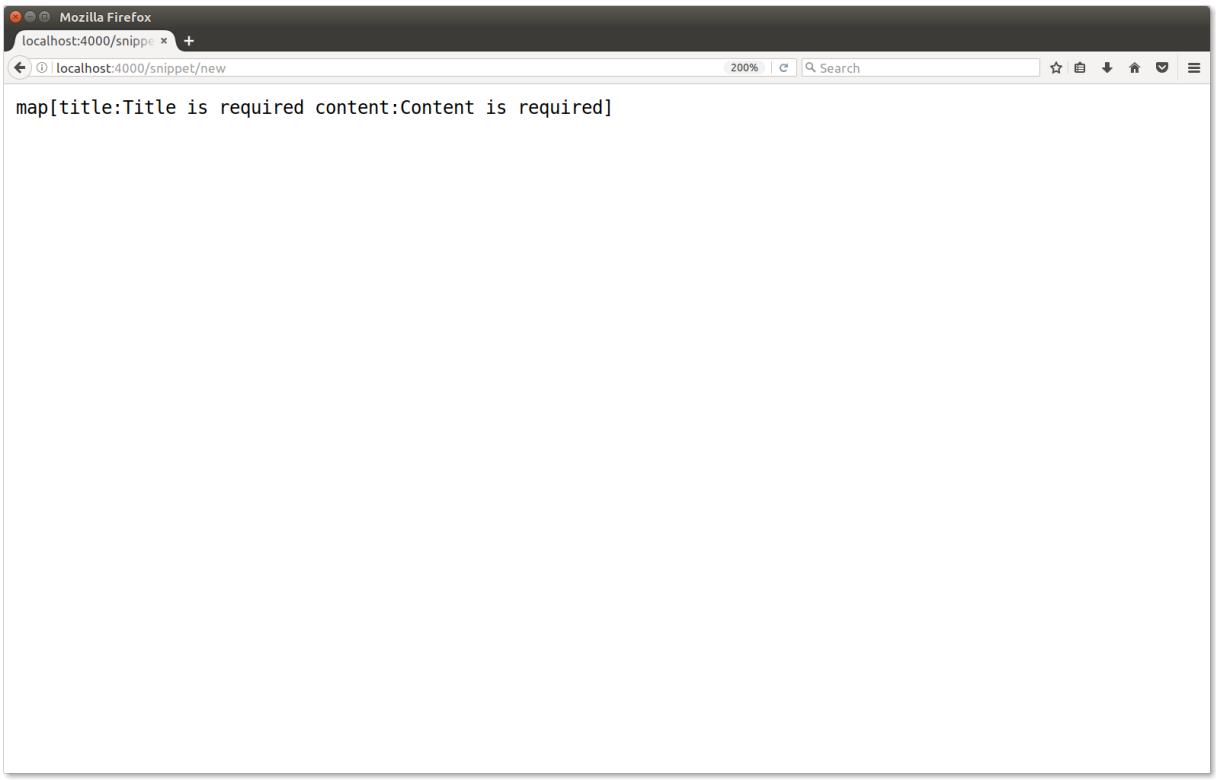
Alright, let's give this a try! Restart the application and try filling in the form with a valid snippet title and content, a bit like this:



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



You might also want to try submitting some blank or invalid forms. In those cases you should see a dump of the validation failure messages, like so:



Notes

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 be populated for all request methods, 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")`. In the event of a conflict, the request body value will take precedent over the query string parameter.

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. However in our case that's not 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 `r.FormValue()` and `r.PostFormValue()` methods. 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 they won't get any feedback 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 we can't use it with form fields which 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
```

To get around this we need to work with `r.PostForm` 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:

```
if len(r.PostForm["items"]) == 0 {
    failures["items"] = "At least one item must be checked"
}
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 is subsequently surfaced by `r.ParseForm()`.

Additionally, if the limit is hit, `MaxBytesReader` sets a flag on `http.ResponseWriter` which instructs our server to closes the underlying TCP connection when the handler returns.

7.2. Displaying Validation Failures

Now that our `CreateSnippet` function is processing the form and validating the data the next step is to manage any validation errors gracefully.

Specifically, if there are any validation errors we want to re-display the form, highlighting the fields which failed validation and automatically re-populating the previously submitted data.

First, let's add a new `Form` field to our `HTMLData` struct, and use this to pass the form data from our handlers to the `new.page.html` template, like so:

cmd/web/views.go

```
package main

...

// Add a Form field to the struct.
type HTMLData struct {
    Form     interface{}
    Path     string
    Snippet  *models.Snippet
    Snippets []*models.Snippet
}

...
```

cmd/web/handlers.go

```
package main

...

func (app *App) NewSnippet(w http.ResponseWriter, r *http.Request) {
    // Pass an empty *forms.NewSnippet object to the new.page.html template. Because
    // it's empty, it won't contain any previously submitted data or validation
    // failure messages.
    app.RenderHTML(w, r, "new.page.html", &HTMLData{
        Form: &forms.NewSnippet{},
    })
}

func (app *App) CreateSnippet(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.ClientError(w, http.StatusBadRequest)
        return
    }

    form := &forms.NewSnippet{
        Title:   r.PostForm.Get("title"),
        Content: r.PostForm.Get("content"),
        Expires: r.PostForm.Get("expires"),
    }

    if !form.Valid() {
        // Re-display the new.page.html template passing in the *forms.NewSnippet
        // object (which contains the validation failure messages and previously
        // submitted data).
        app.RenderHTML(w, r, "new.page.html", &HTMLData{Form: form})
        return
    }

    id, err := app.Database.InsertSnippet(form.Title, form.Content, form.Expires)
    if err != nil {
        app.ServerError(w, err)
        return
    }

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

You're probably wondering why we're using the type `interface{}` for the `HTMLData.Form` field, instead of specifying that it should be of the type `*forms.NewSnippet`. The reason is simply flexibility. Setting the type as `interface{}` allows us to pass *any* object in the `HTMLData.Forms` field, which will be useful later in the book when we want to also use it for user signup and login forms. The downside is that we lose type-safety; there's more chance of errors not being picked up at compile time.

Now that our `*forms.NewSnippet` struct is being passed to our template, all that's left to do is render the validation failure messages and previously submitted data that it contains.

We already know from earlier in the book that you can access exported struct fields by postfixing dot with the field name... so we can retrieve the previously submitted value for the `title` field with `{{.Form.Title}}`.

Dot also works in a similar way with string-keyed maps. You can access the value for a given key by postfixing dot with the key name. So, because the underlying type of the `*forms.NewSnippet.Failures` field is `map[string]string` we can retrieve any failure message for the `title` field with `{{.Form.Failures.Title}}`. Note that map key names *don't* have to be capitalized to access them from a template; I've only used capitalized keys for consistency.

With that in mind, let's update the `ui/html/new.page.html` file to display the data and validation failure messages for each field, if they exist:

ui/html/new.page.html

```
{{define "page-title"}}Add a New Snippet{{end}}

{{define "page-body"}}
<form action="/snippet/new" method="POST">
    {{with .Form}}
        <div>
            <label>Title:</label>
            {{with .Failures.Title}}
                <label class="error">{{.}}</label>
            {{end}}
            <input type="text" name="title" value="{{.Title}}">
        </div>
        <div>
            <label>Content:</label>
            {{with .Failures.Content}}
                <label class="error">{{.}}</label>
            {{end}}
            <textarea name="content">{{.Content}}</textarea>
        </div>
        <div>
            <label>Delete in:</label>
            {{with .Failures.Expires}}
                <label class="error">{{.}}</label>
            {{end}}
            {{$expires := or .Expires "31536000"}}
            <input type="radio" name="expires" value="31536000" {{if (eq $expires "31536000")}}checked{{end}}> One Year
            <input type="radio" name="expires" value="86400" {{if (eq $expires "86400")}}checked{{end}}> One Day
            <input type="radio" name="expires" value="3600" {{if (eq $expires "3600")}}checked{{end}}> One Hour
        </div>
        <div>
            <input type="submit" value="Publish snippet">
        </div>
    {{end}}
</form>
{{end}}
```

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

The only new thing really worth talking about is the line:

```
{{$expires := or .Expires "31536000"}}
```

This is essentially creating a new `$expires` template variable which uses the `or` template function to set the

variable to the value yielded by `.Expires` **or** if that's empty then the default value of `"31536000"` instead.

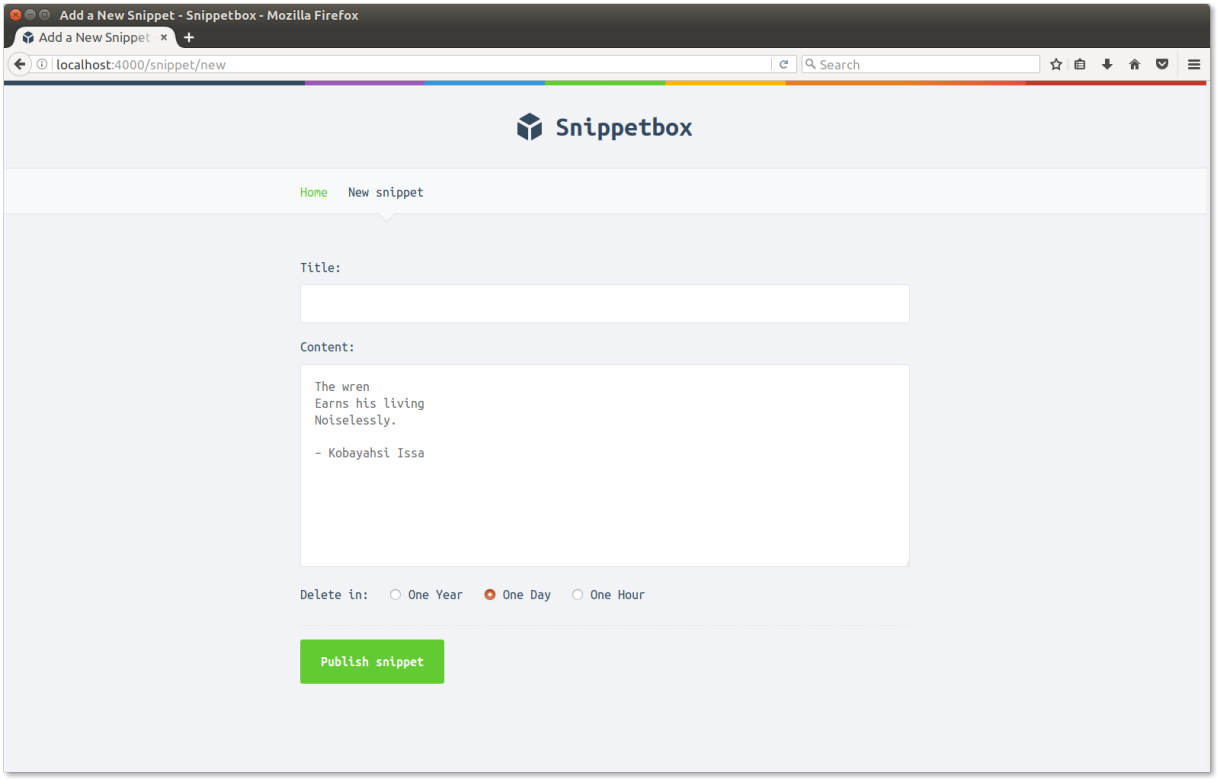
We then use this variable in conjunction with the `eq` function to add the `checked` attribute to the appropriate radio button, like so:

```
{{if (eq $expires "31536000")}}checked{{end}}
```

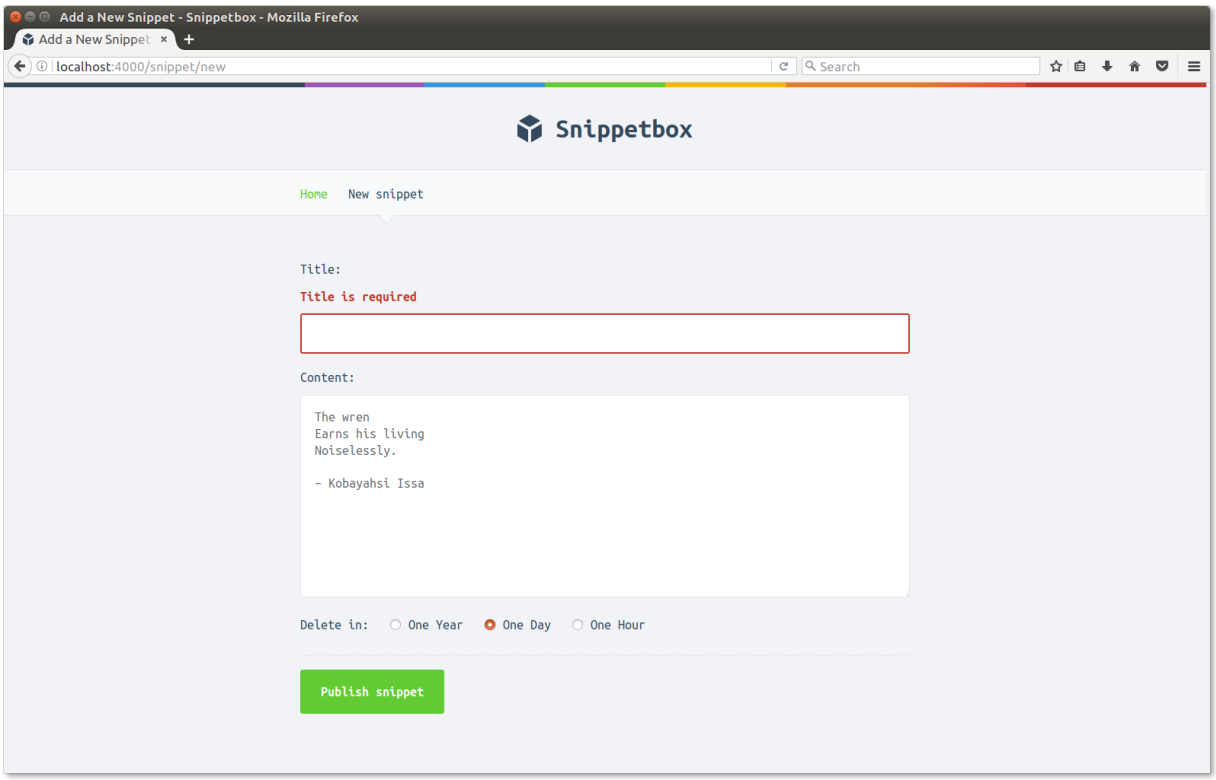
Notice here how we've used `( )` parentheses to group the `eq` function and it's parameters in order to pass its output to the `if` action?

Anyway, restart the application and visit `http://localhost:4000/snippet/new` in your browser.

Try adding some content and changing the default expiry time, but *leave the title field blank* like so:



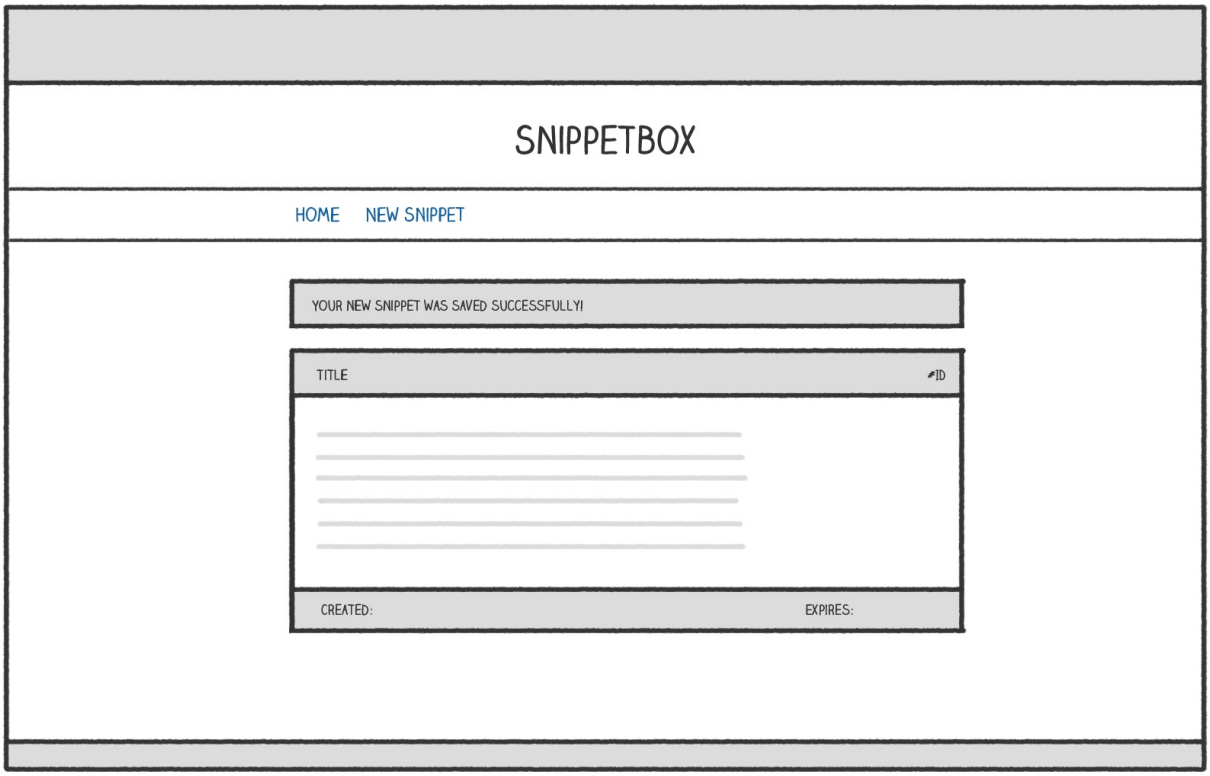
You should now see the form re-displayed, with correctly re-populated content and expiry option and a "Title is required" error message alongside the appropriate field:



Feel free to spend some time playing around with this until you're confident that it's working as you expect it to.

8. Stateful HTTP

A nice touch to improve our application's user experience would be to display a one-time confirmation message which the user sees *after* they've added a new snippet. Like so:



The confirmation message should only show up for the user once (immediately after adding 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 requests from the same user. The most common way to do that is to implement a **session** for the user.

In this section you'll learn:

- What session managers are available to help us implement sessions in Go.
- How to use sessions to safely and securely share data between requests for a particular user.
- How you can customize session behavior (including timeouts and cookie settings) based on your application's needs.

8.1. Choosing a Session Manager

There's a lot of security considerations when it comes to working with sessions, and proper implementation is non-trivial. So 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 only a couple of good session management packages for Go. The two main ones (which aren't framework-specific) are Gorilla Sessions and SCS.

- Gorilla Sessions is the more established and well-known package. It has a simple and easy-to-use API and supports huge range of third-party session stores. Importantly (and unfortunately) if you're using one of the third-party stores Gorilla Sessions doesn't provide a mechanism to renew session tokens, which means that it's vulnerable to session fixation attacks.

- SCS is newer and more streamlined. It supports a smaller range of session stores, but doesn't suffer from the same security issues as Gorilla Sessions. It also provides a nice interface for type-safe manipulation of session data and has a smaller memory footprint. For these reasons, we'll use SCS in the rest of our application build.

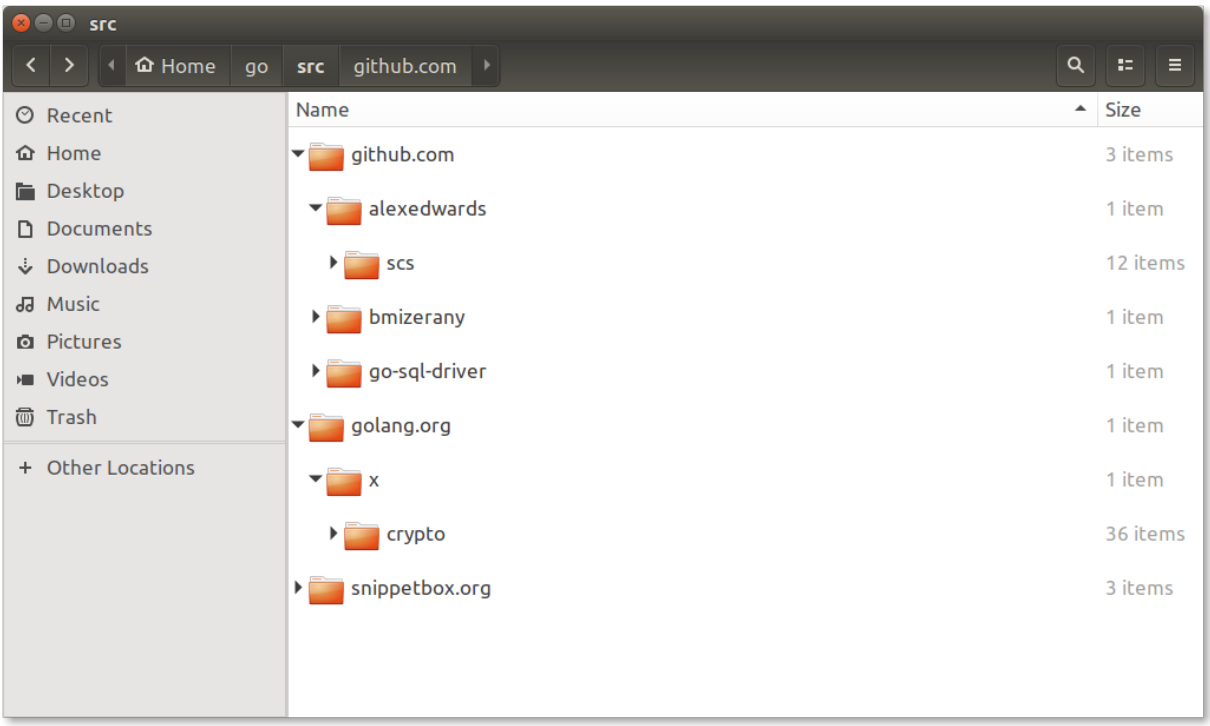
Installing SCS

If you're following along you can use `go get` to install the `scs` package on your computer.

```
$ go get github.com/alexedwards/scs
```

Behind the scenes SCS uses the `nacl/secretbox` package for encryption, so you'll need to install that too:

```
$ go get golang.org/x/crypto/nacl/secretbox
```


src
Home
go
src
github.com
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
Name
Size
github.com
3 items
alexedwards
1 item
scs
12 items
bmizerany
1 item
go-sql-driver
1 item
golang.org
1 item
x
1 item
crypto
36 items
snippetbox.org
3 items

8.2. Working with Session Data

In this chapter I'll run through the fundamentals of setting up and using SCS, but if you're going to use it a production application I recommend reading the documentation to familiarize yourself with the full range of features.

SCS supports a range of session stores but we'll use the default cookie store – which stores session data in an encrypted, authenticated cookie. Using cookies to store session data is fast and easy to setup, but the amount of information you can store is limited (to 4KB). For our application though that will be sufficient.

Establishing a Session Manager

The first thing we need to do establish a **session manager** in our `main.go` file. This essentially holds the configuration settings for our sessions and handles the loading and saving of session data.

We also want to add the session manager to our `App` object, so that our handlers can access it.

I'll demonstrate:

cmd/web/main.go

```
package main

import (
    "database/sql"
    "flag"
    "log"
    "net/http"
    "time" // New import

    "snippetbox.org/pkg/models"

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

func main() {
    // Define a new command-line flag for the session secret (a random key which
    // will be used to encrypt and authenticate session cookies). It should be 32
    // characters long.
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "sb:pass@/snippetbox?parseTime=true", "MySQL DSN")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    secret := flag.String("secret", "s6Nd%+pPbnzHbS*+9Pk8qGWhTzbpa@ge", "Secret key")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")

    flag.Parse()

    db := connect(*dsn)
    defer db.Close()

    // Use the scs.NewCookieManager() function to initialize a new session manager,
    // passing in the secret key as the parameter. Then we configure it so the
    // session always expires after 12 hours and sessions are persisted across
    // browser restarts.
    sessionManager := scs.NewCookieManager(*secret)
    sessionManager.Lifetime(12 * time.Hour)
    sessionManager.Persist(true)

    // Add the session manager to our application dependencies.
    app := &App{
        Database:  &models.Database{db},
        HTMLDir:   *htmlDir,
        Sessions:  sessionManager,
        StaticDir: *staticDir,
    }

    log.Printf("Starting server on %s", *addr)
    err := http.ListenAndServe(*addr, app.Routes())
    log.Fatal(err)
}

...
```

cmd/web/app.go

```
package main

import (
    "snippetbox.org/pkg/models"

    "github.com/alexedwards/scs" // New import
)

// Add a new Sessions field to our application dependencies.
type App struct {
    Database  *models.Database
    HTMLDir   string
    Sessions  *scs.Manager
    StaticDir string
}
```

In the code above we've used the `Lifetime()` and `Persist()` methods to configugure the behavior of our session manager, but there's a range of other methods which you can and should change depending on your application's needs.

Storing and Retrieving Data

Now we're ready to use the session manager in our handler functions. Let's do exactly that and update our `CreateSnippet` handler so it adds a flash message to the session data.

cmd/web/handlers.go

```
package main

...

func (app *App) CreateSnippet(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.ClientError(w, http.StatusBadRequest)
        return
    }

    form := &forms.NewSnippet{
        Title:   r.PostForm.Get("title"),
        Content: r.PostForm.Get("content"),
        Expires: r.PostForm.Get("expires"),
    }

    if !form.Valid() {
        app.RenderHTML(w, r, "new.page.html", &HTMLData{Form: form})
        return
    }

    id, err := app.Database.InsertSnippet(form.Title, form.Content, form.Expires)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // Use session manager's Load() method to fetch the session data for the current
    // request. If there's no existing session for the current user (or their
    // session has expired) then a new, empty, session will be created. Any errors
    // are deferred until the session is actually used.
    session := app.Sessions.Load(r)

    // Use the PutString() method to add a string value ("Your snippet was saved
    // successfully!") and the corresponding key ("flash") to the the session
    // data.
    err = session.PutString(w, "flash", "Your snippet was saved successfully!")
    if err != nil {
        app.ServerError(w, err)
        return
    }

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

We've used the `session.PutString()` method in the code above, but there are a variety of other methods for getting and setting common data types.

Next up we want the `ShowSnippet` handler to check if the current session data contains the key `"flash"`. If it does, we want to pass the value to our `show.page.html` template so that the message can be displayed.

cmd/web/handlers.go

```
package main

...

func (app *App) 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
    }

    snippet, err := app.Database.GetSnippet(id)
    if err != nil {
        app.ServerError(w, err)
        return
    }
    if snippet == nil {
        app.NotFound(w)
        return
    }

    // Load the session data then 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 it will return the empty string. If you want to
    // retrieve a string from the session and not delete it you should use the
    // GetString() method instead.
    session := app.Sessions.Load(r)
    flash, err := session.PopString(w, "flash")
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // Pass the flash message to the template.
    app.RenderHTML(w, r, "show.page.html", &HTMLData{
        Flash:   flash,
        Snippet: snippet,
    })
}

...
```

If you try to run the application now, the compiler will (rightly) grumble that the `Flash` field isn't defined in our `HTMLData` struct. Go ahead and add it in like so:

cmd/web/views.go

```
package main

...

// Add a Flash field to the struct.
type HTMLData struct {
    Flash    string
    Form     interface{}
    Path     string
    Snippet  *models.Snippet
    Snippets []*models.Snippet
}

...
```

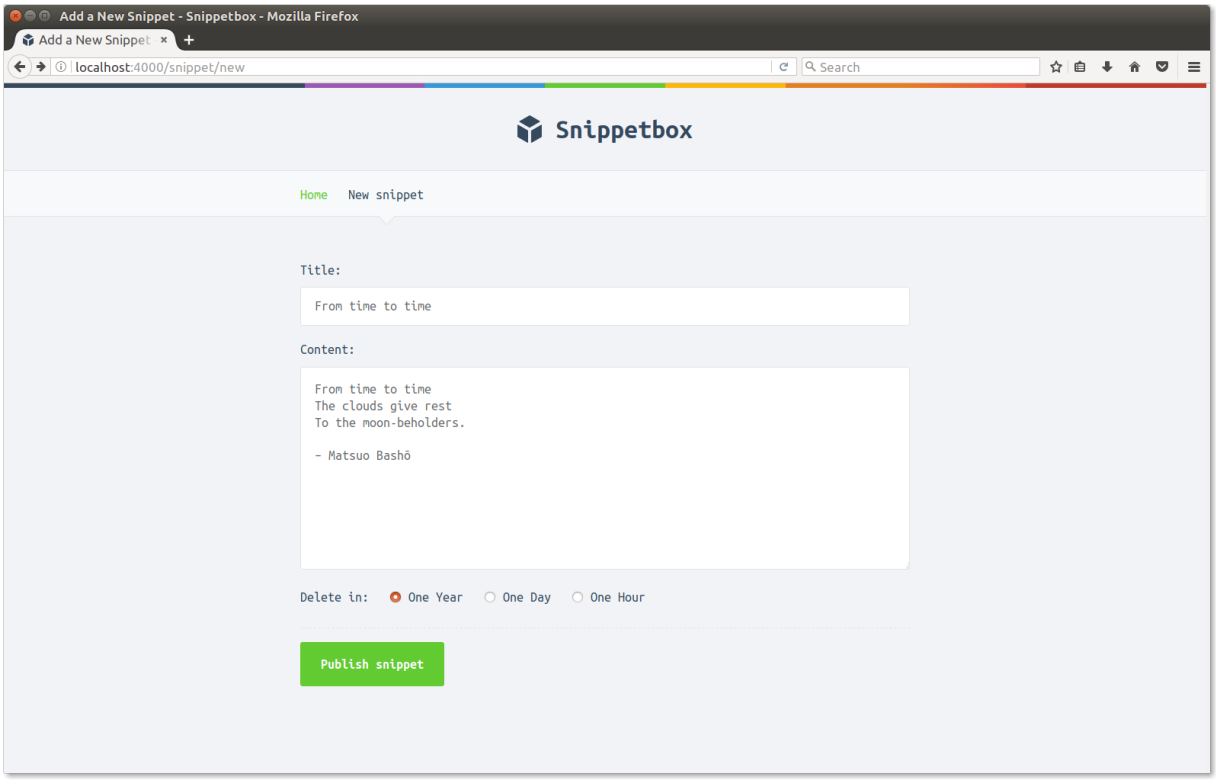
Now – at last – we can update our `ui/html/show.page.html` to display the flash message, if one exists.

ui/html/show.page.html

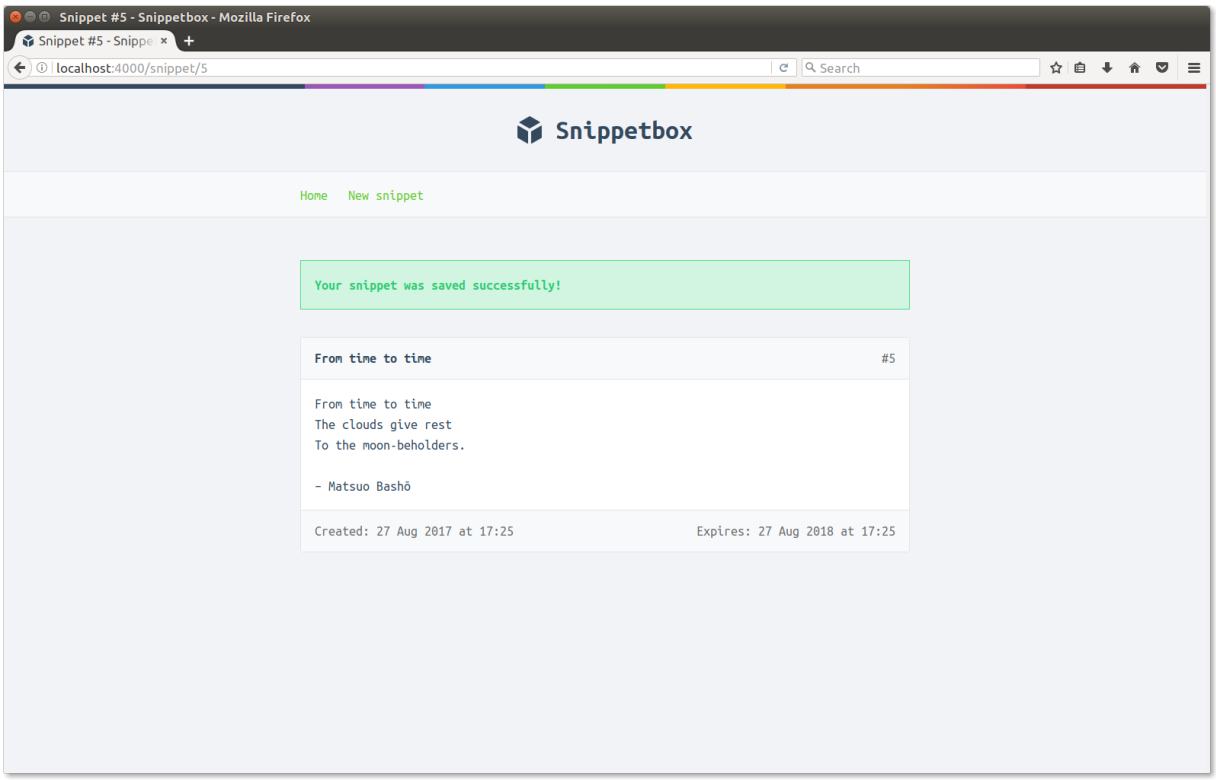
```
{{define "page-title"}}Snippet #{{.Snippet.ID}}{{end}}

{{define "page-body"}}
    {{with .Flash}}
    <div class="flash">{{.}}</div>
    {{end}}
    {{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: {{humanDate .Created}}</time>
                <time>Expires: {{humanDate .Expires}}</time>
            </div>
        </div>
    {{end}}
{{end}}
```

Remember, the `{{with .Flash}}` block will only be evaluated if the value of `.Flash` is not empty. So the end result is that if there's no `"flash"` key in the session, then that block of markup simply won't be displayed.

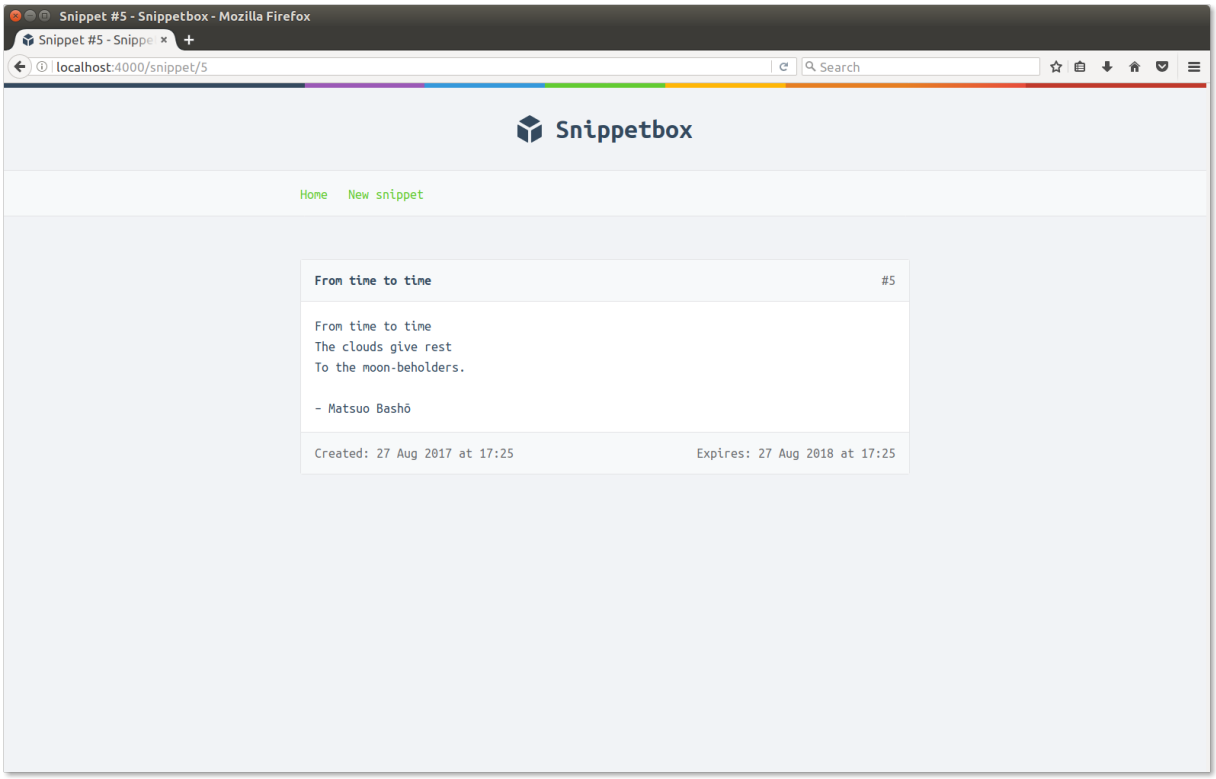
Once that's done, save all your files and restart the application. Try adding a new snippet like so...



And after redirection you should see the flash message now being displayed:



If you try refreshing the page, you can confirm that the flash message is no longer shown – it was a one-off message for the current user immediately after they created the snippet.



9. 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 doing some heavy processing.

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 third-party packages.
- How to create middleware which logs the requests received by your application.
- How to set default headers on each response to improve security for our users and application.

9.1. Logging Requests

As I explained earlier in the book, You can think of a Go web application as essentially being a chain of `ServeHTTP()` methods being called one after another. In our application, our server receives a request and then calls the `pat` router's `ServeHTTP()` method. The router looks up the relevant handler for the request method and URL, and in turn calls the handler's `ServeHTTP()`method.

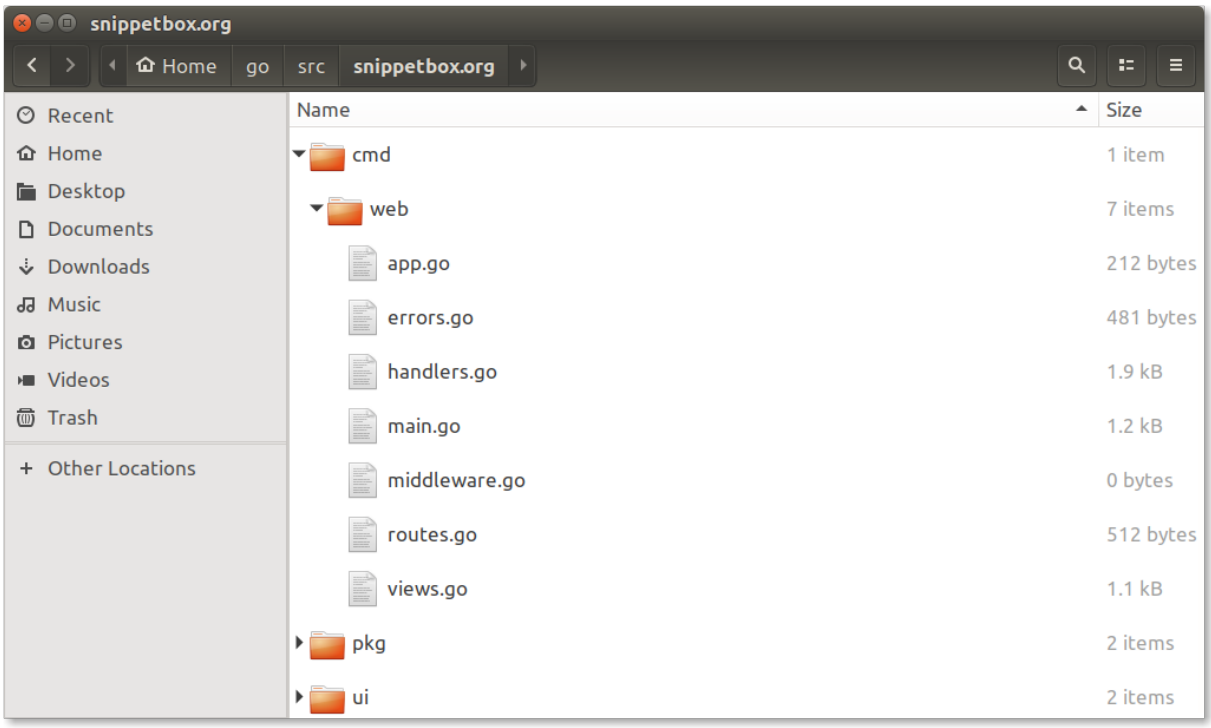
The basic idea of middleware is to insert another handler in this chain. This middleware handler executes some logic, and then calls the `ServeHTTP()` method of the *next* handler.

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.

In this chapter we'll create some `LogRequest` middleware which logs the HTTP requests that our application receives. In particular, we want to log the IP address of the user, and which URL and method are being requested.

The best way to explain the pattern for creating middleware is to demonstrate it and then talk it through. So if you're following along go ahead and create a new `cmd/web/middleware.go` file:

```
$ cd $HOME/go/src/snippetbox.org
$ touch cmd/web/middleware.go
```



And then add the following code:

cmd/web/middleware.go

```
package main

import (
    "log"
    "net/http"
)

func LogRequest(next http.Handler) http.Handler {
    fn := func(w http.ResponseWriter, r *http.Request) {
        pattern := `%s - "%s %s %s"`
        log.Printf(pattern, r.RemoteAddr, r.Proto, r.Method, r.URL.RequestURI())

        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 `LogRequest` function is essentially a wrapper around the `next` handler.
- It establishes an function `fn` which *closes over* the `next` handler to form a closure. When `fn` is run it executes our middleware logic (i.e. it logs the request) 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 that's confusing, you can think of it simply like this: Our `LogRequest` middleware is a function that accepts the next handler in a chain as a parameter. It executes some logic (here, logging the request) and then calls the next handler.

Positioning the Middleware

Because we want the middleware to log every request that is received, it makes sense for the middleware to act *before* a request hits our router. So we want the flow of control through our application to look like:

```
LogRequest → Router → Application Handler
```

So in essence, we want our router to be the `next` parameter and have our `LogRequest` middleware wrap our router. Let's update the `routes.go` file to do exactly that:

cmd/web/routes.go

```
package main

...

func (app *App) Routes() http.Handler {
    mux := pat.New()
    mux.Get("/", http.HandlerFunc(app.Home))
    mux.Get("/snippet/new", http.HandlerFunc(app.NewSnippet))
    mux.Post("/snippet/new", http.HandlerFunc(app.CreateSnippet))
    mux.Get("/snippet/:id", http.HandlerFunc(app.ShowSnippet))

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

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

Go ahead and give this a try. Run the application then play around making some requests. When you check the log in your terminal you should see something like this:

```
$ go run cmd/web/*
2017/08/30 14:10:05 Starting server on :4000
2017/08/30 14:10:53 127.0.0.1:45022 - "HTTP/1.1 GET /"
2017/08/30 14:10:53 127.0.0.1:45022 - "HTTP/1.1 GET /static/css/main.css"
2017/08/30 14:10:53 127.0.0.1:45022 - "HTTP/1.1 GET /static/img/logo.png"
2017/08/30 14:10:56 127.0.0.1:45022 - "HTTP/1.1 GET /snippet/new"
2017/08/30 14:11:02 127.0.0.1:45022 - "HTTP/1.1 POST /snippet/new"
2017/08/30 14:11:03 127.0.0.1:45022 - "HTTP/1.1 GET /snippet/7"
```

Simplifying the Middleware

We can also simplify our `LogRequest` middleware slightly by making use of an anonymous function, like so:

cmd/web/middleware.go

```
package main

...

func LogRequest(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        pattern := `%s - "%s %s %s"`
        log.Printf(pattern, r.RemoteAddr, r.Proto, r.Method, r.URL.RequestURI())

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

This pattern is very common in the wild, and one that you'll often see if you're reading the source code of other applications or third-party packages.

Notes

Flow of Control

It's important to know that when the last handler in the chain returns, control is passes back up the chain in the reverse direction. So when our code is being executed the flow of control actually looks like:

```
LogRequest → Router → Application Handler → Router → LogRequest
```

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.
    })
}
```

Another thing to mention is returning from middleware. Within your middleware, if you call `return` *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.

9.2. Adding Headers

In this chapter we'll look at how to make some middleware which automatically adds the following HTTP headers to every response that our application sends:

```
X-Content-Type-Options: nosniff
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.

Working with Response Headers

This will be the first time in our application build we've done anything with response headers, so I'll quickly explain the basics.

You can control which headers are set on a response via the `w.Header()` method. This returns a `http.Header` object, which is essentially a map holding the response headers that will be written to the connection.

You can change the contents of the map with the `Add()`, `Del()` and `Set()` methods. For example:

```
Set a new cache-control header. If an existing "Cache-Control" header exists
```

```
// it will be overwritten.
w.Header().Set("Cache-Control", "no-cache")

// In contrast, the Add() appends a new value to the "Cache-Control" header.
w.Header().Add("Cache-Control", "must-revalidate")

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

The contents of the header map will be written to the underlying connection the first time that `w.WriteHeader()` or `w.Write()` is called. Changing the map after that will have no effect.

One nuance to note is that the `Set()` and `Add()` methods always canonicalize the header name using the `http.CanonicalHeaderKey()` function. This converts the first letter and any letter following a hyphen to upper case, and the rest of the letters to lowercase.

This can cause problems with (valid) headers names like `"WWW-Authenticate"` and `"X-XSS-Protection"`. To avoid the canonicalization behavior you'll need to edit the underlying header map directly. For example:

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

SecureHeaders Middleware

Let's put this together and create some middleware that adds the security headers from the start of the chapter to every response.

Open your `middleware.go` file and let's create a `SecureHeaders` function with exactly the same pattern that we used in the previous chapter:

cmd/web/middleware.go

```
package main

...

func SecureHeaders(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("X-Content-Type-Options", "nosniff")
        w.Header().Set("X-Frame-Options", "deny")
        w.Header()["X-XSS-Protection"] = []string{"1; mode=block"}

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

And then let's update our `routes.go` file so that the flow of control looks like this:

```
LogRequest ↔ SecureHeaders ↔ Router ↔ Application Handler
```

cmd/web/routes.go

```
package main

...

func (app *App) Routes() http.Handler {
    mux := pat.New()
    mux.Get("/", http.HandlerFunc(app.Home))
    mux.Get("/snippet/new", http.HandlerFunc(app.NewSnippet))
    mux.Post("/snippet/new", http.HandlerFunc(app.CreateSnippet))
    mux.Get("/snippet/:id", http.HandlerFunc(app.ShowSnippet))

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

    // Notice how we are able to build up the chain of middleware though nesting?
    return LogRequest(SecureHeaders(mux))
}
```

Go ahead and give this a try. If you run the application now, and make any kind of request, you should see that the security headers are now included in the response.

```
$ curl -I http://localhost:4000/
HTTP/1.1 200 OK
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
Date: Wed, 30 Aug 2017 18:30:28 GMT
Content-Length: 1408
Content-Type: text/html; charset=utf-8
```

Notes

Deleting System-Generated Headers

You can't use the `w.Header().Del()` method to remove automatically-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
```

10. Security Improvements

In the next 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.

In particular 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 which will help keep user information secure and our server performing quickly.
- How to set timeouts on our server to mitigate Slowloris and other slow-client attacks.

10.1. Running a HTTPS Server

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 (wrongly) used interoperably with TLS. For clarity and accuracy, we'll stick with the term TLS throughout this book.

Generating a Self-Signed Certificate

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 our own **self-signed certificate**.

A self-signed certificate is exactly 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 this first time it's used, but it will nonetheless encrypt HTTPS traffic correctly and is fine for development and testing purposes.

Handily, Go's ships with a `generate_cert.go` tool for creating self-signed TLS certificates. Let's use this to make a certificate for our development server.

If you're following along, create a new `tls` directory in the root of your project repository to hold the certificate...

```
$ cd $HOME/go/src/snippetbox.org
$ mkdir tls
```

And then run the `generate_cert.go` tool using the following command:

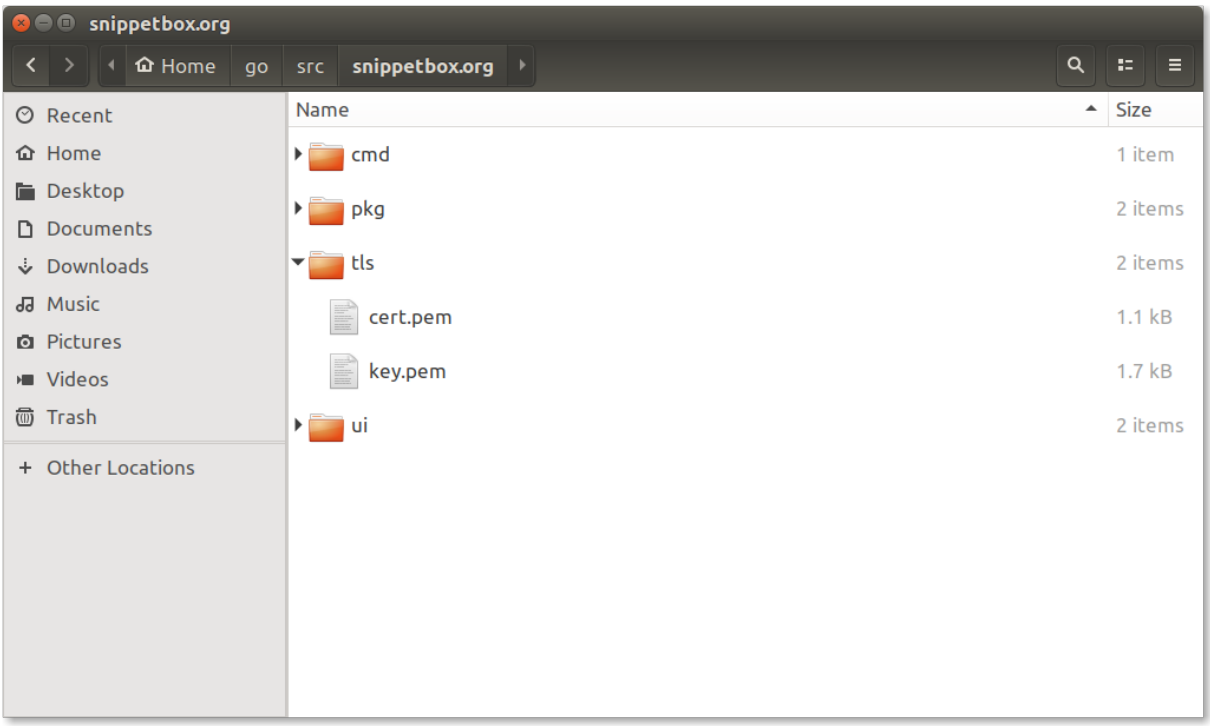
```
$ cd tls
$ go run /usr/local/go/src/crypto/tls/generate_cert.go --rsa-bits=2048 --host=localhost
2017/08/31 12:01:56 written cert.pem
2017/08/31 12:01:56 written key.pem
```

Behind the scenes, the `generate_cert.go` tool works in two stages.

First it generates a 2048-bit RSA key pair, which is essentially a cryptographically secure public key and private key.

It then stores the private key in a `key.pem` file, and generates a self-signed TLS certificate for the host `localhost` containing the public key – which it stores in a `cert.pem` file. Both the private key and certificate are PEM encoded, which is the standard format used by most TLS implementations.

Your project repository should now look something like this:



And that's it! We've now got a self-signed TLS certificate (and corresponding private key) that we can use during development.

Starting 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 to swap our `http.ListenAndServe()` function for `http.ListenAndServeTLS()` instead.

Alter your `main.go` file to match the following code:

cmd/web/main.go

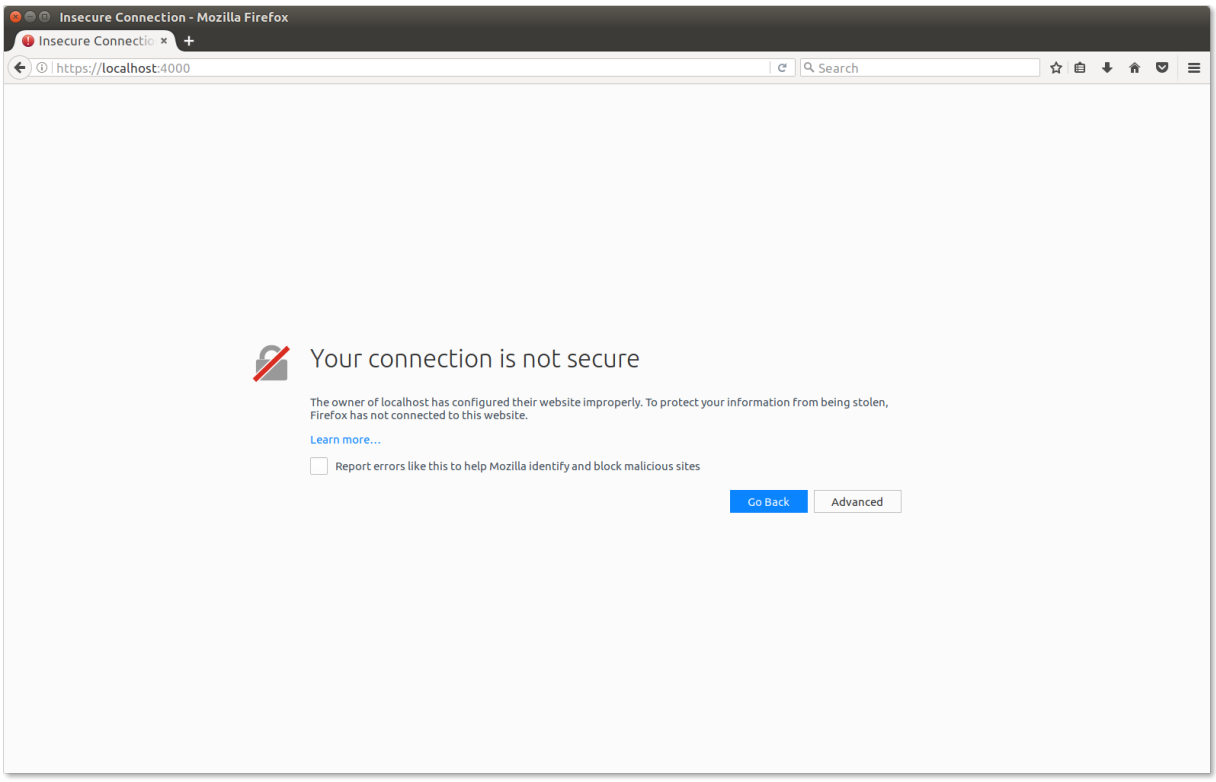
```
package main

...

func main() {
    // Define new command-line flags for the TLS certificate and private key
    // locations. I've set the defaults to point to the self-signed files we've
    // just created.
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "sb:pass@/snippetbox?parseTime=true", "MySQL DSN")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    secret := flag.String("secret", "sb04y4ER5irMeOppyf5qdJG9kQSjWw2F", "Secret key")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")
    tlsCert := flag.String("tls-cert", "./tls/cert.pem", "Path to TLS certificate")
    tlsKey := flag.String("tls-key", "./tls/key.pem", "Path to TLS key")

    flag.Parse()

    db := connect(*dsn)
    defer db.Close()

    sessionManager := scs.NewCookieManager(*secret)
    sessionManager.Lifetime(12 * time.Hour)
    sessionManager.Persist(true)
    sessionManager.Secure(true) // Set the Secure flag on our session cookies

    app := &App{
        Database:  &models.Database{db},
        HTMLDir:   *htmlDir,
        Sessions:  sessionManager,
        StaticDir: *staticDir,
    }

    log.Printf("Starting server on %s", *addr)
    // Use the http.ListenAndServeTLS() function to start the HTTPS server. We
    // pass in the paths to the TLS certificate and corresponding private key as
    // the 2nd and 3rd parameters respectively.
    err := http.ListenAndServeTLS(*addr, *tlsCert, *tlsKey, app.Routes())
    log.Fatal(err)
}

...
```

When we run this, our server will still be listening on port 4000 – the only difference is that it will be talking HTTPS instead of HTTP.

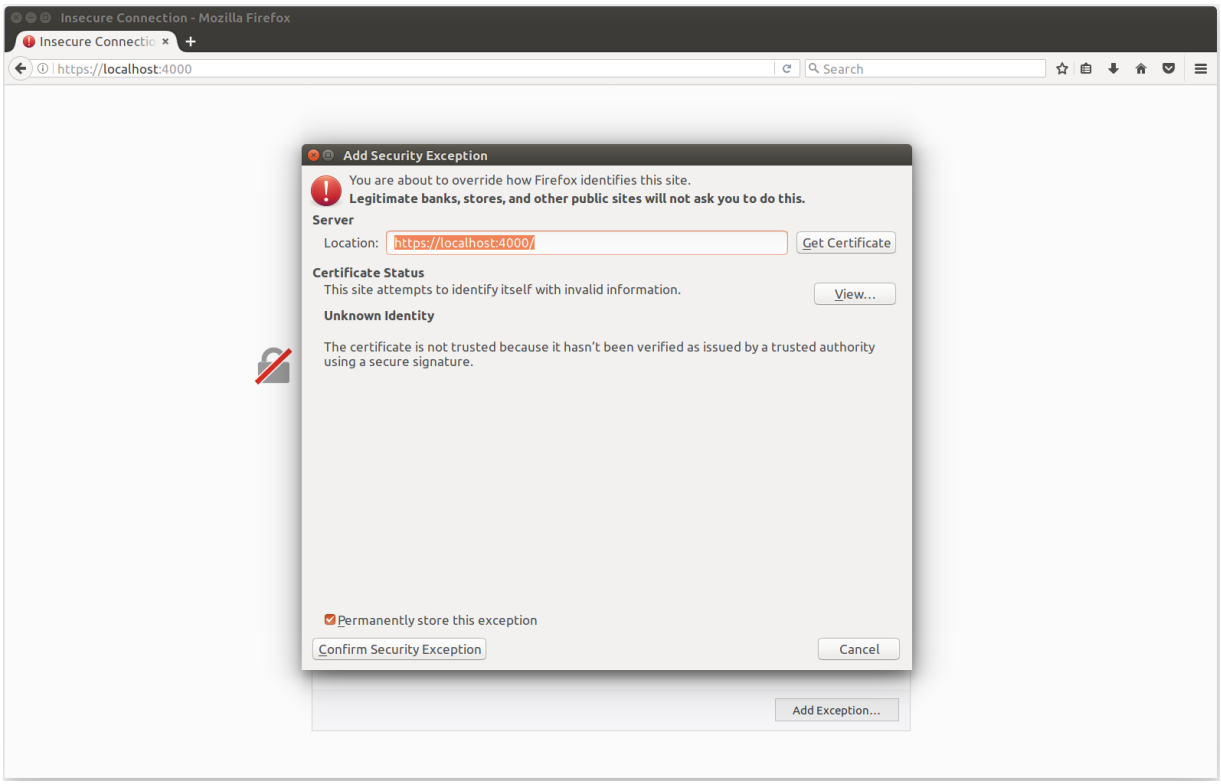
Go ahead and run it as normal:

```
$ cd $HOME/go/src/snippetbox.org/
$ go run cmd/web/*
2017/08/31 12:14:36 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.



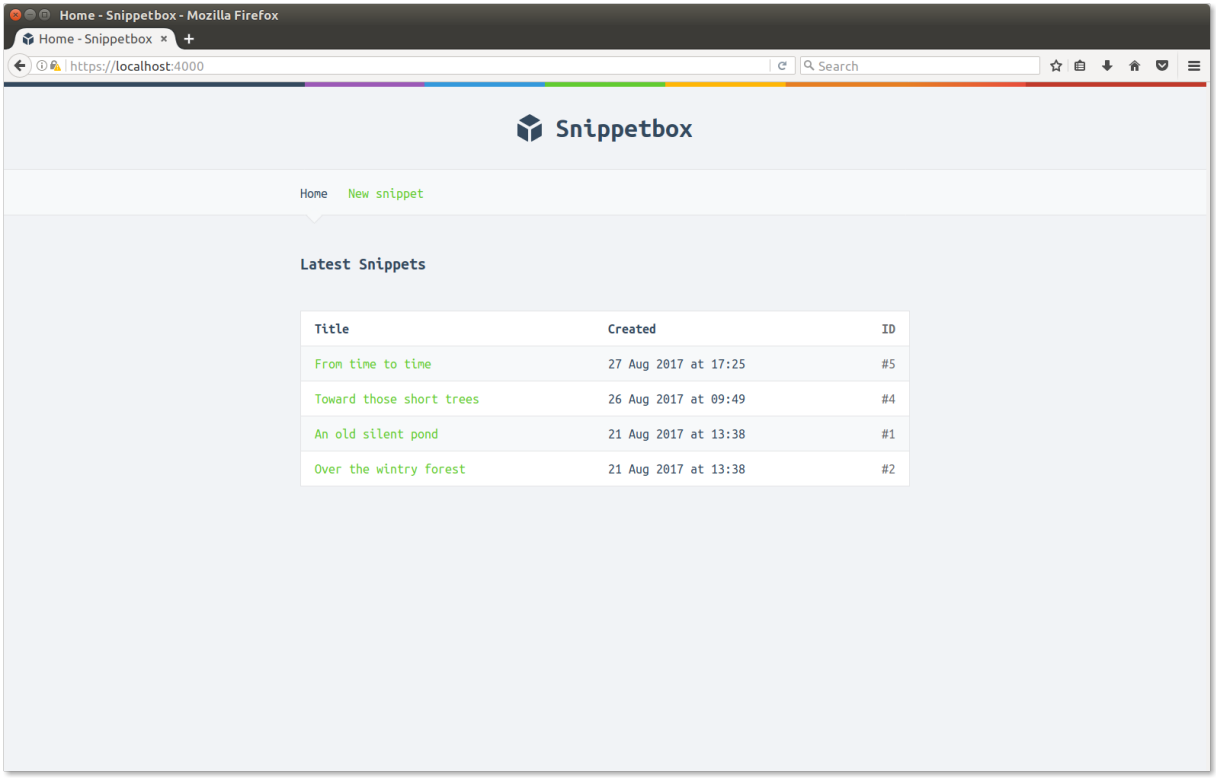
If you're using Firefox like me, click "Advanced" then "Add Exception", and in the dialog box that appears click "Confirm Security Exception".



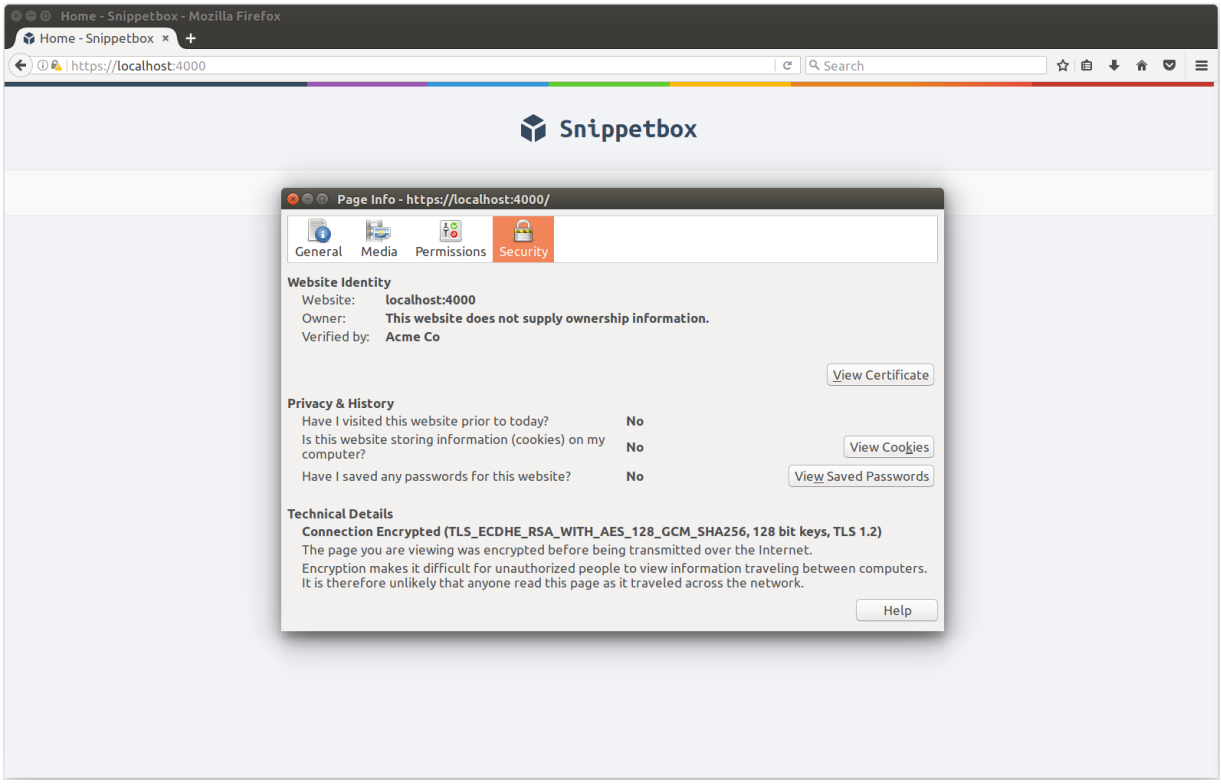
If you're using Chrome or Chromium, click "Advanced" and then the "Proceed to localhost" link.

After that our application homepage should appear (though it will still carry a warning in the URL bar because our TLS certificate is self-signed).

In Firefox, it should look a bit like this:



If you're using Firefox I also recommend pressing `Ctrl+i` to take a look at the Page Info for your homepage:



The technical details section of this 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.

(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.)

Notes

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 the terminal relating to this.

```
$ go run cmd/web/*
2017/08/31 12:27:55 Starting server on :4000
2017/08/31 12:28:11 http: TLS handshake error from 127.0.0.1:36260: tls: first record does not look like a TLS handshake
```

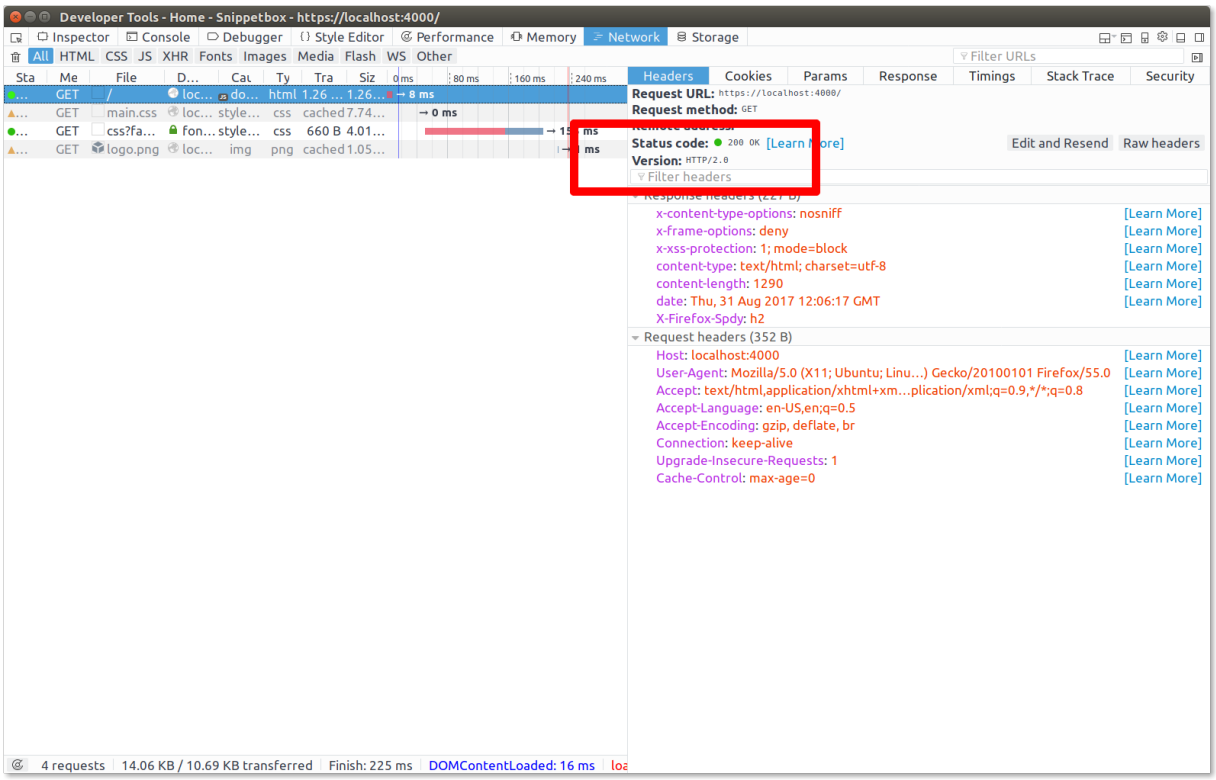
HTTP2

A big plus of using HTTPS is that Go's HTTPS server implementation supports HTTP/2 connections.

If a client supports HTTP/2, our server will now automatically and transparently upgrade connections to 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.

Again, 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. If you take a look at the headers for the homepage you should see that the protocol being used is HTTP/2.



Permissions

It's important to note that the user which you're using to run your Go application must have read permissions for both the `cert.pem` and `key.pem` files, otherwise `http.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:

```
$ ls -l tls/
total 16
-rw-r--r-- 1 alex alex 1094 Aug 31 12:01 cert.pem
-rw------- 1 alex alex 1679 Aug 31 12:01 key.pem
```

Generally, it's a good idea to keep the permissions 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:

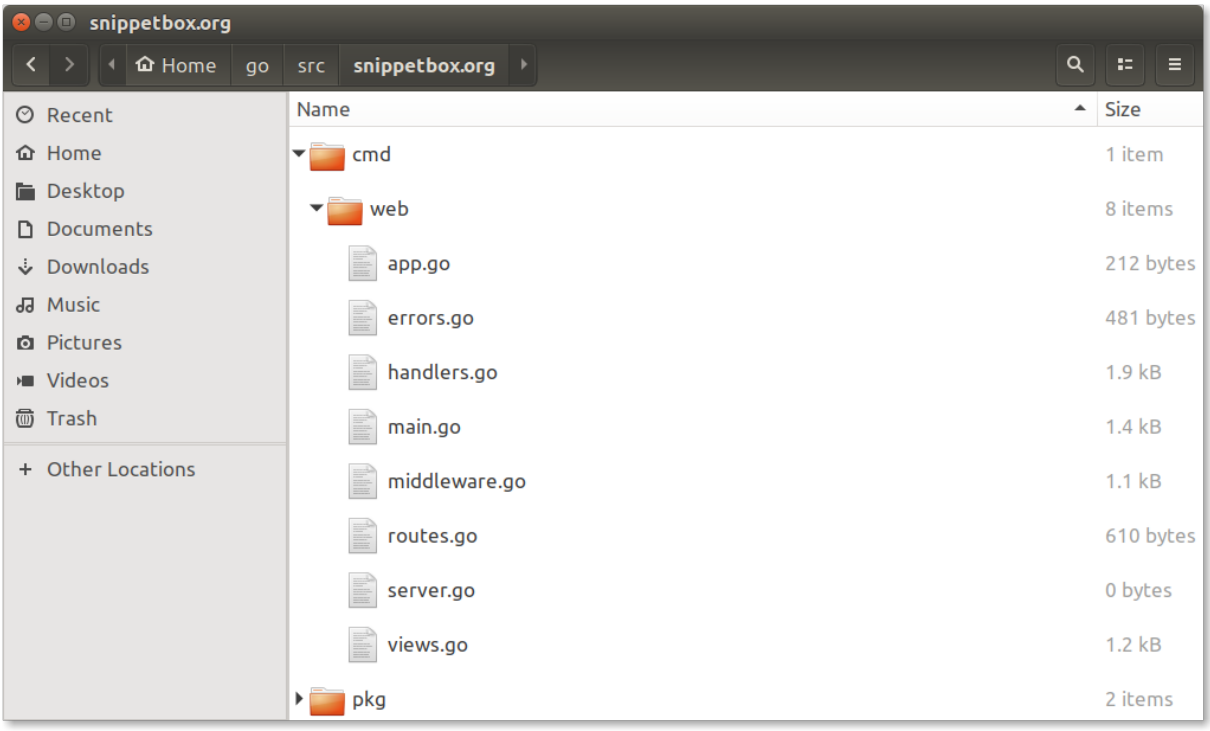
```
$ echo 'tls/' >> .gitignore
```

10.2. 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.

But before we do, let's port the code for starting our HTTPS server into a separate file to prevent our `main()` function from getting too cluttered. Go ahead and create a new `cmd/web/server.go` file:

```
$ cd $HOME/go/src/snippetbox.org
$ touch cmd/web/server.go
```



And add a simple `app.RunServer()` method to control the initialization and running of our HTTPS server, like so:

cmd/web/server.go

```
package main

import (
    "log"
    "net/http"
)

func (app *App) RunServer() {
    log.Printf("Starting server on %s", app.Addr)
    err := http.ListenAndServeTLS(app.Addr, app.TLSCert, app.TLSKey, app.Routes())
    log.Fatal(err)
}
```

This method is pulling in the network address and TLS certificate/key as dependencies from the `App` object, so let's add the necessary `Addr`, `TLSCert` and `TLSKey` fields in the `app.go` file.

cmd/web/app.go

```
package main

...

type App struct {
    Addr      string // Add an Addr field
    Database  *models.Database
    HTMLDir   string
    Sessions  *scs.Manager
    StaticDir string
    TLSCert   string // Add a TLSCert field
    TLSKey    string // Add a TLSKey field
}
```

And then let's hook it together in the `main.go` file:

cmd/web/main.go

```
package main

import (
    "database/sql"
    "flag"
    "log"
    "time"

    "snippetbox.org/pkg/models"

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

func main() {
    addr := flag.String("addr", ":4000", "HTTP network address")
    dsn := flag.String("dsn", "sb:pass@/snippetbox?parseTime=true", "MySQL DSN")
    htmlDir := flag.String("html-dir", "./ui/html", "Path to HTML templates")
    secret := flag.String("secret", "sb04y4ER5irMeOppyf5qdJG9kQSjWw2F", "Secret key")
    staticDir := flag.String("static-dir", "./ui/static", "Path to static assets")
    tlsCert := flag.String("tls-cert", "./tls/cert.pem", "Path to TLS certificate")
    tlsKey := flag.String("tls-key", "./tls/key.pem", "Path to TLS key")

    flag.Parse()

    db := connect(*dsn)
    defer db.Close()

    sessionManager := scs.NewCookieManager(*secret)
    sessionManager.Lifetime(12 * time.Hour)
    sessionManager.Persist(true)
    sessionManager.Secure(true)

    // Add the network address and TLS certificate/key locations to the dependencies.
    app := &App{
        Addr:      *addr,
        Database:  &models.Database{db},
        HTMLDir:   *htmlDir,
        Sessions:  sessionManager,
        StaticDir: *staticDir,
        TLSCert:   *tlsCert,
        TLSKey:    *tlsKey,
    }

    // Call the new RunServer() method to start the server.
    app.RunServer()
}

...
```

This is a small change, but it gives us a really neat, clean and testable structure for our application. The responsibility of the `main()` function is now limited to parsing the runtime configuration settings for the application and initializing the dependencies, and our web application itself is a single, self-contained `App` object.

Tweaking the TLS settings

If you're new to HTTPS and TLS 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, despite that, there's 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 manually initialize a `http.Server` struct which contains the configuration settings for the server... including the `tls.Config` struct.

I'll demonstrate:

cmd/web/server.go

```
package main

import (
    "crypto/tls" // New import
    "log"
    "net/http"
)

func (app *App) RunServer() {
    // Declare a tls.Config variable to hold the non-default TLS settings we want the
    // server to use.
    tlsConfig := &tls.Config{
        PreferServerCipherSuites: true,
        CurvePreferences:         []tls.CurveID{tls.X25519, tls.CurveP256},
    }

    // Initialize a new http.Server struct. We set the Addr and Handler so that
    // the server uses the same network address and routes as before, and we also set
    // the TLSConfig field to use the tlsConfig variable we just created.
    srv := &http.Server{
        Addr:      app.Addr,
        Handler:   app.Routes(),
        TLSConfig: tlsConfig,
    }

    // Call the http.Server's ListenAndServeTLS() method to start the server,
    // passing in the paths to the TLS certificate and corresponding private key.
    log.Printf("Starting server on %s", app.Addr)
    err := srv.ListenAndServeTLS(app.TLSCert, app.TLSKey)
    log.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, Go's favored cipher suites are given preference and we help increase the likelihood that a strong cipher 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 number of elliptic curves, but as of Go 1.9 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.

Notes

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 or 3DES. You can do this via the `tls.Config.CipherSuites` field like so:

```
tlsConfig := &tls.Config{
    PreferServerCipherSuites: true,
    CipherSuites: []uint16{
        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,
        tls.TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
        tls.TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
        tls.TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256,
        tls.TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256,
        tls.TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,
        tls.TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
        tls.TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA,
        tls.TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,
    },
}
```

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:

```
tlsConfig := &tls.Config{
    MinVersion: tls.VersionTLS12,
    MaxVersion: tls.VersionTLS12,
}
```

10.3. Connection Timeouts

At the moment our server is vulnerable to slow-client atacks, such as Slowloris.

If you're not familiar with the principles behind Slowloris, the rough idea that an attacker sends our server many partial, incomplete, HTTP requests. Periodically they send additional headers – adding to but never completing the request – so that the connection is kept open indefinitely. Sooner or later, our server will hit the maximum limit of open connections allowed by the operating system and it will become non-responsive.

We can mitigate this by setting timeouts for our server, so that if the request read (or response write) is not completed in a certain time the connection is automatically closed.

Setting timeouts is pretty straightforward:

cmd/web/server.go

```
package main

import (
    "crypto/tls"
    "log"
    "net/http"
    "time" // New import
)

func (app *App) RunServer() {
    tlsConfig := &tls.Config{
        PreferServerCipherSuites: true,
        CurvePreferences:         []tls.CurveID{tls.X25519, tls.CurveP256},
    }

    // Add Idle, Read and Write timeouts to the server.
    srv := &http.Server{
        Addr:         app.Addr,
        Handler:      app.Routes(),
        TLSConfig:     tlsConfig,
        IdleTimeout:  time.Minute,
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    log.Printf("Starting server on %s", app.Addr)
    err := srv.ListenAndServeTLS(app.TLSCert, app.TLSKey)
    log.Fatal(err)
}
```

All three of these timeouts – `IdleTimeout`, `ReadTimeout` and `WriteTimeout` – are server-wide settings which act on the underlying connection. They 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 `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.

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 that 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 that 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 important to set `WriteTimeout` to a greater value that `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.

Notes

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 is for the read of the HTTP(S) headers only. So, if we set `ReadHeaderTimout` to 3 seconds the 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 a `http.TimeoutHandler`).

For the Snippetbox web application we don't have any actions which 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`.

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 the request header. By default Go allows a maximum header length of 1MB.

If you want to limit the maximum header length to 0.5MB, for example, you would write:

```
srv := &http.Server{
    Addr:           *addr,
    MaxHeaderBytes: 524288,
    ...
}
```

If `MaxHeaderBytes` is exceeded then the user will receive a `431 Request Header Fields Too Large` response.

There's a gotcha to point out here: Go *always* adds an additional 4096 bytes 'slop' or 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,

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 are able to add new snippets.

Unregistered users will still be able to view the snippets, and will also be able to sign up for an account through the application.

A lot of the content in this section is actually 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.

But that said, the patterns themselves are new. In particular you'll learn:

- A secure approach to encrypting and storing user passwords securely in your database using Bcrypt.
- How to implement basic signup, login and logout functionality for users.
- 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.

11.1. Routes and Database Setup

Let's begin this section by adding five new routes to our application, so it looks like this:

Method	URL Path	Handler	Action
GET	/	Home	Display the homepage
GET	/snippet/:id	ShowSnippet	Display a specific snippet
GET	/snippet/new	NewSnippet	Display the new snippet form
POST	/snippet/new	CreateSnippet	Create a new snippet
GET	/user/signup	SignupUser	Display the user signup form
POST	/user/signup	CreateUser	Create a new user
GET	/user/login	LoginUser	Display the user login form
POST	/user/login	VerifyUser	Verify the user credentials
POST	/user/logout	LogoutUser	Logout the user
GET	/static/...	http.FileServer	Serve a specific static file

Notice how the state-changing actions - `CreateUser`, `VerifyUser` and `LogoutUser` are all using `POST` requests, not `GET`?

Open up your `handlers.go` file and add placeholders for the five new handler functions like so:

cmd/web/handlers.go

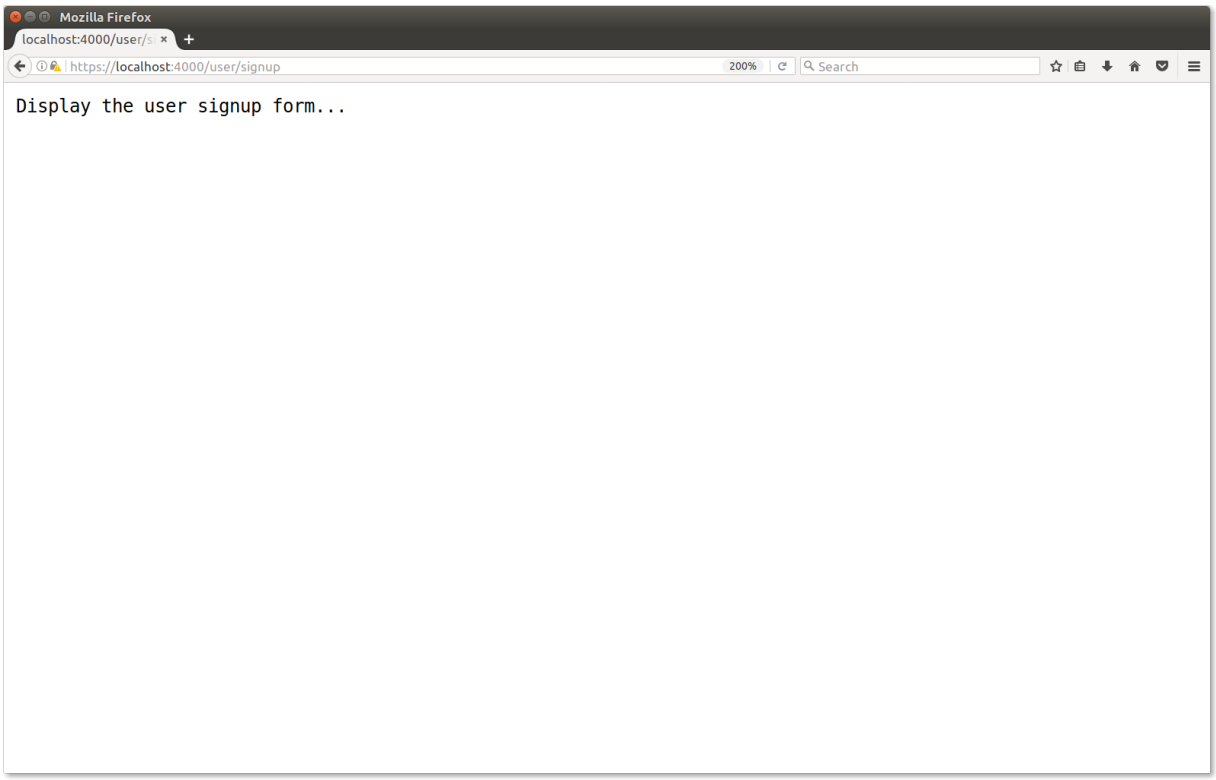
```
package main

...

func (app *App) SignupUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Display the user signup form...")
}

func (app *App) CreateUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Create a new user...")
}

func (app *App) LoginUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Display the user login form...")
}

func (app *App) VerifyUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Verify the user credentials...")
}

func (app *App) LogoutUser(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Logout the user...")
}
```

Once that's done, let's also add the corresponding routes to the `routes.go` file:

cmd/web/routes.go

```
package main

import (
    "net/http"

    "github.com/bmizerany/pat"
)

func (app *App) Routes() http.Handler {
    mux := pat.New()
    mux.Get("/", http.HandlerFunc(app.Home))
    mux.Get("/snippet/new", http.HandlerFunc(app.NewSnippet))
    mux.Post("/snippet/new", http.HandlerFunc(app.CreateSnippet))
    mux.Get("/snippet/:id", http.HandlerFunc(app.ShowSnippet))
    mux.Get("/user/signup", http.HandlerFunc(app.SignupUser))
    mux.Post("/user/signup", http.HandlerFunc(app.CreateUser))
    mux.Get("/user/login", http.HandlerFunc(app.LoginUser))
    mux.Post("/user/login", http.HandlerFunc(app.VerifyUser))
    mux.Post("/user/logout", http.HandlerFunc(app.LogoutUser))

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

    return LogRequest(SecureHeaders(mux))
}
```

If you like, you can run the application at this point and try visiting some of the new routes like `https://localhost:4000/user/signup`. You should see a plain-text response like this:



Adding a User Table

Now that the routes are set up, we need to create a `users` database table to store the details of our registered users. Connect to MySQL from your terminal window as the root user:

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

And execute the following SQL statement to setup the `users` table.

```
CREATE TABLE users (
    id INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    password CHAR(60) NOT NULL,
    created DATETIME NOT NULL
);
```

This `users` table is pretty simple but there's a couple of things worth pointing out:

- There's a `UNIQUE` constraint on the `email` column, which means that the table can't contain two emails which are the same. If we try to insert a duplicate email, MySQL will throw an `ERROR 1062: Duplicate entry` error.
- I've also set the type of the `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.

Password Hashing

If our database is ever compromised by an attacker, we don't want it to contain the plain-text versions of our user's passwords.

Instead, we should store a one-way hash of the password, derived with a slow hashing function such as Argon2, scrypt or bcrypt. Currently Go has good implementations of the scrypt and brcypt algorithms, but a plus-point of the bcrpyt implementation is that it includes some nice helper functions specifically for hashing and checking passwords.

If you're following along, go ahead and install it with `go get`:

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

There are two functions in the `bcrypt` package that we'll use. 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 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. It will return a 60 character long hash which looks a bit like this: `$2a$12$NuTjWXm3KKntReFwyBVHyuf/to.HEwTy.eS206TNfkGfr6HzGJSWG`.

Note that the `bcrypt.GenerateFromPassword()`function will also add a random salt to the password to help avoid rainbow-table 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"))
```

This will return `nil` if the plain-text password matches a particular hash, and an error if they don't match.

11.2. Navigation

Our first proper step in restricting user access will be to update the navigation bar so that when a user is *not logged in* they see links to `Home`, `Signup` and `Login`. And when they *are logged in*, they see links to `Home`, `Add Snippet` and `Logout`.

Let's update the `ui/html/base.html` to toggle these links based on a `{{.LoggedIn}}` flag (which we'll add to the application in a minute).

ui/html/base.html

```
{{define "base"}}
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>{{template "page-title" .}} - Snippetbox</title>
        <link rel="stylesheet" href="/static/css/main.css">
        <link rel="shortcut icon" href="/static/img/favicon.ico" type="image/x-icon">
    </head>
    <body>
        <header>
            <h1><a href="/">Snippetbox</a></h1>
        </header>
        <nav>
            <a href="/" {{if eq .Path "/"}}class="live"{{end}}>
                Home
            </a>
            {{if .LoggedIn}}
            <a href="/snippet/new" {{if eq .Path "/snippet/new"}}class="live"{{end}}>
                New snippet
            </a>
            <form action="/user/logout" method="POST">
                <button>Logout</button>
            </form>
            {{else}}
            <a href="/user/login" {{if eq .Path "/user/login"}}class="live"{{end}}>
                Login
            </a>
            <a href="/user/signup" {{if eq .Path "/user/signup"}}class="live"{{end}}>
                Signup
            </a>
            {{end}}
        </nav>
        <section>
            {{template "page-body" .}}
        </section>
    </body>
</html>
{{end}}
```

Getting the LoggedIn Status

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 visit a form at `/user/login` (which we'll create in a moment) and they enter their email address and password.

2. We then check the database to see if the credentials they entered match one of the users in the `users` table.

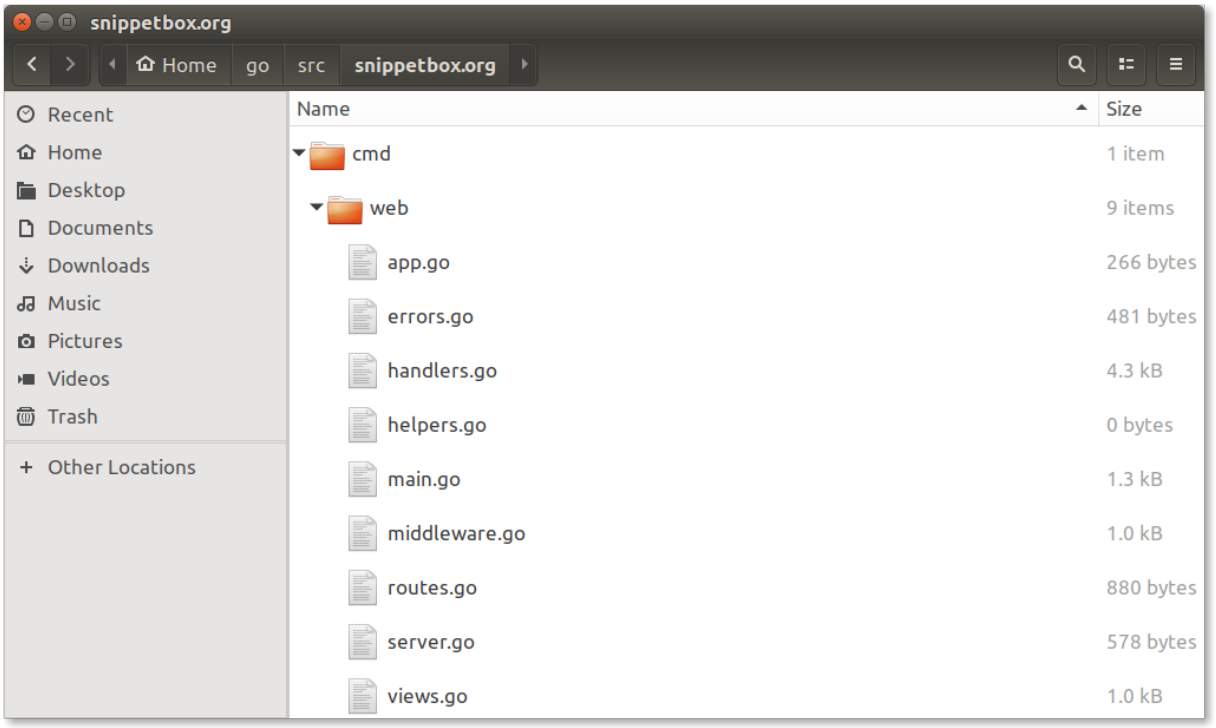
3. If there's a match, we add the relevant `id` value from the `users` table to our session data, using the key `"currentUserID"`.

4. When we receive any subsequent requests, we can check the session data for a `"currentUserID"` value. If it exists, we know that the user has logged in. We can keep checking this until the session expires, when the

user will need to log in again. If there's no `"currentUserID"` in the session, we know that the user is not logged in.

So let's create a helper function that does the checking of a user's status for us. Create a new `cmd/web/helpers.go` file:

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



And create an `app.LoggedIn()` helper method containing the following code:

cmd/web/helpers.go

```
package main

import (
    "net/http"
)

func (app *App) LoggedIn(r *http.Request) (bool, error) {
    // Load the session data for the current request, and use the Exists() method
    // to check if it contains a currentUserID key. This returns true if the
    // key is in the session data; false otherwise.
    session := app.Sessions.Load(r)
    loggedIn, err := session.Exists("currentUserID")
    if err != nil {
        return false, err
    }

    return loggedIn, nil
}
```

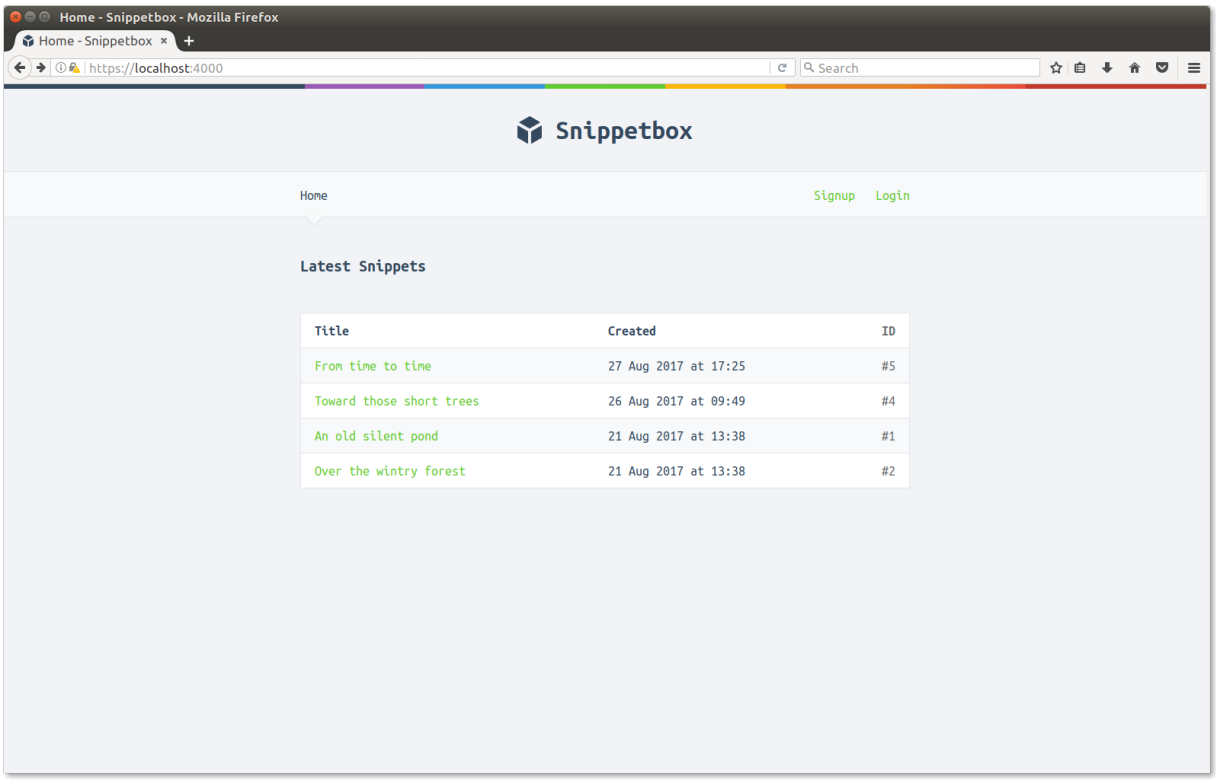
Then let's update the `views.go` file so that the status of the current user is passed to our templates every time we render a web page.

cmd/web/views.go

```
package main

...

// Add a LoggedIn field to the struct.
type HTMLData struct {
    Flash    string
    Form     interface{}
    LoggedIn bool
    Path     string
    Snippet  *models.Snippet
    Snippets []*models.Snippet
}

func (app *App) RenderHTML(w http.ResponseWriter, r *http.Request, page string, data *HTMLData) {
    if data == nil {
        data = &HTMLData{}
    }

    data.Path = r.URL.Path

    // Add the logged in status to the HTMLData.
    var err error
    data.LoggedIn, err = app.LoggedIn(r)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, page),
    }

    funcs := template.FuncMap{
        "humanDate": humanDate,
    }

    ts, err := template.New("").Funcs(funcs).ParseFiles(files...)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    buf := new(bytes.Buffer)
    err = ts.ExecuteTemplate(buf, "base", data)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    buf.WriteTo(w)
}
```

Save all the files and try running the application now. We (obviously) haven't logged into our application yet, so there's no `"currentUserID"` value in the session. In turn, this means that you should see the links to `Signup` and `Login` but the `Add Snippet` link should no longer be visible.

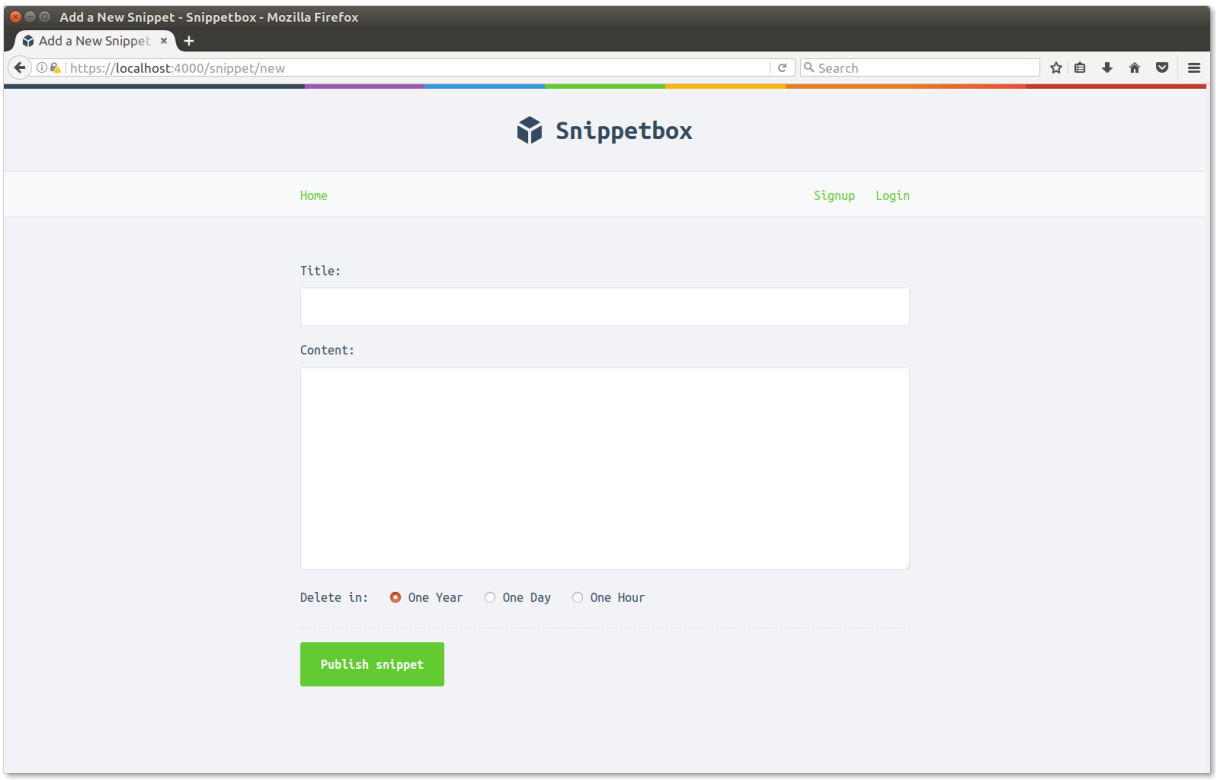
Like so:


Home - Snippetbox - Mozilla Firefox
Home - Snippetbox
https://localhost:4000
Search
Snippetbox
Home
Signup
Login
Latest Snippets
Title
Created
ID
From time to time
27 Aug 2017 at 17:25
#5
Toward those short trees
26 Aug 2017 at 09:49
#4
An old silent pond
21 Aug 2017 at 13:38
#1
Over the wintry forest
21 Aug 2017 at 13:38
#2

11.3. User Authorization

The aim of this chapter is to prevent users who are not logged in from being able to add a new snippet via the web application.

As it stands, we're hiding the navigation link to `Add Snippet` whenever a user isn't logged in, but anyone can still get to the page and submit the form by going to `https://localhost:4000/snippet/new` directly.



Let's fix that, so that if a user tries to visit `/snippet/new` when they're not logged in they are redirected to `/user/login` instead.

The simplest way to do this is via some middleware. Open up the `middleware.go` file and create a `RequireLogin` function as follows:

cmd/web/middleware.go

```
package main

...

func (app *App) RequireLogin(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Call the app.LoggedIn() helper to get the status for the current user.
        loggedIn, err := app.LoggedIn(r)
        if err != nil {
            app.ServerError(w, err)
            return
        }

        // If they are not logged in, redirect them to the login page and return
        // from the middleware chain so that no subsequent handlers in the chain
        // are executed.
        if !loggedIn {
            http.Redirect(w, r, "/user/login", 302)
            return
        }

        // Otherwise call the next handler in the chain.
        next.ServeHTTP(w, r)
    })
}
```

This follows broadly the same middleware pattern that we used before, but it has been implemented as a method on `App` instead of a standalone function. This is so it has access to the same dependencies and helpers that our other handlers do.

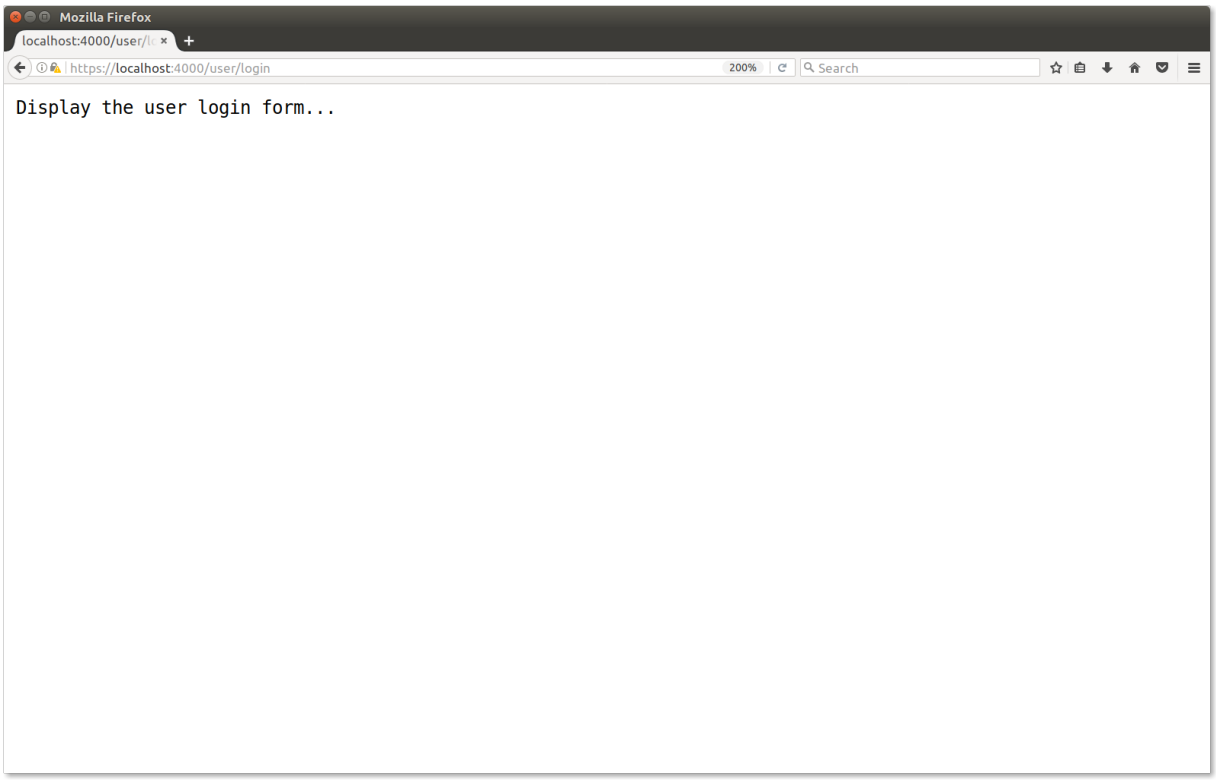
We can now add this middleware to our `routes.go` file to protect specific routes. In our case, we need to protect the `GET /snippet/new` and `POST /snippet/new` routes. Also, there's not much point logging out a user if they're not logged in, so it makes sense to use it on the `POST /user/logout` route as well.

Go ahead and wrap the three necessary handlers with the `app.RequireLogin()` middleware like so:

cmd/web/routes.go

```
package main

import (
    "net/http"

    "github.com/bmizerany/pat"
)

func (app *App) Routes() http.Handler {
    mux := pat.New()
    mux.Get("/", http.HandlerFunc(app.Home))
    mux.Get("/snippet/new", app.RequireLogin(http.HandlerFunc(app.NewSnippet)))
    mux.Post("/snippet/new", app.RequireLogin(http.HandlerFunc(app.CreateSnippet)))
    mux.Get("/snippet/:id", http.HandlerFunc(app.ShowSnippet))
    mux.Get("/user/signup", http.HandlerFunc(app.SignupUser))
    mux.Post("/user/signup", http.HandlerFunc(app.CreateUser))
    mux.Get("/user/login", http.HandlerFunc(app.LoginUser))
    mux.Post("/user/login", http.HandlerFunc(app.VerifyUser))
    mux.Post("/user/logout", app.RequireLogin(http.HandlerFunc(app.LogoutUser)))

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

    return LogRequest(SecureHeaders(mux))
}
```

Save the files, restart the application and try visiting `https://localhost:4000/snippet/new` again now. You should find that you get immediately redirected to `/user/login`.



If you like, you can also confirm with curl that non-logged in users are redirected for the `POST /snippet/new` route too:

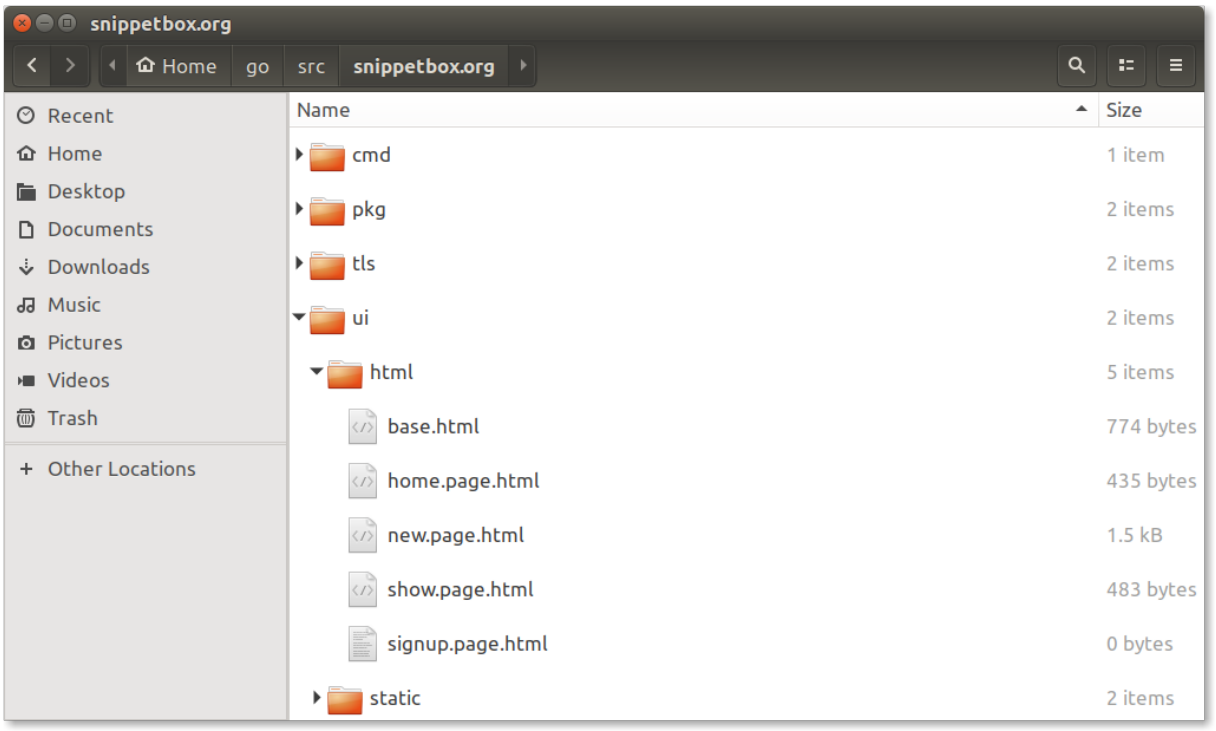
```
$ curl -ki -X POST https://localhost:4000/snippet/new
HTTP/1.1 302 Found
Location: /user/login
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
Date: Sat, 02 Sep 2017 16:38:46 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

11.4. User Signup and Password Encryption

Before we can log in any users we first need a way for them to sign up for an account. Let's cover that in this chapter.

Go ahead an create a new `ui/html/signup.page.html` file containing the following markup.

```
$ cd $HOME/go/src/snippetbox.org
$ touch ui/html/signup.page.html
```



ui/html/signup.page.html

```
{{define "page-title"}}Signup{{end}}

{{define "page-body"}}
<form action="/user/signup" method="POST" novalidate>
    {{with .Form}}
        <div>
            <label>Name:</label>
            {{with .Failures.Name}}
                <label class="error">{{.}}</label>
            {{end}}
            <input type="text" name="name" value="{{.Name}}">
        </div>
        <div>
            <label>Email:</label>
            {{with .Failures.Email}}
                <label class="error">{{.}}</label>
            {{end}}
            <input type="email" name="email" value="{{.Email}}">
        </div>
        <div>
            <label>Password:</label>
            {{with .Failures.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 be straightforward so far. We've made a template for the signup form which contains three fields: `name`, `email` and `password` using the relevant HTML5 input types.

Importantly, 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 another intermediary) caching the plain-text password entered by the user.

Form Validation

Next head over to your `pkg/forms/forms.go` file and create the code to validate the form contents, using the same approach that we did previously. Specifically, we want to make sure that the name, email and password fields are all completed, the email looks like a valid format, and the password is at least 8 characters long.

pkg/forms/forms.go

```
package forms

import (
    "regexp" // New import
    "strings"
    "unicode/utf8"
)

var rxEmail = regexp.MustCompile("^[a-zA-Z0-9.!#$%&'*+\\/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\\.[a-zA-Z0-9](?:[

...

type SignupUser struct {
    Name     string
    Email    string
    Password string
    Failures map[string]string
}

func (f *SignupUser) Valid() bool {
    f.Failures = make(map[string]string)

    if strings.TrimSpace(f.Name) == "" {
        f.Failures["Name"] = "Name is required"
    }

    if strings.TrimSpace(f.Email) == "" {
        f.Failures["Email"] = "Email is required"
    } else if len(f.Email) > 254 || !rxEmail.MatchString(f.Email) {
        f.Failures["Email"] = "Email is not a valid address"
    }

    if utf8.RuneCountInString(f.Password) < 8 {
        f.Failures["Password"] = "Password cannot be shorter than 8 characters"
    }

    return len(f.Failures) == 0
}
```

Notice that we're using a regular expression to sanity check the format of the email address using the `regexp.MatchString()` method?

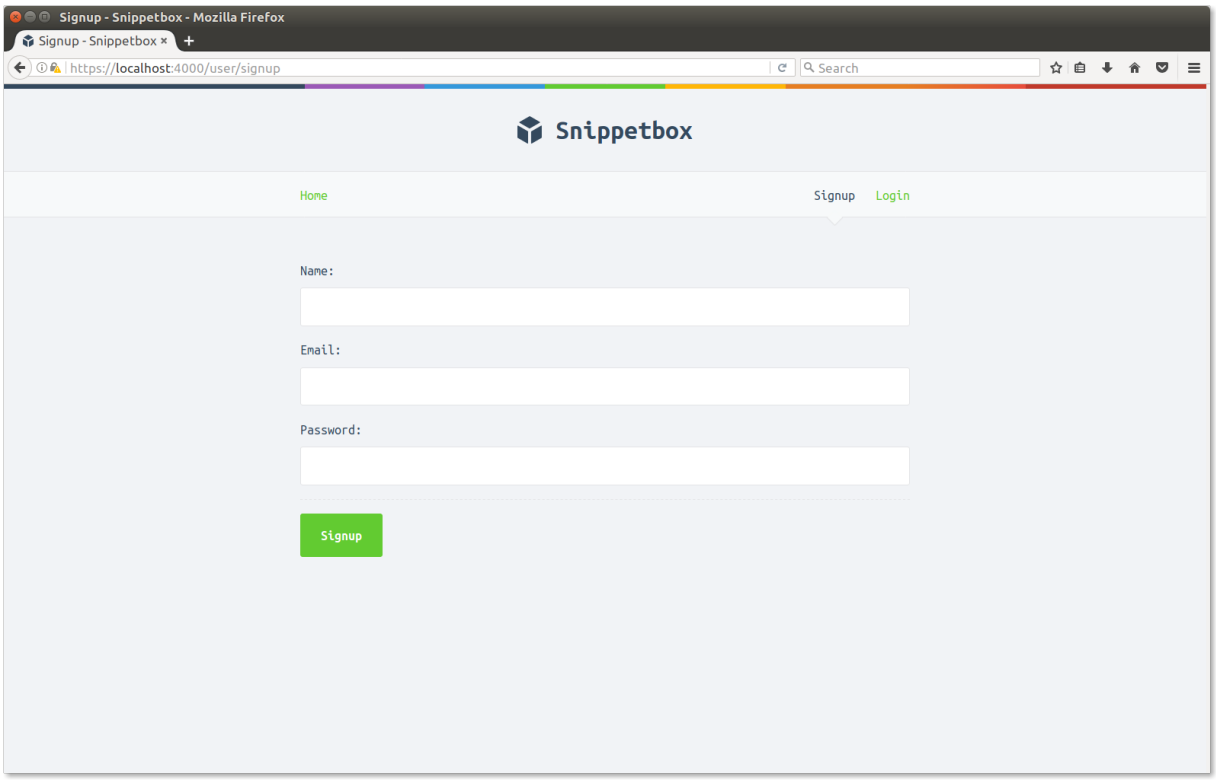
Although there are many different regular expression patterns you could use to check the email format I've opted for the pattern recommended by the W3C and Web Hypertext Application Technology Working Group.

Alright, let's hook this up to our `SignupUser` and `CreateUser` handlers like so:

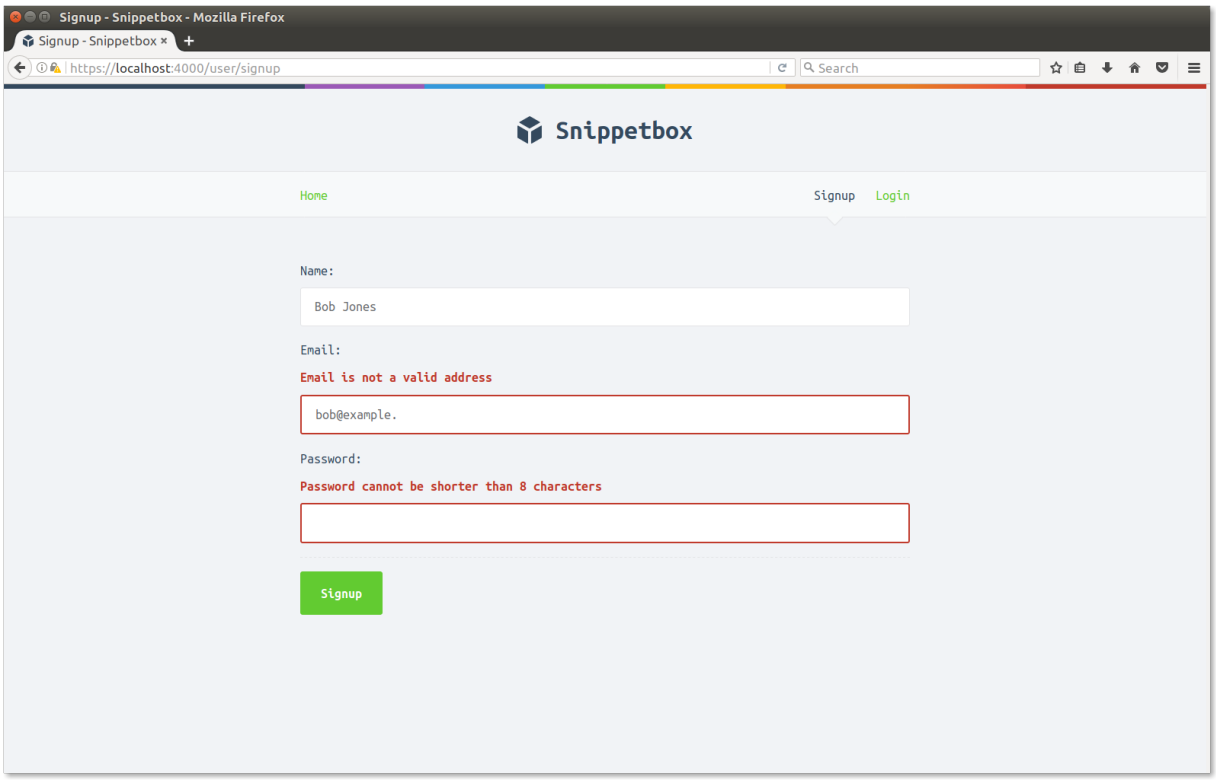
cmd/web/handlers.go

```
package main

...

func (app *App) SignupUser(w http.ResponseWriter, r *http.Request) {
    app.RenderHTML(w, r, "signup.page.html", &HTMLData{
        Form: &forms.SignupUser{},
    })
}

func (app *App) CreateUser(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.ClientError(w, http.StatusBadRequest)
        return
    }

    form := &forms.SignupUser{
        Name:     r.PostForm.Get("name"),
        Email:    r.PostForm.Get("email"),
        Password: r.PostForm.Get("password"),
    }

    if !form.Valid() {
        app.RenderHTML(w, r, "signup.page.html", &HTMLData{Form: form})
        return
    }

    fmt.Fprintln(w, "Create a new user...")
}

...
```

If you run the application now and visit `https://localhost:4000/user/signup` you should see a page which looks like this:



And if you try to submit an invalid form, you should see the appropriate validation failures like so:



Storing the User Details

The next stage is to add an `InsertUser()` method to our database model, so we can create 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 up the `pkg/models/database.go` file and add the following code:

pkg/models/database.go

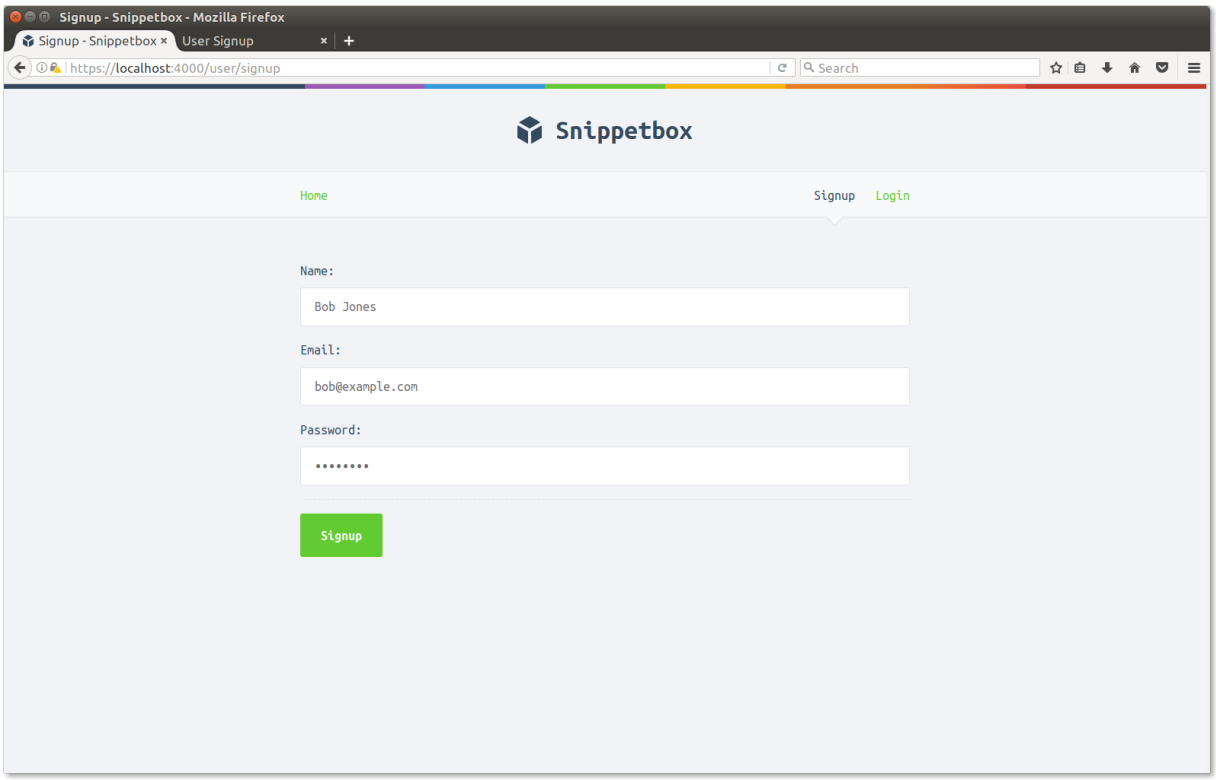
```
package models

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

    "github.com/go-sql-driver/mysql" // New import
    "golang.org/x/crypto/bcrypt"     // New import
)

// Declare a custom error to return if a duplicate email is added.
var ErrDuplicateEmail = errors.New("models: email address already in use")

type Database struct {
    *sql.DB
}

...

func (db *Database) InsertUser(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, password, created)
    VALUES(?, ?, ?, UTC_TIMESTAMP())`

    // Insert the user details and hashed password into the users table. If there
    // we type assert it to a *mysql.MySQLError object so we can check its
    // specific error number. If it's error 1062 we return the ErrDuplicateEmail
    // error instead of the one from MySQL.
    _, err = db.Exec(stmt, name, email, string(hashedPassword))
    if err != nil {
        if err.(*mysql.MySQLError).Number == 1062 {
            return ErrDuplicateEmail
        }
    }
    return err
}
```

We can finish off by calling this from our `CreateUser` handler like so:

cmd/web/handlers.go

```
package main

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

    "snippetbox.org/pkg/forms"
    "snippetbox.org/pkg/models" // New import
)

...

func (app *App) CreateUser(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.ClientError(w, http.StatusBadRequest)
        return
    }

    form := &forms.SignupUser{
        Name:     r.PostForm.Get("name"),
        Email:    r.PostForm.Get("email"),
        Password: r.PostForm.Get("password"),
    }

    if !form.Valid() {
        app.RenderHTML(w, r, "signup.page.html", &HTMLData{Form: form})
        return
    }

    // Try to create a new user record in the database. If the email already exists
    // add a failure message to the form and re-display the form.
    err = app.Database.InsertUser(form.Name, form.Email, form.Password)
    if err == models.ErrDuplicateEmail {
        form.Failures["Email"] = "Address is already in use"
        app.RenderHTML(w, r, "signup.page.html", &HTMLData{Form: form})
        return
    } else if err != nil {
        app.ServerError(w, err)
        return
    }

    // Otherwise, add a confirmation flash message to the session confirming that
    // their signup worked and asking them to log in.
    msg := "Your signup was successful. Please log in using your credentials."
    session := app.Sessions.Load(r)
    err = session.PutString(w, "flash", msg)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // 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, like so:

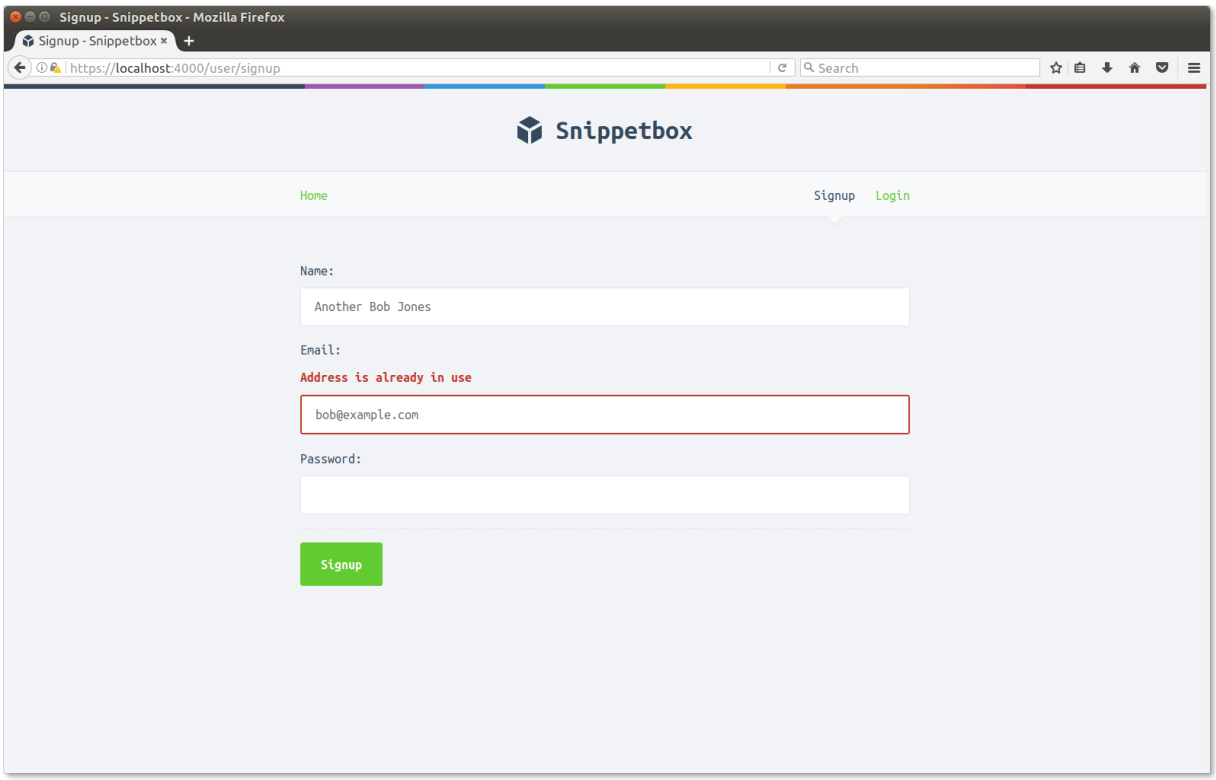


If it works correctly, you should find that your browser redirects you to `https://localhost:4000/user/login` after you submit the form.

At this point it's worth at this point opening up your MySQL database and taking a look 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 password.

```
$ mysql -D snippetbox -u root -p
Enter password:
mysql> SELECT * FROM users;
+----+-----------+-----------------+--------------------------------------------------------------+---------------------+
| id | name      | email           | password                                                     | created             |
+----+-----------+-----------------+--------------------------------------------------------------+---------------------+
|  1 | Bob Jones | bob@example.com | $2a$12$VDPgFm7JxDXQR5Wjw3dnieLUYo1PE0rTqSMgmmlQ3XUUkdsTzzZmC | 2017-09-02 10:31:06 |
+----+-----------+-----------------+--------------------------------------------------------------+---------------------+
1 row in set (0.00 sec)
```

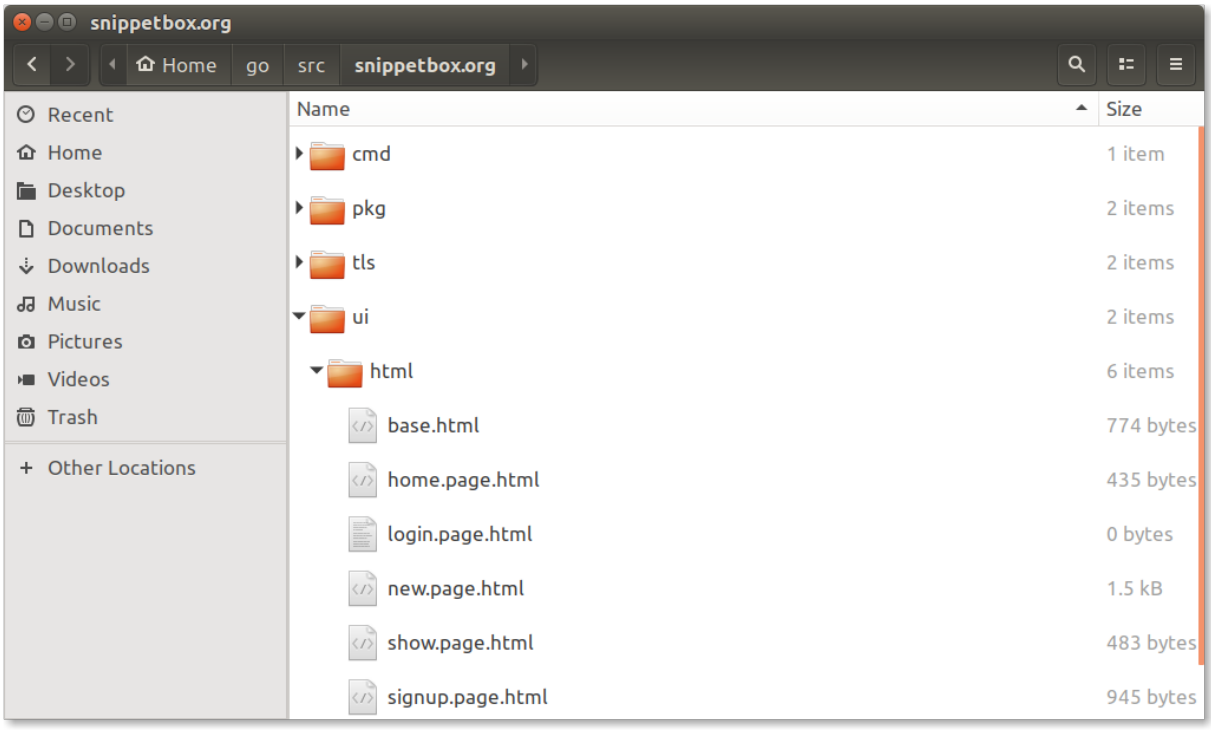
If you like, try heading back to the signup form and adding another account with the same email address. You should get a validation failure like so:


Signup - Snippetbox - Mozilla Firefox
Signup - Snippetbox
https://localhost:4000/user/signup
Search
Snippetbox
Home
Signup
Login
Name:
Another Bob Jones
Email:
Address is already in use
bob@example.com
Password:
Signup

11.5. User Login

The process for creating the user login page follows pretty much the same pattern as the user signup. First, create a `ui/html/login.page.html` template containing the markup below:

```
$ cd $HOME/go/src/snippetbox.org
$ touch ui/html/login.page.html
```



ui/html/login.page.html

```
{{define "page-title"}}Login{{end}}

{{define "page-body"}}
    {{with .Flash}}
    <div class="flash">{{.}}</div>
    {{end}}
    <form action="/user/login" method="POST" novalidate>
        {{with .Form}}
            {{with .Failures.Generic}}
                <div class="error">{{.}}</div>
            {{end}}
            <div>
                <label>Email:</label>
                {{with .Failures.Email}}
                    <label class="error">{{.}}</label>
                {{end}}
                <input type="email" name="email" value="{{.Email}}">
            </div>
            <div>
                <label>Password:</label>
                {{with .Failures.Password}}
                    <label class="error">{{.}}</label>
                {{end}}
                <input type="password" name="password">
            </div>
            <div>
                <input type="submit" value="Login">
            </div>
        {{end}}
    </form>
{{end}}
```

Notice how we've included a `{{with .Failures.Generic}}` action at the top of the form? We'll use this to display a generic "invalid credentials" message if the user's login fails, which is more secure than explicitly indicating which of the fields is wrong. There's also a `{{with .Flash}}` action above the form to display any flash messages.

Again, we're not re-displaying the user's password if the login fails.

Form Validation

Now let's add a couple of validation rules to the `pkg/forms/forms.go` file. The only thing we'll check is that the user hasn't left either of the two fields blank.

pkg/forms/forms.go

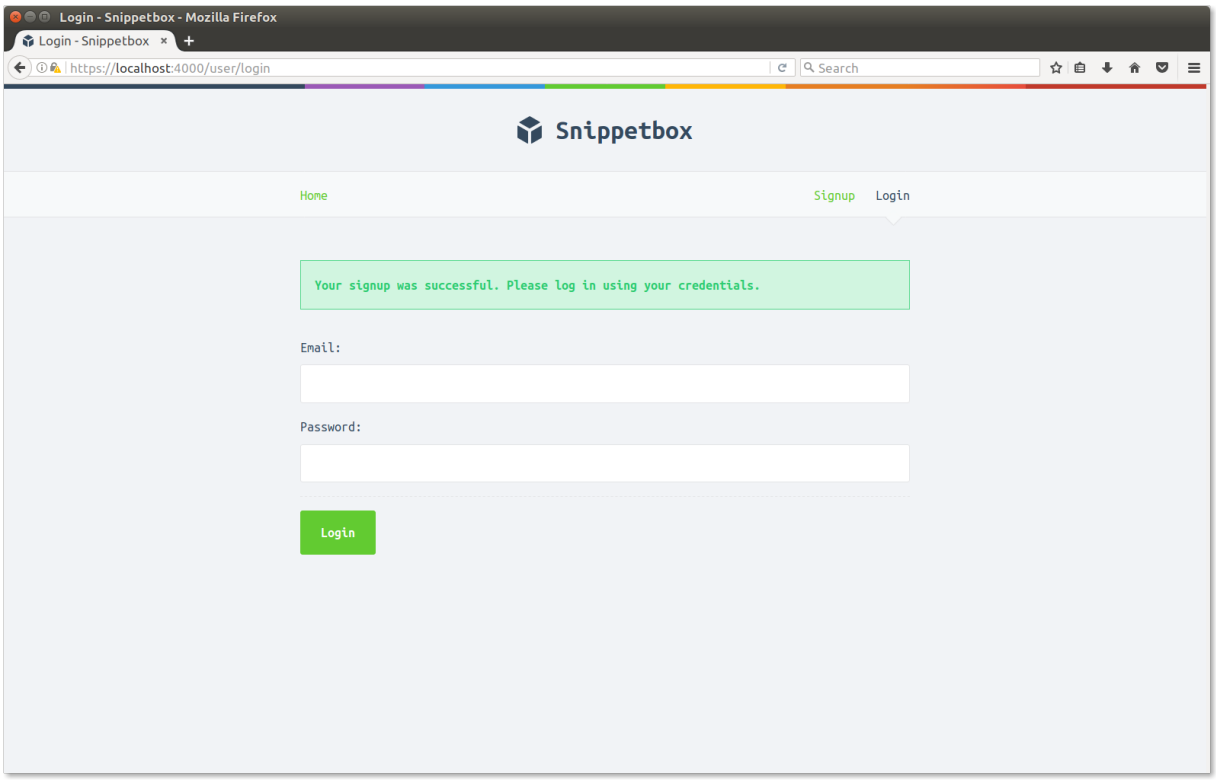
```
package forms

...

type LoginUser struct {
    Email    string
    Password string
    Failures map[string]string
}

func (f *LoginUser) Valid() bool {
    f.Failures = make(map[string]string)

    if strings.TrimSpace(f.Email) == "" {
        f.Failures["Email"] = "Email is required"
    }

    if strings.TrimSpace(f.Password) == "" {
        f.Failures["Password"] = "Password is required"
    }

    return len(f.Failures) == 0
}
```

Then let's update the `LoginUser` and `VerifyUser` handers to display the login page and carry out the validation. Note that we want the `LoginUser` handler (which displays the empty form) to check the session for a flash message and include that when rendering the template too.

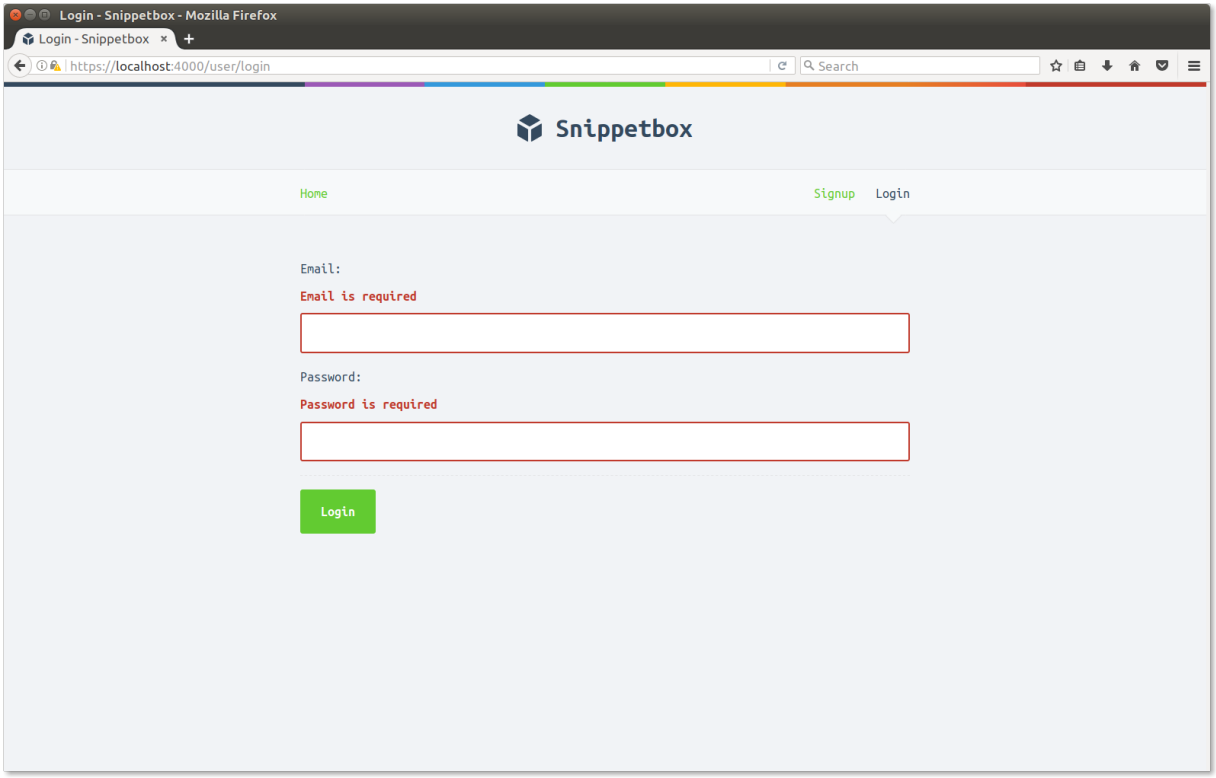
cmd/web/handlers.go

```
package main

...

func (app *App) LoginUser(w http.ResponseWriter, r *http.Request) {
    session := app.Sessions.Load(r)
    flash, err := session.PopString(w, "flash")
    if err != nil {
        app.ServerError(w, err)
        return
    }

    app.RenderHTML(w, r, "login.page.html", &HTMLData{
        Flash: flash,
        Form:  &forms.LoginUser{},
    })
}

func (app *App) VerifyUser(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.ClientError(w, http.StatusBadRequest)
        return
    }

    form := &forms.LoginUser{
        Email:    r.PostForm.Get("email"),
        Password: r.PostForm.Get("password"),
    }

    if !form.Valid() {
        app.RenderHTML(w, r, "login.page.html", &HTMLData{Form: form})
        return
    }

    fmt.Fprintln(w, "Verify the user credentials...")
}

...
```

If you restart the application and visit `https://localhost:4000/user/login` you should now see the login form. If it's the first time visiting that URL since you signed up, it should also include a flash message similar to the below screenshot.



Again, you can check that the validation works by submitting an empty form:



Verifying the User Details

The next step is the interesting part: *how do we verify that the email and password are correct?*

Let's add an `VerifyUser()` method to our database model which does 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 a generic "invalid credentials" error.

2. If the email does exist, we want to compare the bcrpyt hashed password to the plain-text password that the user provided when logging in. If they don't match, we want to return a generic "invalid credentials" error. If they do match, we want to return the user's ID.

Go ahead and add the following code to your `pkg/models/database.go` file:

pkg/models/database.go

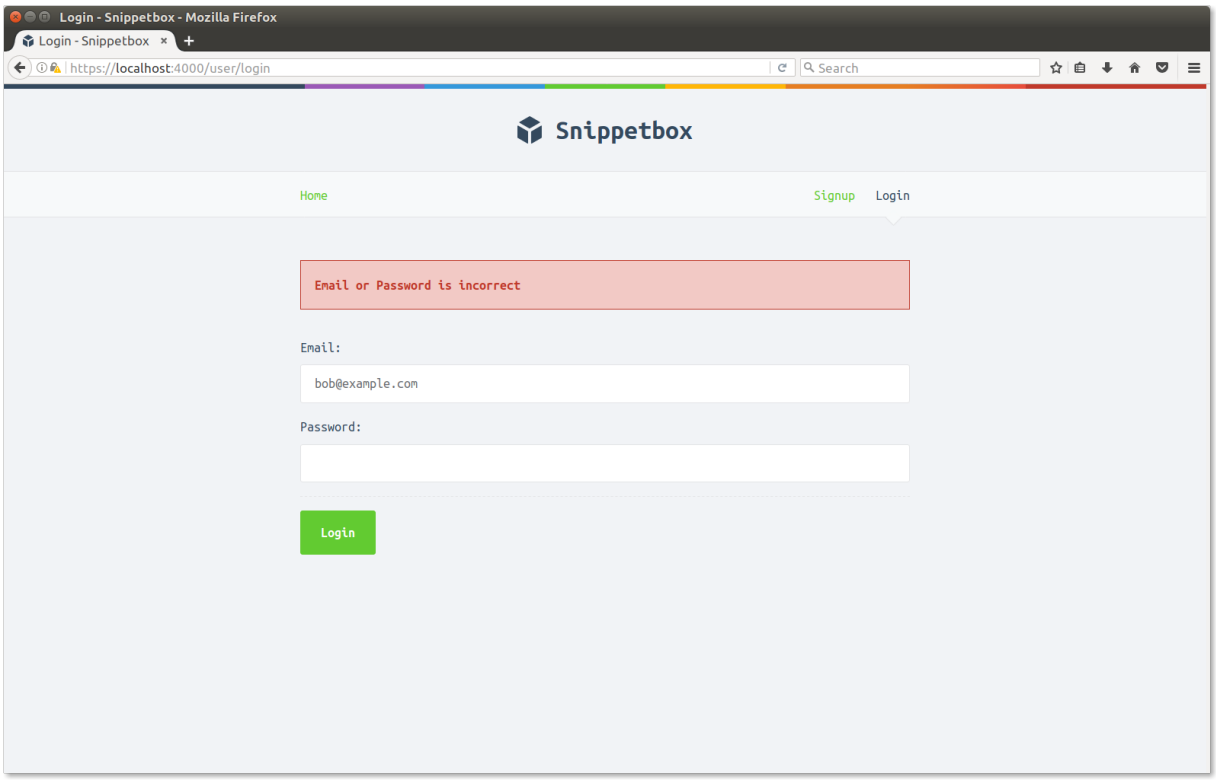
```
package models

import (
    "database/sql"
    "errors"

    "github.com/go-sql-driver/mysql"
    "golang.org/x/crypto/bcrypt"
)

// Create a new ErrInvalidCredentials error that we can return.
var (
    ErrDuplicateEmail     = errors.New("models: email address already in use")
    ErrInvalidCredentials = errors.New("models: invalid user credentials")
)

...

func (db *Database) VerifyUser(email, password string) (int, error) {
    // Retrieve the id and hashed password associated with the given email. If no
    // matching email exists, we return the ErrInvalidCredentials error.
    var id int
    var hashedPassword []byte
    row := db.QueryRow("SELECT id, password FROM users WHERE email = ?", email)
    err := row.Scan(&id, &hashedPassword)
    if err == sql.ErrNoRows {
        return 0, 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, ErrInvalidCredentials
    } else if err != nil {
        return 0, err
    }

    // Otherwise, the password is correct. Return the user ID.
    return id, nil
}

...
```

Let's now use this in our `VerifyUser` handler function:

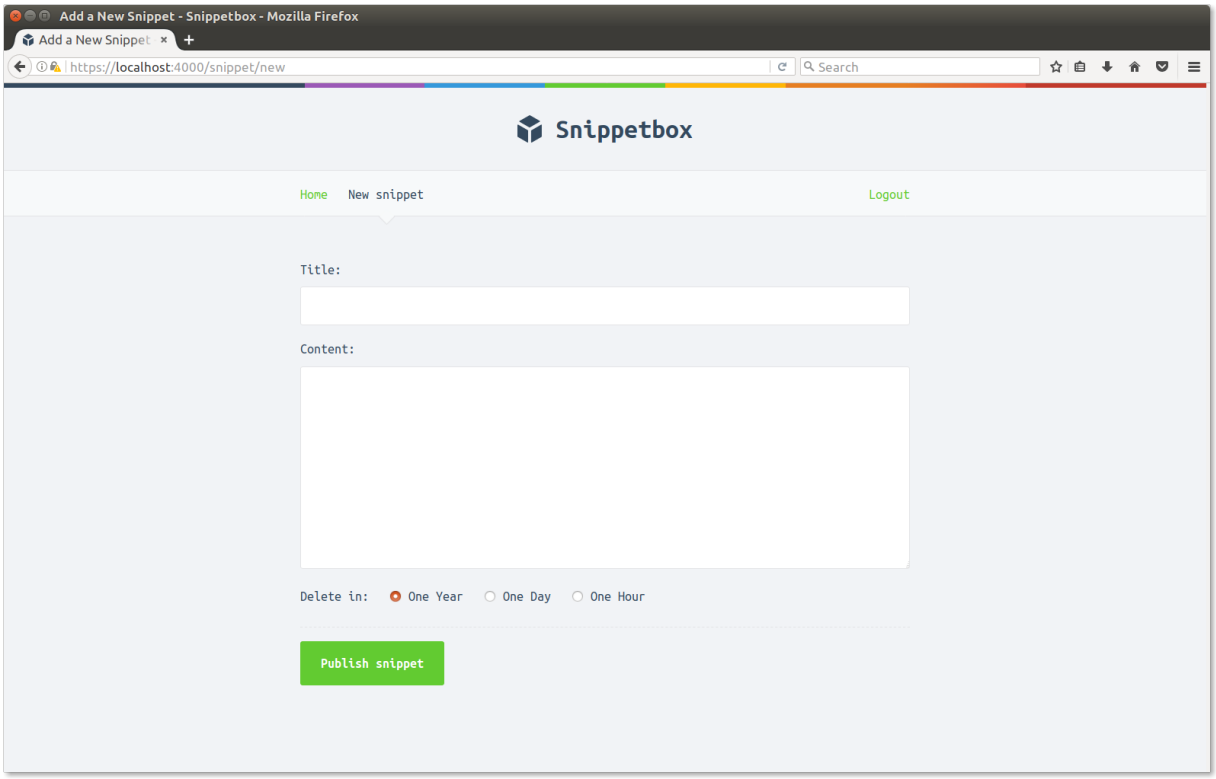
cmd/web/handlers.go

```
package main

...

func (app *App) VerifyUser(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        app.ClientError(w, http.StatusBadRequest)
        return
    }

    form := &forms.LoginUser{
        Email:    r.PostForm.Get("email"),
        Password: r.PostForm.Get("password"),
    }

    if !form.Valid() {
        app.RenderHTML(w, r, "login.page.html", &HTMLData{Form: form})
        return
    }

    // Check whether the credentials are valid. If they're not, add a generic error
    // message to the form failures map, and re-display the login page.
    currentUserID, err := app.Database.VerifyUser(form.Email, form.Password)
    if err == models.ErrInvalidCredentials {
        form.Failures["Generic"] = "Email or Password is incorrect"
        app.RenderHTML(w, r, "login.page.html", &HTMLData{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 'logged
    // in'.
    session := app.Sessions.Load(r)
    err = session.PutInt(w, "currentUserID", currentUserID)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // Redirect the user to the Add Snippet page.
    http.Redirect(w, r, "/snippet/new", http.StatusSeeOther)
}

...
```

So let's give this a try. Restart the application and try submitting some invalid user credentials. You should get a validation failure which looks like this:



But when you input the correct ones, the application should log you in and redirect you to the Add Snippet page, like so:



Notice how the navigation has changed, and the logout link is now visible?

Notes

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, there would be a risk of a session fixation attack occurring. You must use the `session.RenewToken()` to change the session token *before* you make any to privilege levels (e.g. login and logout operations).

11.6. User Logout

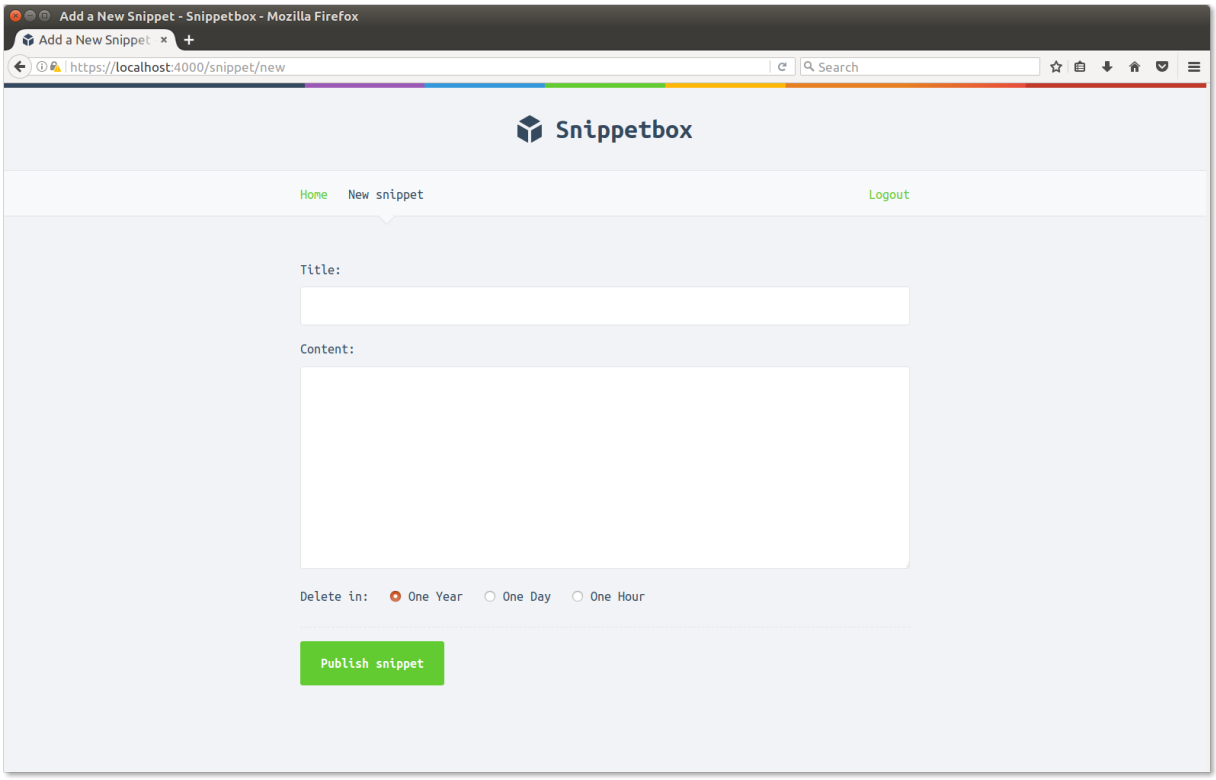
This brings us nicely to logging out a user. Implementing the user logout is pretty straightforward in comparison to the signup and login – all we need to do is remove the `"currentUserID"` value from the session.

Let's update the `LogoutUser` hander to do exactly that.

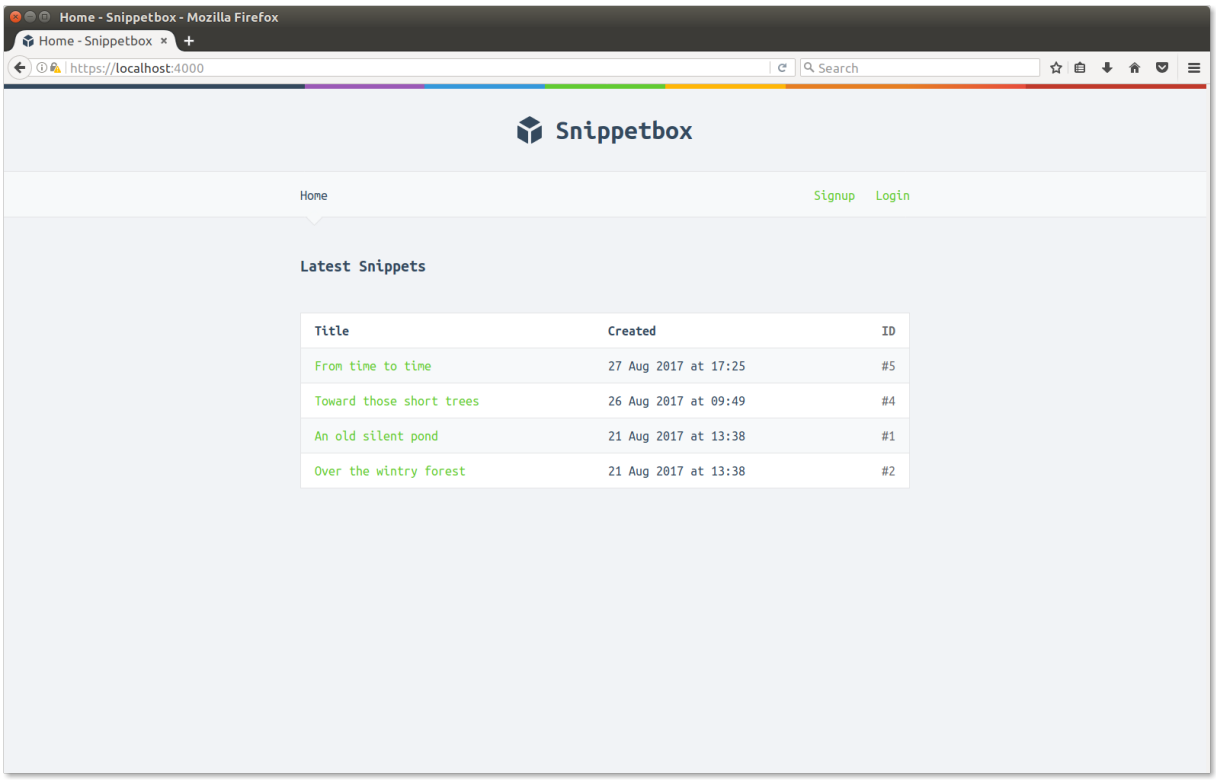
cmd/web/handlers.go

```
package main

...

func (app *App) LogoutUser(w http.ResponseWriter, r *http.Request) {
    // Remove the currentUserID from the session data.
    session := app.Sessions.Load(r)
    err := session.Remove(w, "currentUserID")
    if err != nil {
        app.ServerError(w, err)
        return
    }

    // Redirect the user to the homepage.
    http.Redirect(w, r, "/", 303)
}
```

Save the file and restart the server. If you now click the `Logout` link...



You should be logged out and redirected to the homepage. Again, notice how the navigation has changed?


Home - Snippetbox - Mozilla Firefox
Home - Snippetbox
https://localhost:4000
Search
Snippetbox
Home
Signup
Login
Latest Snippets
Title
Created
ID
From time to time
27 Aug 2017 at 17:25
#5
Toward those short trees
26 Aug 2017 at 09:49
#4
An old silent pond
21 Aug 2017 at 13:38
#1
Over the wintry forest
21 Aug 2017 at 13:38
#2

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 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/new` to add a new snippet to our database.
- Since the user is still logged in to our application, the request is processed with the user's 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 also may be at risk from novel login and logout CSRF attacks.

Preventing CSRF Attacks

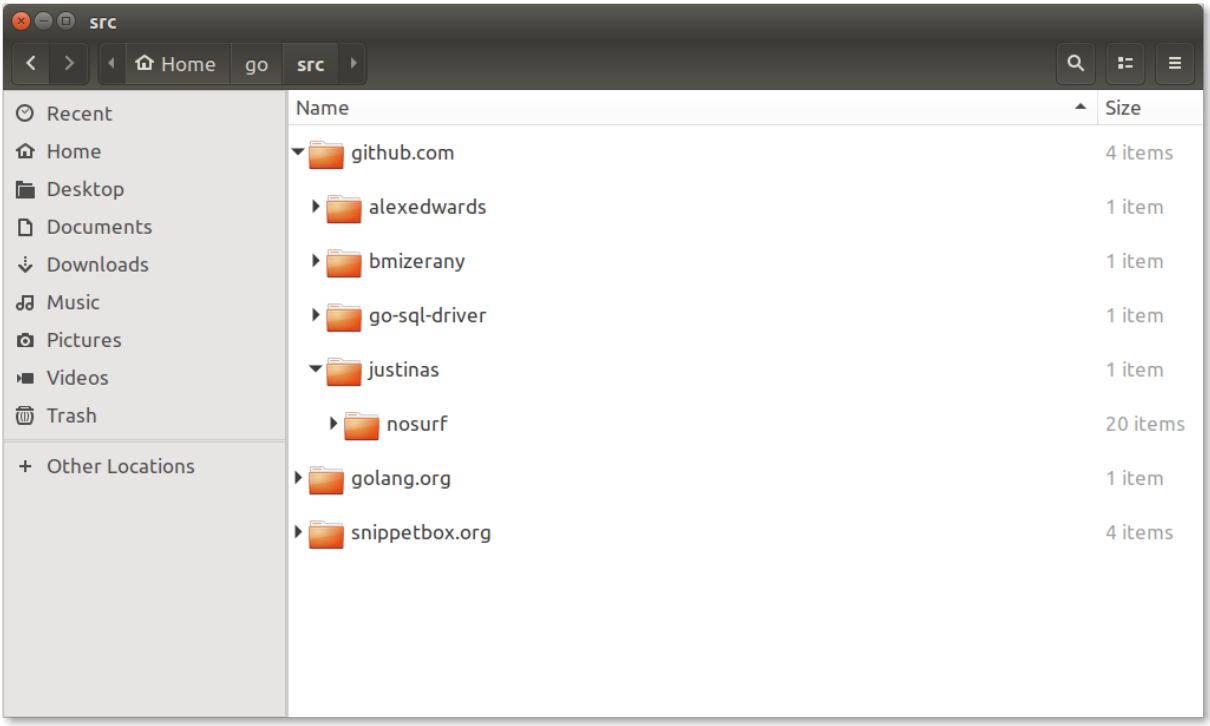
Like session management and password hashing, when it comes to CSRF protection it's probably safer to use a tried-and-tested third-party package instead of rolling your own implementation.

The two most popular packages for Go are Gorilla CSRF and nosurf. They both do roughly the same thing, using the Double Submit Cookie pattern to prevent CSRF attacks. In this pattern a random **CSRF token** is generated and sent to the user in a **CSRF cookie**. We then also need to add this CSRF token to in a hidden field in each form that's vulnerable to CSRF. When the form is submitted, the packages then check that the hidden field value and cookie value match.

Out of the two packages, we'll opt for nosurf – mainly because it's self-contained and doesn't have any additional dependencies.

If you're following along, go ahead and install it:

```
$ go get github.com/justinas/nosurf
```



Using nosurf

Open up your `middleware.go` file and create a new `NoSurf` function like so:

cmd/web/middleware.go

```
package main

import (
    "log"
    "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.HandlerFunc) http.Handler {
    csrfHandler := nosurf.New(next)
    csrfHandler.SetBaseCookie(http.Cookie{
        HttpOnly: true,
        Path:     "/",
        Secure:   true,
    })

    return csrfHandler
}
```

Notice how the signature of this is slightly different to our other middleware? Instead of accepting a `http.Handler` it accepts a `http.HandlerFunc` instead. Making this slight tweak will help neaten up our routes slightly, as we can pass our handler functions straight to the middleware without converting them first.

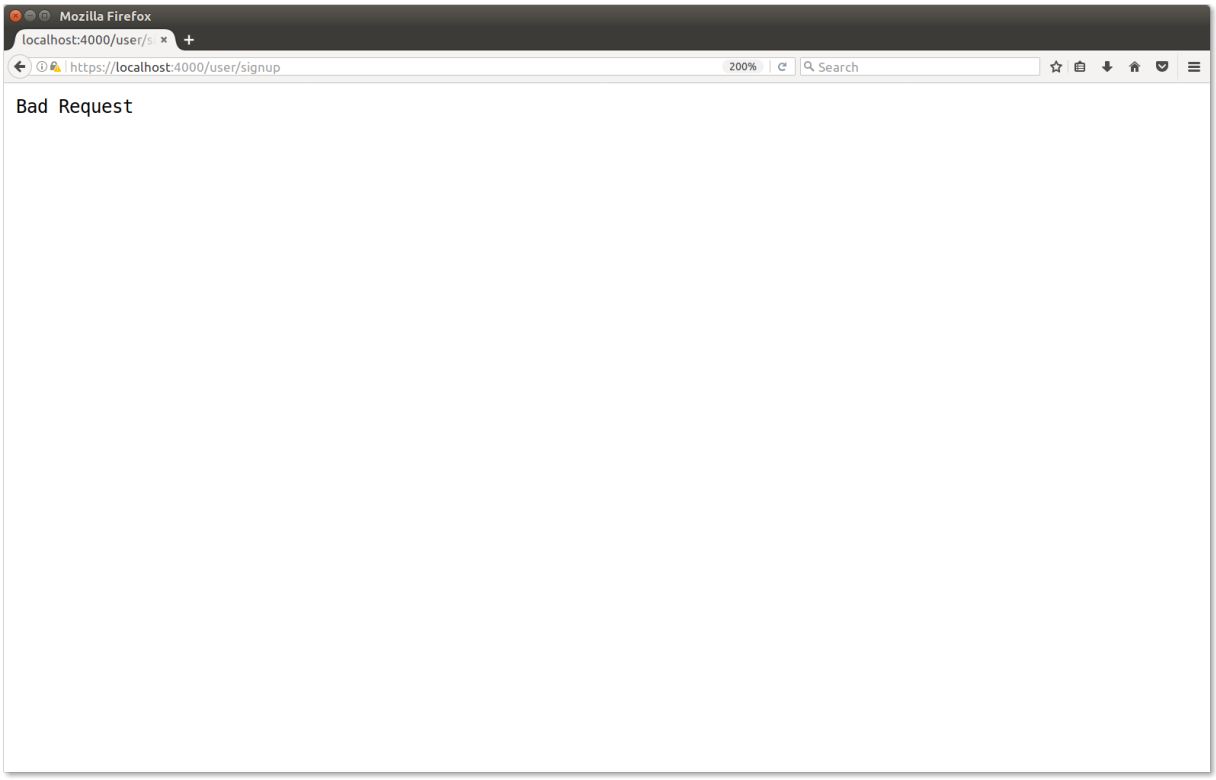
Update the `routes.go` file to use the NoSurf middleware on all our dynamic routes:

cmd/web/routes.go

```
package main

...

func (app *App) Routes() http.Handler {
    // Wrap all of our web page route with the NoSurf middleware.
    mux := pat.New()
    mux.Get("/", NoSurf(app.Home))
    mux.Get("/snippet/new", app.RequireLogin(NoSurf(app.NewSnippet)))
    mux.Post("/snippet/new", app.RequireLogin(NoSurf(app.CreateSnippet)))
    mux.Get("/snippet/:id", NoSurf(app.ShowSnippet))
    mux.Get("/user/signup", NoSurf(app.SignupUser))
    mux.Post("/user/signup", NoSurf(app.CreateUser))
    mux.Get("/user/login", NoSurf(app.LoginUser))
    mux.Post("/user/login", NoSurf(app.VerifyUser))
    mux.Post("/user/logout", app.RequireLogin(NoSurf(app.LogoutUser)))

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

    return LogRequest(SecureHeaders(mux))
}
```

At this point, you might like to fire up the application and try submitting one of the forms.

When you do, the request should be intercepted by the `NoSurf` middleware and you should receive a `400 Bad Request` response.



To make the form submissions work, we need to use the `nosurf.Token()` function to get the CSRF token and add it to a hidden `csrf_token` field in our forms.

Because the logout form can potentially appear on every page, it makes sense to generate this token each time we render a page:

cmd/web/views.go

```
package main

import (
    "bytes"
    "html/template"
    "net/http"
    "path/filepath"
    "time"

    "snippetbox.org/pkg/models"

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

...

type HTMLData struct {
    CSRFToken string
    Flash     string
    Form      interface{}
    LoggedIn  bool
    Path      string
    Snippet   *models.Snippet
    Snippets  []*models.Snippet
}

func (app *App) RenderHTML(w http.ResponseWriter, r *http.Request, page string, data *HTMLData) {
    if data == nil {
        data = &HTMLData{}
    }

    data.Path = r.URL.Path

    // Always add the CSRF token to the data for our templates.
    data.CSRFToken = nosurf.Token(r)

    var err error
    data.LoggedIn, err = app.LoggedIn(r)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    files := []string{
        filepath.Join(app.HTMLDir, "base.html"),
        filepath.Join(app.HTMLDir, page),
    }

    funcs := template.FuncMap{
        "humanDate": humanDate,
    }

    ts, err := template.New("").Funcs(funcs).ParseFiles(files...)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    buf := new(bytes.Buffer)
    err = ts.ExecuteTemplate(buf, "base", data)
    if err != nil {
        app.ServerError(w, err)
        return
    }

    buf.WriteTo(w)
}
```

Finally, let's update all the forms in our application to use this token.

ui/html/base.html

```
{{define "base"}}
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>{{template "page-title" .}} - Snippetbox</title>
        <link rel="stylesheet" href="/static/css/main.css">
        <link rel="shortcut icon" href="/static/img/favicon.ico" type="image/x-icon">
    </head>
    <body>
        <header>
            <h1><a href="/">Snippetbox</a></h1>
        </header>
        <nav>
            <a href="/" {{if eq .Path "/"}}class="live"{{end}}>
                Home
            </a>
            {{if .LoggedIn}}
            <a href="/snippet/new" {{if eq .Path "/snippet/new"}}class="live"{{end}}>
                New snippet
            </a>
            <form action="/user/logout" method="POST">
                <!-- Add a hidden input containing the CSRF token -->
                <input type="hidden" name="csrf_token" value="{{.CSRFToken}}">
                <button>Logout</button>
            </form>
            {{else}}
            <a href="/user/login" {{if eq .Path "/user/login"}}class="live"{{end}}>
                Login
            </a>
            <a href="/user/signup" {{if eq .Path "/user/signup"}}class="live"{{end}}>
                Signup
            </a>
            {{end}}
        </nav>
        <section>
            {{template "page-body" .}}
        </section>
    </body>
</html>
{{end}}
```

ui/html/login.page.html

```
{{define "page-title"}}Login{{end}}

{{define "page-body"}}
    {{with .Flash}}
    <div class="flash">{{.}}</div>
    {{end}}
    <form action="/user/login" method="POST" novalidate>
        <!-- Add a hidden input containing the CSRF token -->
        <input type="hidden" name="csrf_token" value="{{.CSRFToken}}">
        {{with .Form}}
            {{with .Failures.Generic}}
                <div class="error">{{.}}</div>
            {{end}}
            <div>
                <label>Email:</label>
                {{with .Failures.Email}}
                    <label class="error">{{.}}</label>
                {{end}}
                <input type="email" name="email" value="{{.Email}}">
            </div>
            <div>
                <label>Password:</label>
                {{with .Failures.Password}}
                    <label class="error">{{.}}</label>
                {{end}}
                <input type="password" name="password">
            </div>
            <div>
                <input type="submit" value="Login">
            </div>
        {{end}}
    </form>
{{end}}
```

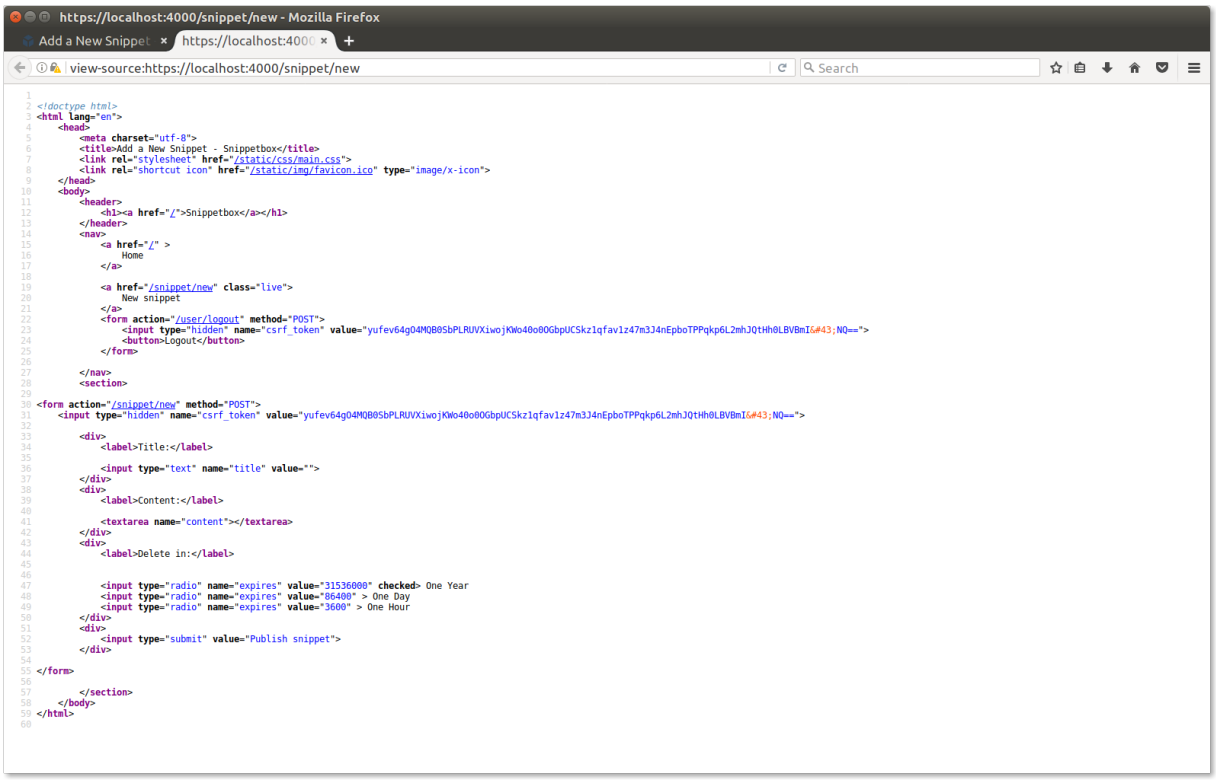
ui/html/new.page.html

```
{{define "page-title"}}Add a New Snippet{{end}}

{{define "page-body"}}
<form action="/snippet/new" method="POST">
    <!-- Add a hidden input containing the CSRF token -->
    <input type="hidden" name="csrf_token" value="{{.CSRFToken}}">
    {{with .Form}}
        <div>
            <label>Title:</label>
            {{with .Failures.Title}}
                <label class="error">{{.}}</label>
            {{end}}
            <input type="text" name="title" value="{{.Title}}">
        </div>
        <div>
            <label>Content:</label>
            {{with .Failures.Content}}
                <label class="error">{{.}}</label>
            {{end}}
            <textarea name="content">{{.Content}}</textarea>
        </div>
        <div>
            <label>Delete in:</label>
            {{with .Failures.Expires}}
                <label class="error">{{.}}</label>
            {{end}}
            {{$expires := or .Expires "31536000"}}
            <input type="radio" name="expires" value="31536000" {{if (eq $expires "31536000")}}checked{{end}}> One Year
            <input type="radio" name="expires" value="86400" {{if (eq $expires "86400")}}checked{{end}}> One Day
            <input type="radio" name="expires" value="3600" {{if (eq $expires "3600")}}checked{{end}}> One Hour
        </div>
        <div>
            <input type="submit" value="Publish snippet">
        </div>
    {{end}}
</form>
{{end}}
```

ui/html/signup.page.html

```
{{define "page-title"}}Signup{{end}}

{{define "page-body"}}
<form action="/user/signup" method="POST" novalidate>
    <!-- Add a hidden input containing the CSRF token -->
    <input type="hidden" name="csrf_token" value="{{.CSRFToken}}">
    {{with .Form}}
        <div>
            <label>Name:</label>
            {{with .Failures.Name}}
                <label class="error">{{.}}</label>
            {{end}}
            <input type="text" name="name" value="{{.Name}}">
        </div>
        <div>
            <label>Email:</label>
            {{with .Failures.Email}}
                <label class="error">{{.}}</label>
            {{end}}
            <input type="email" name="email" value="{{.Email}}">
        </div>
        <div>
            <label>Password:</label>
            {{with .Failures.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 has a CSRF token in a hidden field, like so.




```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Add a New Snippet - Snippetbox</title>
        <link rel="stylesheet" href="/static/css/main.css">
        <link rel="shortcut icon" href="/static/img/favicon.ico" type="image/x-icon">
    </head>
    <body>
        <header>
            <h1><a href="/">Snippetbox</a></h1>
        </header>
        <nav>
            <a href="/" >
                Home
            </a>

            <a href="/snippet/new" class="live">
                New snippet
            </a>
            <form action="/user/logout" method="POST">
                <input type="hidden" name="csrf_token" value="yufev64gO4MQB0SbPLRUVXiwojKWo40o0GbpUCSkz1qfav1z47m3J4nEpboTPPqkp6L2mhJQtHh0LBVBmI&amp;#43;NQ==">
                <button>Logout</button>
            </form>

        </nav>
        <section>

<form action="/snippet/new" method="POST">
    <input type="hidden" name="csrf_token" value="yufev64gO4MQB0SbPLRUVXiwojKWo40o0GbpUCSkz1qfav1z47m3J4nEpboTPPqkp6L2mhJQtHh0LBVBmI&amp;#43;NQ==">

        <div>
            <label>Title:</label>

            <input type="text" name="title" value="">
        </div>
        <div>
            <label>Content:</label>

            <textarea name="content"></textarea>
        </div>
        <div>
            <label>Delete in:</label>


            <input type="radio" name="expires" value="31536000" checked> One Year
            <input type="radio" name="expires" value="86400" > One Day
            <input type="radio" name="expires" value="3600" > One Hour
        </div>
        <div>
            <input type="submit" value="Publish snippet">
        </div>

</form>

        </section>
    </body>
</html>
```

If you submit the forms now, they should work exactly the same as previously but they're now safe from malicious CSRF requests.

12. Appendices

- How HTTPS Works

12.1. How HTTPS Works

You can think of a TLS connection happening in two stages. The first 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. The data is encrypted and signed using the session keys generated during the handshake.

In summary:

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 chosen.

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 of 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 is able to 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 to 5 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 this back to the client. It signs what it sends with it's own RSA private key.

In step 5, the client then encrypts the session keys using the ECDHE public key, not 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 have to 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).