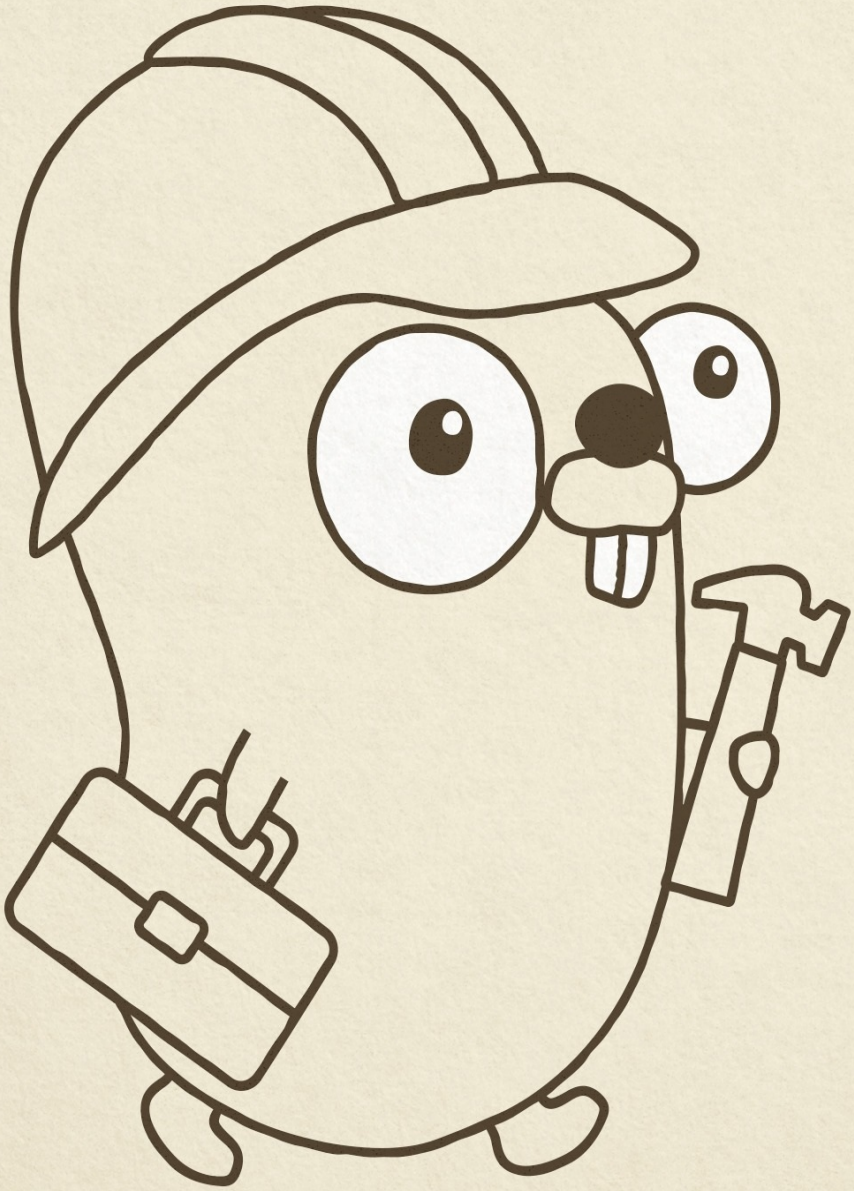
BUILDING WEB APPS WITH GO



Jeremy Saenz

Table of Contents

Introduction

Welcome to **Building Web Apps with Go**! If you are reading this then you have just started your journey from noob to pro. No seriously, web programming in Go is so fun and easy that you won't even notice how much information you are learning along the way!

Keep in mind that there are still portions of this book that are incomplete and need some love. The beauty of open source publishing is that I can give you an incomplete book and it is still of value to you.

Before we get into all the nitty gritty details, let's start with some ground rules:

Prerequisites

To keep this tutorial small and focused, I'm assuming that you are prepared in the following ways:

1. You have installed the Go Programming Language.
2. You have setup a `GOPATH` by following the How to Write Go Code tutorial.
3. You are somewhat familiar with the basics of Go. (The Go Tour is a pretty good place to start)
4. You have installed all the required packages
5. You have installed the Heroku Toolbelt
6. You have a Heroku account

Required Packages

For the most part we will be using the built in packages from the standard library to build out our web apps. Certain lessons such as Databases, Middleware and URL Routing will require a third party package. Here is a list of all the go packages you will need to install before starting:

Name	Import Path	Description
httprouter	github.com/julienschmidt/httprouter	A high performance HTTP request router that scales well
Negroni	github.com/codegangsta/negroni	Idiomatic HTTP Middleware
Black Friday	github.com/russross/blackfriday	a markdown processor
Render	gopkg.in/unrolled/render.v1	Easy rendering for JSON, XML, and HTML
SQLite3	github.com/mattn/go-sqlite3	sqlite3 driver for go

You can install (or update) these packages by running the following command in your console

```
go get -u <import_path>
```

For instance, if you wish to install Negroni, the following command would be:

```
go get -u github.com/codegangsta/negroni
```

Go Makes Things Simple

If you have built a web application before, you surely know that there are quite a lot of concepts to keep in your head. HTTP, HTML, CSS, JSON, databases, sessions, cookies, forms, middleware, routing and controllers are just a few among the many things your web app *may* need to interact with.

While each one of these things *can be important* in the building of your web applications, not every one of them *is important* for any given app. For instance, a web API may just use JSON as its serialization format, thus making concepts like HTML not relevant for that particular web app.

The Go Way

The Go community understands this dilemma. Rather than rely on large, heavyweight frameworks that try to cover all the bases, Go programmers pull in the bare necessities to get the job done. This minimalist approach to web programming may be off-putting at first, but the result of this effort is a much simpler program in the end.

Go makes things simple, it's as easy as that. If we train ourselves to align with the *"Go way"* of programming for the web, we will end up with more **simple**, **flexible**, and **maintainable** web applications.

Power in Simplicity

As we go through the exercises in this book, I think you will be surprised by how simple some of these programs can be whilst still affording a bunch of functionality.

When sitting down to craft your own web applications in Go, think hard about the components and concepts that your app will be focused on, and use just those pieces. This book will be covering a wide array of web topics, but do not feel obligated to use them all. In the words of our friend Lonestar, *"Take only what you need to survive"*.


TAKE ONLY WHAT YOU NEED TO SURVIVE.

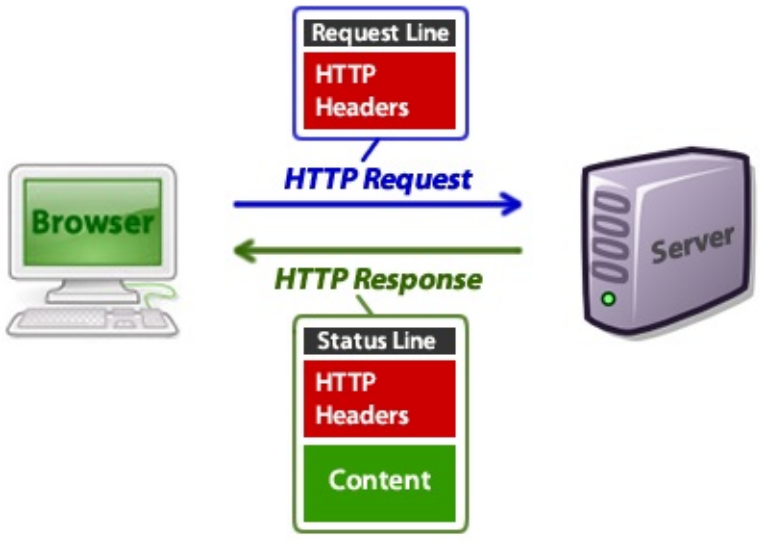
The net/http Package

You have probably heard that Go is fantastic for building web applications of all shapes and sizes. This is partly due to the fantastic work that has been put into making the standard library clean, consistent, and easy to use.

Perhaps one of the most important packages for any budding Go web developer is the `net/http` package. This package allows you to build HTTP servers in Go with its powerful compositional constructs. Before we start coding, let's do an extremely quick overview of HTTP.

HTTP Basics

When we talk about building web applications, we usually mean that we are building HTTP servers. HTTP is a protocol that was originally designed to transport HTML documents from a server to a client web browser. Today, HTTP is used to transport a whole lot more than HTML.



The important thing to notice in this diagram is the two points of interaction between the *Server* and the *Browser*. The *Browser* makes an HTTP request with some information, the *Server* then processes that request and returns a *Response*.

This pattern of request-response is one of the key focal points in building web applications in Go. In fact, the `net/http` package's most important piece is the `http.Handler` Interface.

The http.Handler Interface

As you become more familiar with Go, you will notice how much of an impact *interfaces* make in the design of your programs. The `net/http` interface encapsulates the request-response pattern in one method:

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

Implementors of this interface are expected to inspect and process data coming from the `http.Request` object and write out a response to the `http.ResponseWriter` object.

The `http.ResponseWriter` interface looks like this:

```
type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(int)
}
```

Composing Web Services

Because much of the `net/http` package is built off of well defined interface types, we can (and are expected to) build our web applications with composition in mind. Each `http.Handler` implementation can be thought of as its own web server.

Many patterns can be found in that simple but powerful assumption. Throughout this book we will cover some of these patterns and how we can use them to solve real world problems.

Exercise: 1 Line File Server

Let's solve a real world problem in 1 line of code.

Most of the time people just need to serve static files. Maybe you have a static HTML landing page and just want to serve up some HTML, images, and CSS and call it a day. Sure, you could pull in Apache or Python's `SimpleHTTPServer`, but Apache is too much for this little site and `SimpleHTTPServer` is, well, too slow.

We will begin by creating a new project in our `GOPATH`.

```
cd GOPATH/src
mkdir fileserver && cd fileserver
```

Create a **main.go** with our typical go boilerplate.

```
package main

import "net/http"

func main() {
}
```

All we need to import is the `net/http` package for this to work. Remember that this is all part of the standard library in Go.

Let's write our fileserver code:

```
http.ListenAndServe(":8080", http.FileServer(http.Dir(".")))
```

The `http.ListenAndServe` function is used to start the server, it will bind to the address we gave it (`:8080`) and when it receives an HTTP request, it will hand it off to the `http.Handler` that we supply as the second argument. In our case it is the built-in `http.FileServer`.

The `http.FileServer` function builds an `http.Handler` that will serve an entire directory of files and figure out which file to serve based on the request path. We told the FileServer to serve the current working directory with `http.Dir(".")`.

The entire program looks like this:

```
package main

import "net/http"

func main() {
    http.ListenAndServe(":8080", http.FileServer(http.Dir(".")))
}
```

Let's build and run our fileserver program:

```
go build
./fileserver
```

If we visit `localhost:8080/main.go` we should see the contents of our **main.go** file in our web browser. We can run this program from any directory and serve the tree as a static file server. All in 1 line of Go code.

Creating a Basic Web App

Now that we are done going over the basics of HTTP, let's create a simple but useful web application in Go.

Pulling from our fileserver program that we implemented last chapter, we will implement a Markdown generator using the `github.com/russross/blackfriday` package.

HTML Form

For starters, we will need a basic HTML form for the markdown input:

```
<html>
  <head>
    <link href="/css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <div class="container">
      <div class="page-title">
        <h1>Markdown Generator</h1>
        <p class="lead">Generate your markdown with Go</p>
        <hr />
      </div>

      <form action="/markdown" method="POST">
        <div class="form-group">
          <textarea class="form-control" name="body" cols="30" rows="10"></textarea>
        </div>

        <div class="form-group">
          <input type="submit" class="btn btn-primary pull-right" />
        </div>
      </form>
    </div>
    <script src="/js/bootstrap.min.js"></script>
  </body>
</html>
```

Put this HTML into a file named `index.html` in the "public" folder of our application and the `bootstrap.min.css` from http://getbootstrap.com/ in the "public/css" folder. Notice that the form makes an HTTP POST to the "/markdown" endpoint of our application. We don't actually handle that route right now, so let's add it.

The "/markdown" route

The program to handle the '/markdown' route and serve the public `index.html` file looks like this:

```
package main

import (
    "net/http"

    "github.com/russross/blackfriday"
)

func main() {
    http.HandleFunc("/markdown", GenerateMarkdown)
    http.Handle("/", http.FileServer(http.Dir("public")))
    http.ListenAndServe(":8080", nil)
}

func GenerateMarkdown(rw http.ResponseWriter, r *http.Request) {
    markdown := blackfriday.MarkdownCommon([]byte(r.FormValue("body")))
    rw.Write(markdown)
}
```

Let's break it down into smaller pieces to get a better idea of what is going on.

```
http.HandleFunc("/markdown", GenerateMarkdown)
http.Handle("/", http.FileServer(http.Dir("public")))
```

We are using the `http.HandleFunc` and `http.Handle` methods to define some simple routing for our application. It is important to note that calling `http.Handle` on the "/" pattern will act as a catch-all route, so we define that route last. `http.FileServer` returns an `http.Handler` so we use `http.Handle` to map a pattern string to a handler. The alternative method, `http.HandleFunc`, uses an `http.HandlerFunc` instead of an `http.Handler`. This may be more convenient, to think of handling routes via a function instead of an object.

```
func GenerateMarkdown(rw http.ResponseWriter, r *http.Request) {
    markdown := blackfriday.MarkdownCommon([]byte(r.FormValue("body")))
    rw.Write(markdown)
}
```

Our GenerateMarkdown function implements the standard `http.HandlerFunc` interface and renders HTML from a form field containing markdown-formatted text. In this case, the content is retrieved with `r.FormValue("body")`. It is very common to get input from the

`http.Request` object that the `http.HandlerFunc` receives as an argument. Some other examples of input are the `r.Header`, `r.Body`, and `r.URL` members.

We finalize the request by writing it out to our `http.ResponseWriter`. Notice that we didn't explicitly send a response code. If we write out to the response without a code, the `net/http` package will assume that the response is a `200 OK`. This means that if something did happen to go wrong, we should set the response code via the `rw.WriteHeader()` method.

```
  http.ListenAndServe(":8080", nil)
```

The last bit of this program starts the server, we pass `nil` as our handler, which assumes that the HTTP requests will be handled by the `net/http` packages default `http.ServeMux`, which is configured using `http.Handle` and `http.HandleFunc`, respectively.

And that is all you need to be able to generate markdown as a service in Go. It is a surprisingly small amount of code for the amount of heavy lifting it does. In the next chapter we will learn how to deploy this application to the web using Heroku.

Deployment

Heroku makes deploying applications easy. It is a perfect platform for small to medium size web applications that are willing to sacrifice a little bit of flexibility in infrastructure to gain a fairly pain-free environment for deploying and maintaining web applications.

I am choosing to deploy our web application to Heroku for the sake of this tutorial because in my experience it has been the fastest way to get a web application up and running in no time. Remember that the focus of this tutorial is how to build web applications in Go and not getting caught up in all of the distraction of provisioning, configuring, deploying, and maintaining the machines that our Go code will be run on.

Getting setup

If you don't already have a Heroku account, sign up at id.heroku.com/signup. It's quick, easy and free.

Application management and configuration is done through the Heroku toolbelt, which is a free command line tool maintained by Heroku. We will be using it to create our application on Heroku. You can get it from toolbelt.heroku.com.

Changing the Code

To make sure the application from our last chapter will work on Heroku, we will need to make a few changes. Heroku gives us a `PORT` environment variable and expects our web application to bind to it. Let's start by importing the "os" package so we can grab that `PORT` environment variable:

```
import (
    "net/http"
    "os"

    "github.com/russross/blackfriday"
)
```

Next, we need to grab the `PORT` environment variable, check if it is set, and if it is we should bind to that instead of our hardcoded port (8080).

```
port := os.Getenv("PORT")
if port == "" {
  port = "8080"
}
```

Lastly, we want to bind to that port in our `http.ListenAndServe` call:

```
http.ListenAndServe(":"+port, nil)
```

The final code should look like this:

```
package main

import (
    "net/http"
    "os"

    "github.com/russross/blackfriday"
)

func main() {
    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    http.HandleFunc("/markdown", GenerateMarkdown)
    http.Handle("/", http.FileServer(http.Dir("public")))
    http.ListenAndServe(":"+port, nil)
}

func GenerateMarkdown(rw http.ResponseWriter, r *http.Request) {
    markdown := blackfriday.MarkdownCommon([]byte(r.FormValue("body")))
    rw.Write(markdown)
}
```

Configuration

We need a couple small configuration files to tell Heroku how it should run our application. The first one is the `Procfile`, which allows us to define which processes should be run for our application. By default, Go will name the executable after the containing directory of your main package. For instance, if my web application lived in `GOPATH/github.com/codegangsta/bwag/deployment`, my `Procfile` will look like this:

```
web: deployment
```

Specifically to run Go applications, we need to also specify a `.godir` file to tell Heroku which dir is in fact our package directory.

```
deployment
```

Deployment

Once all these things in place, Heroku makes it easy to deploy.

Initialize the project as a Git repository:

```
git init
git add -A
git commit -m "Initial Commit"
```

Create your Heroku application (specifying the Go buildpack):

```
heroku create -b https://github.com/kr/heroku-buildpack-go.git
```

Push it to Heroku and watch your application be deployed!

```
git push heroku master
```

View your application in your browser:

```
heroku open
```

URL Routing

For some simple applications, the default `http.ServeMux` can take you pretty far. If you need more power in how you parse URL endpoints and route them to the proper handler, you may need to pull in a third party routing framework. For this tutorial, we will use the popular `github.com/julienschmidt/httprouter` library as our router. `github.com/julienschmidt/httprouter` is a great choice for a router as it is a very simple implementation with one of the best performance benchmarks out of all the third party Go routers.

In this example, we will create some routing for a RESTful resource called "posts". Below we define mechanisms to view index, show, create, update, destroy, and edit posts.

```
package main

import (
    "fmt"
    "net/http"

    "github.com/julienschmidt/httprouter"
)

func main() {
    r := httprouter.New()
    r.GET("/", HomeHandler)

    // Posts collection
    r.GET("/posts", PostsIndexHandler)
    r.POST("/posts", PostsCreateHandler)

    // Posts singular
    r.GET("/posts/:id", PostShowHandler)
    r.PUT("/posts/:id", PostUpdateHandler)
    r.GET("/posts/:id/edit", PostEditHandler)

    fmt.Println("Starting server on :8080")
    http.ListenAndServe(":8080", r)
}

func HomeHandler(rw http.ResponseWriter, r *http.Request, p httprouter.Params) {
    fmt.Fprintln(rw, "Home")
}

func PostsIndexHandler(rw http.ResponseWriter, r *http.Request, p httprouter.Params) {
    fmt.Fprintln(rw, "posts index")
}
```

```
func PostsCreateHandler(rw http.ResponseWriter, r *http.Request, p httprouter.Params) {
    fmt.Fprintln(rw, "posts create")
}

func PostShowHandler(rw http.ResponseWriter, r *http.Request, p httprouter.Params) {
    id := p.ByName("id")
    fmt.Fprintln(rw, "showing post", id)
}

func PostUpdateHandler(rw http.ResponseWriter, r *http.Request, p httprouter.Params) {
    fmt.Fprintln(rw, "post update")
}

func PostDeleteHandler(rw http.ResponseWriter, r *http.Request, p httprouter.Params) {
    fmt.Fprintln(rw, "post delete")
}

func PostEditHandler(rw http.ResponseWriter, r *http.Request, p httprouter.Params) {
    fmt.Fprintln(rw, "post edit")
}
```

Exercises

1. Explore the documentation for `github.com/julienschmidt/httprouter`.
2. Find out how well `github.com/julienschmidt/httprouter` plays nicely with existing `http.Handler`s like `http.FileServer`
3. `httprouter` has a very simple interface. Explore what kind of abstractions can be built on top of this fast router to make building things like RESTful routing easier.

Middleware

If you have some code that needs to be run for every request, regardless of the route that it will eventually end up invoking, you need some way to stack `http.Handlers` on top of each other and run them in sequence. This problem is solved elegantly through middleware packages. Negroni is a popular middleware package that makes building and stacking middleware very easy while keeping the composable nature of the Go web ecosystem intact.

Negroni comes with some default middleware such as Logging, Error Recovery, and Static file serving. So out of the box Negroni will provide you with a lot of value without a lot of overhead.

The example below shows how to use a Negroni stack with the built in middleware and how to create your own custom middleware.

```
package main

import (
    "log"
    "net/http"

    "github.com/codegangsta/negroni"
)

func main() {
    // Middleware stack
    n := negroni.New(
        negroni.NewRecovery(),
        negroni.HandlerFunc(MyMiddleware),
        negroni.NewLogger(),
        negroni.NewStatic(http.Dir("public")),
    )

    n.Run(":8080")
}

func MyMiddleware(rw http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
    log.Println("Logging on the way there...")

    if r.URL.Query().Get("password") == "secret123" {
        next(rw, r)
    } else {
        http.Error(rw, "Not Authorized", 401)
    }

    log.Println("Logging on the way back...")
}
```

Exercises

1. Think of some cool middleware ideas and try to implement them using Negroni.
2. Explore how Negroni can be composed with `github.com/gorilla/mux` using the `http.Handler` interface.
3. Play with creating Negroni stacks for certain groups of routes instead of the entire application.

Rendering

Rendering is the process of taking data from your application or database and presenting it for the client. The client can be a browser that renders HTML, or it can be another application that consumes JSON as its serialization format. In this chapter we will learn how to render both of these formats using the methods that Go provides for us in the standard library.

JSON

JSON is quickly becoming the ubiquitous serialization format for web APIs, so it may be the most relevant when learning how to build web apps using Go. Fortunately, Go makes it simple to work with JSON -- it is extremely easy to turn existing Go structs into JSON using the `encoding/json` package from the standard library.

```
package main

import (
    "encoding/json"
    "net/http"
)

type Book struct {
    Title  string `json:"title"`
    Author string `json:"author"`
}

func main() {
    http.HandleFunc("/", ShowBooks)
    http.ListenAndServe(":8080", nil)
}

func ShowBooks(w http.ResponseWriter, r *http.Request) {
    book := Book{"Building Web Apps with Go", "Jeremy Saenz"}

    js, err := json.Marshal(book)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    w.Write(js)
}
```

Exercises

1. Read through the JSON API docs and find out how to rename and ignore fields for JSON serialization.
2. Instead of using the `json.Marshal` method, try using the `json.Encoder` API.
3. Figure our how to pretty print JSON with the `encoding/json` package.

HTML Templates

Serving HTML is an important job for some web applications. Go has one of my favorite templating languages to date. Not for its features, but for its simplicity and out of the box security. Rendering HTML templates is almost as easy as rendering JSON using the 'html/template' package from the standard library. Here is what the source code for rendering HTML templates looks like:

```
package main

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

type Book struct {
    Title  string
    Author string
}

func main() {
    http.HandleFunc("/", ShowBooks)
    http.ListenAndServe(":8080", nil)
}

func ShowBooks(w http.ResponseWriter, r *http.Request) {
    book := Book{"Building Web Apps with Go", "Jeremy Saenz"}

    fp := path.Join("templates", "index.html")
    tmpl, err := template.ParseFiles(fp)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    if err := tmpl.Execute(w, book); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

This is the following template we will be using. It should be placed in a `templates/index.html` file in the directory your program is run from:

```
<html>
  <h1>{{ .Title }}</h1>
  <h3>by {{ .Author }}</h3>
</html>
```

Exercises

1. Look through the docs for `text/template` and `html/template` package. Play with the templating language a bit to get a feel for its goals, strengths, and weaknesses.
2. In the example we parse the files on every request, which can be a lot of performance overhead. Experiment with parsing the files at the beginning of your program and executing them in your `http.Handler` (hint: make use of the `Copy()` method on `html.Template`).
3. Experiment with parsing and using multiple templates.

Using the render package

If you want rendering JSON and HTML to be even simpler, there is the `github.com/unrolled/render` package. This package was inspired by the `martini-contrib/render` package and is my goto when it comes to rendering data for presentation in my web applications.

```
package main

import (
    "net/http"

    "gopkg.in/unrolled/render.v1"
)

func main() {
    r := render.New(render.Options{})
    mux := http.NewServeMux()

    mux.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
        w.Write([]byte("Welcome, visit sub pages now."))
    })

    mux.HandleFunc("/data", func(w http.ResponseWriter, req *http.Request) {
        r.Data(w, http.StatusOK, []byte("Some binary data here."))
    })

    mux.HandleFunc("/json", func(w http.ResponseWriter, req *http.Request) {
        r.JSON(w, http.StatusOK, map[string]string{"hello": "json"})
    })

    mux.HandleFunc("/html", func(w http.ResponseWriter, req *http.Request) {
        // Assumes you have a template in ./templates called "example.tmpl"
        // $ mkdir -p templates && echo "<h1>Hello {{.}}.</h1>" > templates/example.tmpl
        r.HTML(w, http.StatusOK, "example", nil)
    })

    http.ListenAndServe(":8080", mux)
}
```

Exercises

1. Have fun playing with all of the options available when calling `render.New()`
2. Try using the `.yield` helper function (with the curly braces) and a layout with HTML

templates.

Testing

Testing is an important part of any application. There are two approaches we can take to testing Go web applications. The first approach is a unit-test style approach. The other is more of an end-to-end approach. In this chapter we'll cover both approaches.

Unit Testing

Unit testing allows us to test a `http.HandlerFunc` directly without running any middleware, routers, or any other type of code that might otherwise wrap the function.

```
package main

import (
    "fmt"
    "net/http"
)

func HelloWorld(res http.ResponseWriter, req *http.Request) {
    fmt.Fprint(res, "Hello World")
}

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

This is the test file. It should be placed in the same directory as your application and name `main_test.go`.

```
package main

import (
    "net/http"
    "net/http/httptest"
    "testing"
)

func Test_HelloWorld(t *testing.T) {
    req, err := http.NewRequest("GET", "http://example.com/foo", nil)
    if err != nil {
        t.Fatal(err)
    }

    res := httptest.NewRecorder()
    HelloWorld(res, req)

    exp := "Hello World"
    act := res.Body.String()
    if exp != act {
        t.Fatalf("Expected %s gog %s", exp, act)
    }
}
```

Exercises

1. Change the output of `HelloWorld` to print a parameter and then test that the parameter is rendered.
2. Create a POST request and test that the request is properly handled.

End To End Testing

End to end allows us to test applications through the whole request cycle. Where unit testing is meant to just test a particular function, end to end tests will run the middleware, router, and other that a request my pass through.

```
package main

import (
    "fmt"
    "net/http"

    "github.com/codegangsta/negroni"
    "github.com/julienschmidt/httprouter"
)

func HelloWorld(res http.ResponseWriter, req *http.Request, p httprouter.Params) {
    fmt.Fprint(res, "Hello World")
}

func App() http.Handler {
    n := negroni.Classic()

    m := func(res http.ResponseWriter, req *http.Request, next http.HandlerFunc) {
        fmt.Fprint(res, "Before...")
        next(res, req)
        fmt.Fprint(res, "...After")
    }
    n.Use(negroni.HandlerFunc(m))

    r := httprouter.New()

    r.GET("/", HelloWorld)
    n.UseHandler(r)
    return n
}

func main() {
    http.ListenAndServe(":3000", App())
}
```

This is the test file. It should be placed in the same directory as your application and name `main_test.go`.

```
package main

import (
    "io/ioutil"
    "net/http"
    "net/http/httptest"
    "testing"
)

func Test_App(t *testing.T) {
    ts := httptest.NewServer(App())
    defer ts.Close()

    res, err := http.Get(ts.URL)
    if err != nil {
        t.Fatal(err)
    }

    body, err := ioutil.ReadAll(res.Body)
    res.Body.Close()

    if err != nil {
        t.Fatal(err)
    }

    exp := "Before...Hello World...After"

    if exp != string(body) {
        t.Fatalf("Expected %s got %s", exp, body)
    }
}
```

Exercises

1. Create another piece of middleware that mutates the status of the request.
2. Create a POST request and test that the request is properly handled.

Controllers

Controllers are a fairly familiar topic in other web development communities. Since most web developers rally around the mighty net/http interface, not many controller implementations have caught on strongly. However, there is great benefit in using a controller model. It allows for clean, well defined abstractions above and beyond what the net/http handler interface can alone provide.

Handler Dependencies

In this example we will experiment with building our own controller implementation using some standard features in Go. But first, lets start with the problems we are trying to solve. Say we are using the `render` library that we talked about in previous chapters:

```
var Render = render.New(render.Options{})
```

If we want our `http.Handler`s to be able access our `render.Render` instance, we have a couple options.

1. Use a global variable: This isn't too bad for small programs, but when the program gets larger it quickly becomes a maintenance nightmare.

2. Pass the variable through a closure to the http.Handler: This is a great idea, and we should be using it most of the time. The implementation ends up looking like this:

```
func MyHandler(r *render.Render) http.Handler {
  return http.HandlerFunc(func(rw http.ResponseWriter, r *http.Request) {
    // now we can access r
  })
}
```

Case for Controllers

When your program grows in size, you will start to notice that many of your `http.Handler`s will share the same dependencies and you will have a lot of these closurized `http.Handlers` with the same arguments. The way I like to clean this up is to write a little base controller implementation that affords me a few wins:

1. Allows me to share the dependencies across `http.Handler`s that have similar goals or

concepts.
2. Avoids global variables and functions for easy testing/mocking.
3. Gives me a more centralized and Go-like mechanism for handling errors.

The great part about controllers is that it gives us all these things without importing an external package! Most of this functionality comes from clever use of the Go feature set, namely Go structs and embedding. Let's take a look at the implementation.

```
package main

import "net/http"

// Action defines a standard function signature for us to use when creating
// controller actions. A controller action is basically just a method attached to
// a controller.
type Action func(rw http.ResponseWriter, r *http.Request) error

// This is our Base Controller
type AppController struct{}

// The action function helps with error handling in a controller
func (c *AppController) Action(a Action) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, r *http.Request) {
        if err := a(rw, r); err != nil {
            http.Error(rw, err.Error(), 500)
        }
    })
}
```

Thats it! That is all the implementation that we need to have the power of controllers at our fingertips. All we have left to do is implement an example controller:

```
package main

import (
    "net/http"

    "gopkg.in/unrolled/render.v1"
)

type MyController struct {
    AppController
    *render.Render
}

func (c *MyController) Index(rw http.ResponseWriter, r *http.Request) error {
    c.JSON(rw, 200, map[string]string{"Hello": "JSON"})
    return nil
}

func main() {
    c := &MyController{Render: render.New(render.Options{})}
    http.ListenAndServe(":8080", c.Action(c.Index))
}
```

Exercises

1. Extend `MyController` to have multiple actions for different routes in your application.
2. Play with more controller implementations, get creative.
3. Override the `Action` method on `MyController` to render a error HTML page.

Databases

One of the most asked questions I get about web development in Go is how to connect to a SQL database. Thankfully, Go has a fantastic SQL package in the standard library that allows us to use a whole slew of drivers for different SQL databases. In this example we will connect to a SQLite database, but the syntax (minus some small SQL semantics) is the same for a MySQL or PostgreSQL database.

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "net/http"

    _ "github.com/mattn/go-sqlite3"
)

func main() {
    db := NewDB()
    log.Println("Listening on :8080")
    http.ListenAndServe(":8080", ShowBooks(db))
}

func ShowBooks(db *sql.DB) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, r *http.Request) {
        var title, author string
        err := db.QueryRow("select title, author from books").Scan(&title, &author)
        if err != nil {
            panic(err)
        }

        fmt.Fprintf(rw, "The first book is '%s' by '%s'", title, author)
    })
}

func NewDB() *sql.DB {
    db, err := sql.Open("sqlite3", "example.sqlite")
    if err != nil {
        panic(err)
    }

    _, err = db.Exec("create table if not exists books(title text, author text)")
    if err != nil {
        panic(err)
    }

    return db
}
```

Exercises

1. Make use of the `Query` function on our `sql.DB` instance to extract a collection of rows and map them to structs.
2. Add the ability to insert new records into our database by using an HTML form.

3. `go get github.com/jmoiron/sqlx` and observe the improvements made over the existing database/sql package in the standard library.

Tips and Tricks

Wrap a http.HandlerFunc closure

Sometimes you want to pass data to a http.HandlerFunc on initialization. This can easily be done by creating a closure of the `http.HandlerFunc`:

```
func MyHandler(database *sql.DB) http.Handler {
  return http.HandlerFunc(func(rw http.ResponseWriter, r *http.Request) {
    // you now have access to the *sql.DB here
  })
}
```

Using `gorilla/context` for request-specific data

It is pretty often that we need to store and retrieve data that is specific to the current HTTP request. Use `gorilla/context` to map values and retrieve them later. It contains a global mutex on a map of request objects.

```
func MyHandler(w http.ResponseWriter, r *http.Request) {
    val := context.Get(r, "myKey")

    // returns ("bar", true)
    val, ok := context.GetOk(r, "myKey")
    // ...

}
```

Moving Forward

You've done it! You have gotten a taste of Go web development tools and libraries. At the time of this writing, this book is still in flux. This section is reserved for more Go web resources to continue your learning.