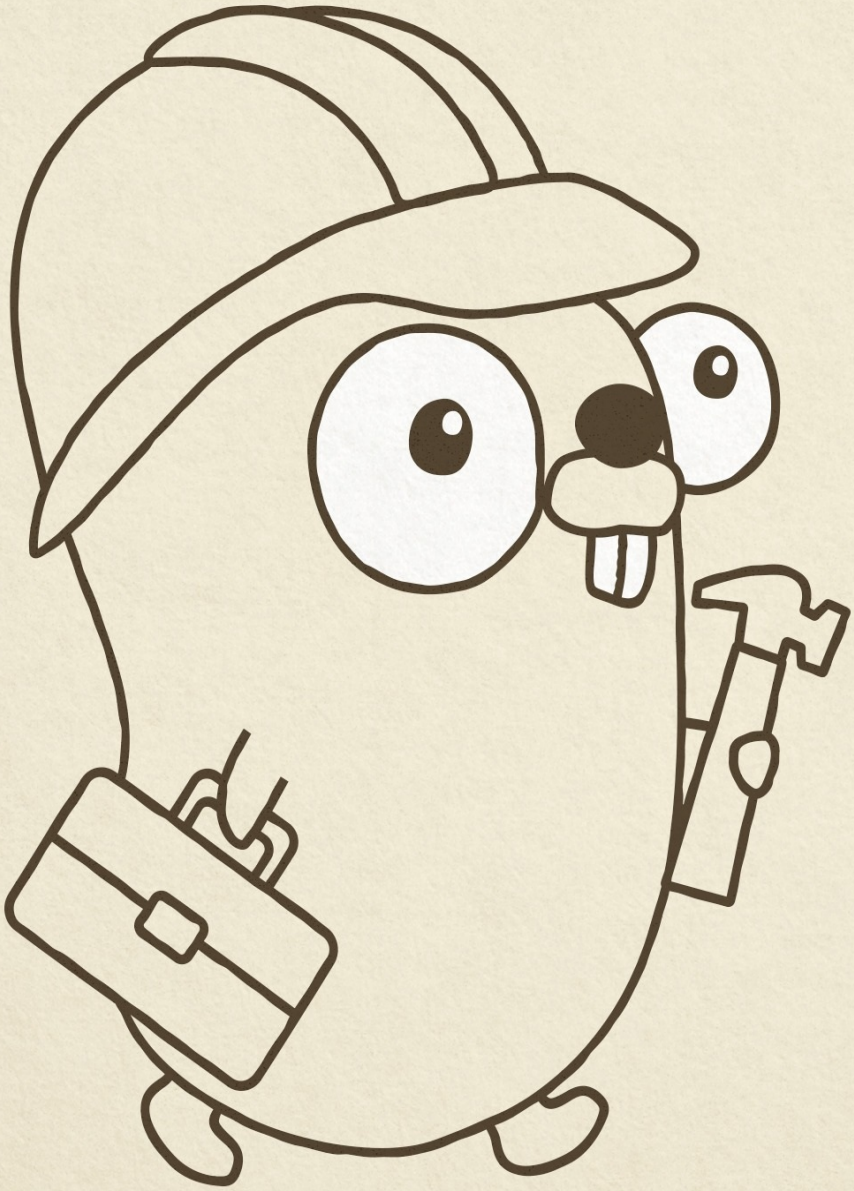
BUILDING WEB APPS WITH GO



Jeremy Saenz

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!

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, URL Routing, and Controllers 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
Gorilla Mux	github.com/gorilla/mux
Negroni	github.com/codegangsta/negroni
Controller	github.com/codegangsta/controller
Black Friday	github.com/russross/blackfriday
Render	gopkg.in/unrolled/render.v1
SQLite3	github.com/mattn/go-sqlite3

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 it's 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"*.

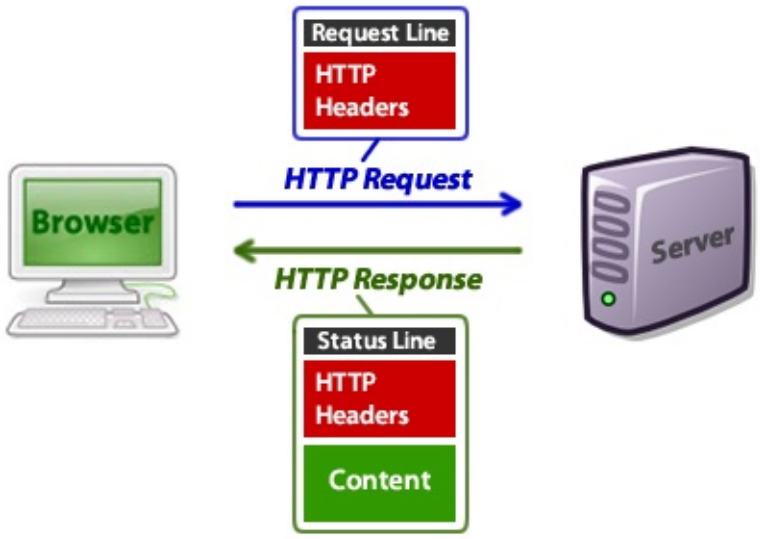
The net/http Package

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 it's 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 it's 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:

```
<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>
```

Put this HTML into a `index.html` in the "public" folder of our application. 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` to use an `http.HandlerFunc` instead of and `http.Handler`. This can be an easier convenience 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 markdown input. In this case it happens to be a `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 is the `r.Header` `r.Body` and `r.URL`.

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 necessarily get 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 here. Signup is 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 install it here

Changing the Code

To make sure the application from our last chapter will work on Heroku, we will have to make a couple 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/codegangtsa/deploygo`, my `Procfile` will look like this:

```
web: deploygo
```

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

```
deploygo
```

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 git 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/gorilla/mux` library as our router. 'github.com/gorilla/mux' is a great choice for a router as it has an interface that is familiar for `http.ServeMux` users, yet has a ton of extra features built around the idea of finding the right http.Handler for the given URL path.

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. Thankfully with `github.com/gorilla/mux`, we don't have to do too much copy-pasting to accomplish this.

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
)

func main() {
    r := mux.NewRouter().StrictSlash(true)
    r.HandleFunc("/", HomeHandler)

    // Posts collection
    posts := r.Path("/posts").Subrouter()
    posts.Methods("GET").HandlerFunc(PostsIndexHandler)
    posts.Methods("POST").HandlerFunc(PostsCreateHandler)

    // Posts singular
    post := r.PathPrefix("/posts/{id}/").Subrouter()
    post.Methods("GET").Path("/edit").HandlerFunc(PostEditHandler)
    post.Methods("GET").HandlerFunc(PostShowHandler)
    post.Methods("PUT", "POST").HandlerFunc(PostUpdateHandler)
    post.Methods("DELETE").HandlerFunc(PostDeleteHandler)

    fmt.Println("Starting server on :3000")
    http.ListenAndServe(":3000", r)
}

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

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

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

func PostShowHandler(rw http.ResponseWriter, r *http.Request) {
    id := mux.Vars(r)["id"]
    fmt.Fprintln(rw, "showing post", id)
}

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

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

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

Exercises

1. Explore the documentation for `github.com/gorilla/mux`.
2. Play with the different chain methods for creating filters and subrouters.
3. Find how well 'github.com/gorilla/mux' plays nicely with existing `http.Handler`'s like `http.FileServer`

Middleware

If you have pieces of code that need 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 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 it's 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 I figure JSON would be the most relevant when learning how to build web apps using Go. Fortunately, JSON is also very easy to work with in Go. 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 Profile struct {
  Name    string
  Hobbies []string
}

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

func ProfileHandler(w http.ResponseWriter, r *http.Request) {
  profile := Profile{"Alex", []string{"snowboarding", "programming"}}

  js, err := json.Marshal(profile)
  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.

```
package main

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

type Profile struct {
  Name    string
  Hobbies []string
}

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

func foo(w http.ResponseWriter, r *http.Request) {
  profile := Profile{"Alex", []string{"snowboarding", "programming"}}

  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, profile); err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
  }
}
```

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 it's 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 execute 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 application.

```
package main

import (
    "encoding/xml"
    "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("0.0.0.0:3000", mux)
}
```

Exercises

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

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"
    "log"

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

func NewDB() *sql.DB {
    db, err := sql.Open("sqlite3", "example.sqlite")
    checkErr(err)

    _, err = db.Exec("create table if not exists posts(title text, body text)")
    checkErr(err)

    return db
}

func checkErr(err error) {
    if err != nil {
        log.Fatalln(err)
    }
}
```

Exercises

1. Now that we have written the code to initialize the database, create some HTTP handlers to add data to the database. Use an html form to POST data to the endpoint and insert it into the sqlite database.
2. Make use of the `Query` function on our `sql.DB` instance to extract a collection of rows and map them to structs.
3. go get github.com/jmoiron/sqlx and observe the improvements made over the existing database/sql package in the standard library.

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.

In this example we will experiment with the controller pattern using `github.com/codegangsta/controller` to construct a new controller instance on every request. This allows us to avoid use of global variables, state, and logic by moving domain-specific logic into it's respective controller implementation.

```
package view

import (
    "net/http"

    "github.com/codegangsta/controller"
    "gopkg.in/unrolled/render.v1"
)

var Renderer = render.New(render.Options{})

type ViewController struct {
    controller.Base
    View     map[string]interface{}
    renderer *render.Render
}

func (c *ViewController) Init(rw http.ResponseWriter, r *http.Request) error {
    c.renderer = Renderer
    c.View = make(map[string]interface{})
    return c.Base.Init(rw, r)
}

func (c *ViewController) HTML(code int, name string, opts ...render.HTMLOptions) {
    c.renderer.HTML(c.ResponseWriter, code, name, c.View, opts...)
}
```

Exercises

1. Extend this ViewController implementation to render JSON as well as HTML.
2. Play with more controller implementations, get creative.
3. Create more controllers that embed the ViewController struct.

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 an 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.

Table of Contents