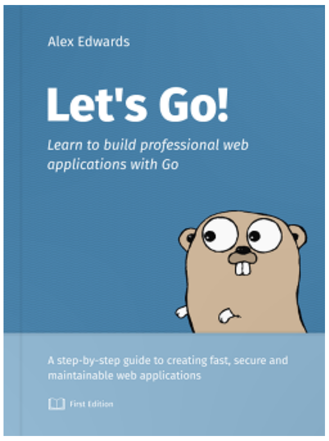
Alex Edwards
Let's Go!
Learn to build professional web applications with Go
A step-by-step guide to creating fast, secure and maintainable web applications
First Edition

Serving Static Sites with Go

Last updated: 26th January 2017

Filed under: golang tutorial

I've recently moved the site you're reading right now from a Sinatra application to an (almost) static one served by Go. While it's fresh in my head, here's an explanation of principles behind creating and serving static sites with Go.

Let's begin with a simple but real-world example: serving vanilla HTML and CSS files from a particular location.

Start by creating a directory to hold the project:

```
$ mkdir static-site
$ cd static-site
```

Along with an `app.go` file to hold our code, and some sample HTML and CSS files in a `static` directory.

```
$ touch app.go
$ mkdir -p static/stylesheets
$ touch static/example.html static/stylesheets/main.css
```

```
File: static/example.html
```

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>A static page</title>
  <link rel="stylesheet" href="/stylesheets/main.css">
</head>
<body>
  <h1>Hello from a static page</h1>
</body>
</html>
```

```
File: static/stylesheets/main.css
```

```
body {color: #c0392b}
```

Once those files are created, the code we need to get up and running is wonderfully compact:

```
File: app.go
```

```
package main

import (
  "log"
  "net/http"
)

func main() {
  fs := http.FileServer(http.Dir("static"))
  http.Handle("/", fs)

  log.Println("Listening...")
  http.ListenAndServe(":3000", nil)
}
```

Let's step through this.

First we use the FileServer function to create a handler which responds to all HTTP requests with the contents of a given FileSystem. For our FileSystem we're using the `static` directory relative to our application, but you could use any other directory on your machine (or indeed any object that implements the FileSystem interface). Next we use the Handle function to register our FileServer as the handler for all requests, and launch the server listening on port 3000.

It's worth pointing out that in Go the pattern `"/"` matches all request paths, rather than just the empty path.

Go ahead and run the application:

```
$ go run app.go
Listening...
```

And open localhost:3000/example.html in your browser. You should see the HTML page we made with a big red heading.

Almost-Static Sites

If you're creating a lot of static HTML files by hand, it can be tedious to keep repeating boilerplate content. Let's explore using the Template package to put shared markup in a *layout* file.

At the moment *all* requests are being handled by our FileServer. Let's make a slight adjustment to our application so it only handles request paths that begin with the pattern `/static/` instead.

```
File: app.go
```

```
...
func main() {
  fs := http.FileServer(http.Dir("static"))
  http.Handle("/static/", http.StripPrefix("/static/", fs))

  log.Println("Listening...")
  http.ListenAndServe(":3000", nil)
}
```

Notice that because our `static` directory is set as the root of the FileSystem, we need to strip off the `/static/` prefix from the request path *before* searching the FileSystem for the given file. We do this using the StripPrefix function.

If you restart the application, you should find the CSS file we made earlier available at localhost:3000/static/stylesheets/main.css.

Now let's create a `templates` directory, containing a `layout.html` file with shared markup, and an `example.html` file with some page-specific content.

```
$ mkdir templates
$ touch templates/layout.html templates/example.html
```

```
File: templates/layout.html
```

```
{{define "layout"}}
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{template "title"}}</title>
  <link rel="stylesheet" href="/static/stylesheets/main.css">
</head>
<body>
  {{template "body"}}
</body>
</html>
{{end}}
```

```
File: templates/example.html
```

```
{{define "title"}}A templated page{{end}}

{{define "body"}}
<h1>Hello from a templated page</h1>
{{end}}
```

If you've used templating in other web frameworks or languages before, this should hopefully feel familiar.

Go templates – in the way we're using them here – are essentially just named text blocks surrounded by `{{define}}` and `{{end}}` tags. Templates can be embedded into each other, as we do above where the `layout` template embeds both the `title` and `body` templates.

Let's update the application code to use these:

```
File: app.go
```

```go
package main

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

func main() {
  fs := http.FileServer(http.Dir("static"))
  http.Handle("/static/", http.StripPrefix("/static/", fs))

  http.HandleFunc("/", serveTemplate)

  log.Println("Listening...")
  http.ListenAndServe(":3000", nil)
}

func serveTemplate(w http.ResponseWriter, r *http.Request) {
  lp := filepath.Join("templates", "layout.html")
  fp := filepath.Join("templates", filepath.Clean(r.URL.Path))

  tmpl, _ := template.ParseFiles(lp, fp)
  tmpl.ExecuteTemplate(w, "layout", nil)
}
```

So what's changed here?

First we've added the `html/template` and `path` packages to the import statement.

We've then specified that all the requests *not* picked up by the static file server should be handled with a new `serveTemplate` function (if you were wondering, Go matches patterns based on length, with longer patterns take precedence over shorter ones).

In the `serveTemplate` function, we build paths to the layout file and the template file corresponding with the request. Rather than manual concatenation we use filepath.Join, which has the advantage joining paths using the correct separator for your OS.

Importantly, because the URL path is untrusted user input, we use filepath.Clean to sanitise the URL path before using it.

(Note that even though filepath.Join automatically runs the *joined path* through filepath.Clean, to help prevent directory traversal attacks you need to manually sanitise any untrusted inputs *before* joining them.)

We then use the ParseFiles function to bundle the requested template and layout into a *template set*. Finally, we use the ExecuteTemplate function to render a named template in the set, in our case the `layout` template.

Restart the application:

```
$ go run app.go
Listening...
```

And open localhost:3000/example.html in your browser. If you look at the source you should find the markup from both templates merged together. You might also notice that the `Content-Type` and `Content-Length` headers have automatically been set for us.

Lastly, let's make the code a bit more robust. We should:

- Send a `404` response if the requested template doesn't exist.
- Send a `404` response if the requested template path is a directory.
- Send a `500` response if the `template.ParseFiles` or `template.ExecuteTemplate` functions throw an error, and log the detailed error message.

```
File: app.go
```

```
package main
```

```
import (
  "html/template"
  "log"
  "net/http"
  "os"
  "path/filepath"
)

func main() {
  fs := http.FileServer(http.Dir("static"))
  http.Handle("/static/", http.StripPrefix("/static/", fs))
  http.HandleFunc("/", serveTemplate)

  log.Println("Listening...")
  http.ListenAndServe(":3000", nil)
}

func serveTemplate(w http.ResponseWriter, r *http.Request) {
  lp := filepath.Join("templates", "layout.html")
  fp := filepath.Join("templates", filepath.Clean(r.URL.Path))

  // Return a 404 if the template doesn't exist
  info, err := os.Stat(fp)
  if err != nil {
    if os.IsNotExist(err) {
      http.NotFound(w, r)
      return
    }
  }

  // Return a 404 if the request is for a directory
  if info.IsDir() {
    http.NotFound(w, r)
    return
  }

  tmpl, err := template.ParseFiles(lp, fp)
  if err != nil {
    // Log the detailed error
    log.Println(err.Error())
    // Return a generic "Internal Server Error" message
    http.Error(w, http.StatusText(500), 500)
    return
  }

  if err := tmpl.ExecuteTemplate(w, "layout", nil); err != nil {
    log.Println(err.Error())
```

```
        http.Error(w, http.StatusText(500), 500)
    }
}
```

If you enjoyed this blog post, don't forget to check out my new book about how to build professional web applications with Go!

Follow me on Twitter @ajmedwards.