An Introduction to Handlers and Servemuxes in Go

Last updated: December 6th, 2021

bit like controllers. Generally speaking, they're responsible for carrying out your application logic and writing response headers and bodies.

Whereas a servemux (also known as a *router*) stores a mapping between the predefined URL paths for your application and the corresponding handlers. Usually you have one servemux for your application containing all your routes.

Go's `net/http` package ships with the simple but effective `http.ServeMux` servemux, plus a few functions to generate common handlers including `http.FileServer()`, `http.NotFoundHandler()` and `http.RedirectHandler()`.

Let's take a look at a simple (but slightly contrived!) example which uses these:

```
$ mkdir example
$ cd example
$ go mod init example.com
$ touch main.go
```

```
↳ File: main.go
```

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // Use the http.NewServeMux() function to create an empty servemux.
    mux := http.NewServeMux()

    // Use the http.RedirectHandler() function to create a handler which 307
    // redirects all requests it receives to http://example.org.
    rh := http.RedirectHandler("http://example.org", 307)

    // Next we use the mux.Handle() function to register this with our new
    // servemux, so it acts as the handler for all incoming requests with the URL
    // path /foo.
    mux.Handle("/foo", rh)
```

```
    log.Print("Listening...")

    // Then we create a new server and start listening for incoming requests
    // with the http.ListenAndServe() function, passing in our servemux for it to
    // match requests against as the second argument.
    http.ListenAndServe(":3000", mux)
}
```

Go ahead and run the application:

```
$ go run main.go
2021/12/06 15:09:43 Listening...
```

And if you make a request to `http://localhost:3000/foo` you should find that it gets successfully redirected like so:

```
$ curl -IL localhost:3000/foo
HTTP/1.1 307 Temporary Redirect
Content-Type: text/html; charset=utf-8
Location: http://example.org
Date: Mon, 06 Dec 2021 14:10:18 GMT

HTTP/1.1 200 OK
Content-Encoding: gzip
Accept-Ranges: bytes
Age: 254488
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Mon, 06 Dec 2021 14:10:18 GMT
Etag: "3147526947+gzip"
Expires: Mon, 13 Dec 2021 14:10:18 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (dcb/7EEF)
X-Cache: HIT
Content-Length: 648
```

Whereas all other requests should be met with a `404 Not Found` error response.

```
$ curl -IL localhost:3000/bar
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Mon, 06 Dec 2021 14:22:51 GMT
Content-Length: 19
```

Custom handlers

The handlers that ship with `net/http` are useful, but most of the time when building a web application you'll want to use your own custom handlers instead. *So how do you do that?*

The first thing to explain is that *anything in Go can be a handler so long as it satisfies the* `http.Handler` *interface*, which looks like this:

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

If you're not familiar with interfaces in Go I've written an explanation here, but in simple terms all it means is that a handler *must* have a `ServeHTTP()` method with the following signature:

```
ServeHTTP(http.ResponseWriter, *http.Request)
```

To help demonstrate, let's create a custom handler which responds with the current time in a specific format. Like this:

```
type timeHandler struct {
    format string
}
```

```
func (th timeHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    tm := time.Now().Format(th.format)
    w.Write([]byte("The time is: " + tm))
}
```

The exact code here isn't too important.

All that really matters is that we have an object (in this case it's a `timeHandler` 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.

Let's try this out in a concrete example:

```
↳ File: main.go

package main

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

type timeHandler struct {
    format string
}

func (th timeHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    tm := time.Now().Format(th.format)
    w.Write([]byte("The time is: " + tm))
}

func main() {
    mux := http.NewServeMux()

    // Initialise the timeHandler in exactly the same way we would any normal
    // struct.
```

```
    th := timeHandler{format: time.RFC1123}

    // Like the previous example, we use the mux.Handle() function to register
    // this with our ServeMux.
    mux.Handle("/time", th)

    log.Print("Listening...")
    http.ListenAndServe(":3000", mux)
}
```

Run the application, then go ahead and try making a request to `http://localhost:3000/time`. You should get a response containing the current time, similar to this:

```
$ curl localhost:3000/time
The time is: Mon, 06 Dec 2021 15:33:21 CET
```

Let's step through what's happening here:

1. When our Go server receives an incoming HTTP request, it hands it off to our servemux (the one that we passed to the `http.ListenAndServe()` function).
2. The servemux then looks up the appropriate handler based on the request path (in this case, the `/time` path maps to our `timeHandler` handler).
3. The serve mux then calls the `ServeHTTP()` method of the handler, which in turn writes out the HTTP response.

The eagle-eyed of you might have also noticed something interesting: the signature for the `http.ListenAndServe()` function is `ListenAndServe(addr string, handler Handler)`, but we passed a servemux as the second argument.

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

For me it simplifies things to think of `http.ServeMux` as *just being a special kind of handler*, which instead of providing a response itself passes the request on to

a second handler. This isn't as much of a leap as it first sounds — chaining handlers together is very commonplace in Go.

Functions as handlers

For simple cases (like the example above) defining new a custom type just to make a handler feels a bit verbose. Fortunately, we can rewrite the handler as a simple function instead:

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    tm := time.Now().Format(time.RFC1123)
    w.Write([]byte("The time is: " + tm))
}
```

Now, if you've been following along, you're probably looking at that and wondering: *How can that be a handler? It doesn't have a* `ServeHTTP()` *method.*

And you'd be correct. This function itself **is not a handler**. But we can *coerce* it into being a handler by converting it to a `http.HandlerFunc` type.

Basically, any function which has the signature `func(http.ResponseWriter, *http.Request)` can be converted into a `http.HandlerFunc` type. This is useful because `http.HandlerFunc` objects come with an inbuilt `ServeHTTP()` method which — rather cleverly and conveniently — executes the content of the original function.

If that sounds confusing, try taking a look at the relevant source code. You'll see that it's a very succinct way of making a function satisfy the `http.Handler` interface.

Let's reproduce the our application using this technique:

```
↳ File: main.go
```

```
package main

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

func timeHandler(w http.ResponseWriter, r *http.Request) {
    tm := time.Now().Format(time.RFC1123)
    w.Write([]byte("The time is: " + tm))
}

func main() {
    mux := http.NewServeMux()

    // Convert the timeHandler function to a http.HandlerFunc type.
    th := http.HandlerFunc(timeHandler)

    // And add it to the ServeMux.
    mux.Handle("/time", th)

    log.Print("Listening...")
    http.ListenAndServe(":3000", mux)
}
```

In fact, converting a function to a `http.HandlerFunc` type and then adding it to a servemux like this is so common that Go provides a shortcut: the `mux.HandleFunc()` method. You can use this like so:

```
func main() {
    mux := http.NewServeMux()

    mux.HandleFunc("/time", timeHandler)

    log.Print("Listening...")
```

```
    http.ListenAndServe(":3000", mux)
}
```

Passing variables to handlers

Most of the time using a function as a handler like this works well. But there is a bit of a limitation when things start getting more complex.

You've probably noticed that, unlike the method before, we've had to hardcode the time format in the `timeHandler` function. *What happens when you want to pass information or variables from `main()` to a handler?*

A neat approach is to put our handler logic into a closure, and *close over* the variables we want to use, like this:

```
↳ File: main.go

package main

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

func timeHandler(format string) http.Handler {
    fn := func(w http.ResponseWriter, r *http.Request) {
        tm := time.Now().Format(format)
        w.Write([]byte("The time is: " + tm))
    }
    return http.HandlerFunc(fn)
}

func main() {
    mux := http.NewServeMux()

    th := timeHandler(time.RFC1123)
```

```
    mux.Handle("/time", th)

    log.Print("Listening...")
    http.ListenAndServe(":3000", mux)
}
```

The `timeHandler()` function now has a subtly different role. Instead of coercing the function into a handler (like we did previously), we are now using it to *return a handler*. There's two key elements to making this work.

First it creates `fn`, an anonymous function which accesses — or closes over — the `format` variable forming a *closure*. Regardless of what we do with the 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 it'll always have access to the `format` variable.

Secondly our closure has the signature `func(http.ResponseWriter, *http.Request)`. As you may remember from a moment ago, this means that we can convert it into a `http.HandlerFunc` type (so that it satisfies the `http.Handler` interface). Our `timeHandler()` function then returns this converted closure.

In this example we've just been passing a simple string to a handler. But in a real-world application you could use this method to pass database connection, template map, or any other application-level context. It's a good alternative to using global variables, and has the added benefit of making neat self-contained handlers for testing.

You might also see this same pattern written as:

```
func timeHandler(format string) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        tm := time.Now().Format(format)
        w.Write([]byte("The time is: " + tm))
    })
}
```

Or using an implicit conversion to the `http.HandlerFunc` type on return:

```
func timeHandler(format string) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        tm := time.Now().Format(format)
        w.Write([]byte("The time is: " + tm))
    }
}
```

The default servemux

You've probably seen the *default servemux* mentioned in a lot of places, from the simplest Hello World examples to the Go source code.

It took me a long time to realise it isn't anything special. The default servemux is just a plain ol' servemux like we've already been using, which gets instantiated by default when the `net/http` package is used and is stored in a global variable. Here's the relevant line from the Go source:

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

Generally speaking, I recommended against using the default servemux because it makes your code *less clear and explicit* and it poses a security risk. Because it's 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 servemux to expose a malicious handler to the web.

Instead it's better to use your own locally-scoped servemux, like we have been so far. But if you *do* decide to use the default servemux...

The `net/http` package provides a couple of shortcuts for registering routes with the default servemux: `http.Handle()` and `http.HandleFunc()`. These do exactly

the same as their namesake functions we've already looked at, with the difference that they add handlers to the default servemux instead of one that you've created.

Additionally, `http.ListenAndServe()` will fall back to using the default servemux if no other handler is provided (that is, the second argument is set to `nil`).

So as a final step, let's demonstrate how to use the default servemux in our application instead:

↳ File: main.go

```
package main

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

func timeHandler(format string) http.Handler {
    fn := func(w http.ResponseWriter, r *http.Request) {
        tm := time.Now().Format(format)
        w.Write([]byte("The time is: " + tm))
    }
    return http.HandlerFunc(fn)
}

func main() {
    // Note that we skip creating the ServeMux...

    var format string = time.RFC1123
    th := timeHandler(format)

    // We use http.Handle instead of mux.Handle...
    http.Handle("/time", th)

    log.Print("Listening...")
    // And pass nil as the handler to ListenAndServe.
```

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

If you enjoyed this article, you might like to check out my recommended tutorials list or check out my books Let's Go and Let's Go Further, which teach you everything you need to know about how to build professional production-ready web applications and APIs with Go.

Filed under: #golang #tutorial

Enjoy the rest of your Wednesday!