Making and Using HTTP Middleware

Last updated: May 10th, 2020

When you're building a web application there's probably some shared functionality that you want to run for many (or even all) HTTP requests. You might want to log every request, gzip every response, or check a cache before doing some expensive processing.

One way of organising this shared functionality is to set it up as *middleware* — self-contained code which independently acts on a request before or after your normal application handlers. In Go a common place to use middleware is between a router (such as `http.ServeMux`) and your application handlers, so that the flow of control for a HTTP request looks like:

```
Router → Middleware Handler → Application Handler
```

In this post I'm going to explain how to make custom middleware that works in this pattern, as well as running through some concrete examples of using third-party middleware packages.

The Basic Principles

Making and using middleware in Go is fundamentally simple. We want to:

- Implement our middleware so that it satisfies the `http.Handler` interface.
- Build up a *chain of handlers* containing both our middleware handler and our normal application handler, which we can register with a router.

I'll explain how.

Hopefully you're already familiar with the following pattern for constructing a handler:

```
func messageHandler(message string) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte(message)
    })
}
```

In this code we put our handler logic (a simple `w.Write()`) in an anonymous function which closes-over the `message` variable to form a *closure*. We then convert the closure to a handler with the `http.HandlerFunc()` adapter, and then return it.

Note: If this pattern is confusing or unfamiliar to you, I recommend reading this primer which explains it in-depth before continuing.

We can use this same pattern to help us create a chain of handlers. Instead of passing a string into the closure (like above) we could pass *the next handler in the chain* as a variable, and then transfer control to this next handler by calling it's `ServeHTTP()` method. This gives us a complete pattern for constructing middleware:

```
func exampleMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Our middleware logic goes here...
        next.ServeHTTP(w, r)
    })
}
```

You'll notice that this middleware function has a `func(http.Handler) http.Handler` signature. It accepts a handler as a parameter and returns a handler. This is useful for two reasons:

- Because it returns a handler we can register the middleware function directly with the standard `http.ServeMux` router in Go's `net/http` package.
- We can create an arbitrarily long handler chain by nesting middleware functions inside each other.

For example:

```
mux := http.NewServeMux()
mux.Handle("/", middlewareOne(middlewareTwo(finalHandler)))
```

Illustrating the Flow of Control

Let's look at a stripped-down example with some middleware that writes some log messages in your terminal window:

main.go

```
package main

import (
    "log"
    "net/http"
)

func middlewareOne(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Print("Executing middlewareOne")
        next.ServeHTTP(w, r)
        log.Print("Executing middlewareOne again")
    })
}

func middlewareTwo(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Print("Executing middlewareTwo")
        if r.URL.Path == "/foo" {
            return
        }

        next.ServeHTTP(w, r)
        log.Print("Executing middlewareTwo again")
    })
}
```

```go
func final(w http.ResponseWriter, r *http.Request) {
    log.Print("Executing finalHandler")
    w.Write([]byte("OK"))
}

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

    finalHandler := http.HandlerFunc(final)
    mux.Handle("/", middlewareOne(middlewareTwo(finalHandler)))

    log.Print("Listening on :3000...")
    err := http.ListenAndServe(":3000", mux)
    log.Fatal(err)
}
```

If you run this application and make a request to `http://localhost:3000`, you should see some log output similar to this:

```
$ go run main.go
2020/05/08 12:31:42 Listening on :3000...
2020/05/08 12:32:05 Executing middlewareOne
2020/05/08 12:32:05 Executing middlewareTwo
2020/05/08 12:32:05 Executing finalHandler
2020/05/08 12:32:05 Executing middlewareTwo again
2020/05/08 12:32:05 Executing middlewareOne again
```

It's clear to see how control is being passed through the handler chain in the order we nested them, and then back up again in the *reverse direction*.

We can stop control propagating through the chain at any point by issuing a `return` from a middleware handler.

In the example above I've included a conditional return in the `middlewareTwo` function. Try it by visiting `http://localhost:3000/foo` and checking the log again – you'll see that this time the request gets no further than `middlewareTwo` before passing back up the chain.

```
2020/05/08 12:33:22 Executing middlewareOne
2020/05/08 12:33:22 Executing middlewareTwo
2020/05/08 12:33:22 Executing middlewareOne again
```

Understood. How About a Proper Example?

OK, let's say that we're building a service which processes requests containing a JSON body.

We want to create some middleware which a) checks for the existence of a `Content-Type` header and b) if the header exists, check that it has the mime type `application/json`. If either of those checks fail, we want our middleware to write an error message and to stop the request from reaching our application handlers.

`main.go`

```
package main

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

func enforceJSONHandler(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        contentType := r.Header.Get("Content-Type")

        if contentType != "" {
            mt, _, err := mime.ParseMediaType(contentType)
            if err != nil {
                http.Error(w, "Malformed Content-Type header", http.StatusBadRequest)
                return
            }

            if mt != "application/json" {
                http.Error(w, "Content-Type header must be application/json", http.StatusUnsı
                return
            }
        }

        next.ServeHTTP(w, r)
    })
}

func final(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("OK"))
}

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

    finalHandler := http.HandlerFunc(final)
    mux.Handle("/", enforceJSONHandler(finalHandler))

    log.Print("Listening on :3000...")
    err := http.ListenAndServe(":3000", mux)
    log.Fatal(err)
}
```

Note: In the code above we're using the `mime.ParseMediaType()` function to extract the mime type from the header (which may include optional parameters, such as `charset` or `boundary`).

This looks good. Let's test it by making some requests using cURL:

```
$ curl -i localhost:3000
HTTP/1.1 400 Bad Request
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Fri, 08 May 2020 10:42:36 GMT
Content-Length: 37

Content-Type header must be provided

$ curl -i -H "Content-Type: application/xml" localhost:3000
HTTP/1.1 415 Unsupported Media Type
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Fri, 08 May 2020 10:42:39 GMT
Content-Length: 45

Content-Type header must be application/json

$ curl -i -H "Content-Type: application/json; charset=UTF-8" localhost:3000
HTTP/1.1 200 OK
Date: Fri, 08 May 2020 10:42:43 GMT
Content-Length: 2
Content-Type: text/plain; charset=utf-8

OK
```

Using Third-Party Middleware

Rather than rolling your own middleware all the time you might decide to save time and effort and use an existing third-party package. Let's take a look at a couple of examples to help demonstrate some common patterns that you might come across.

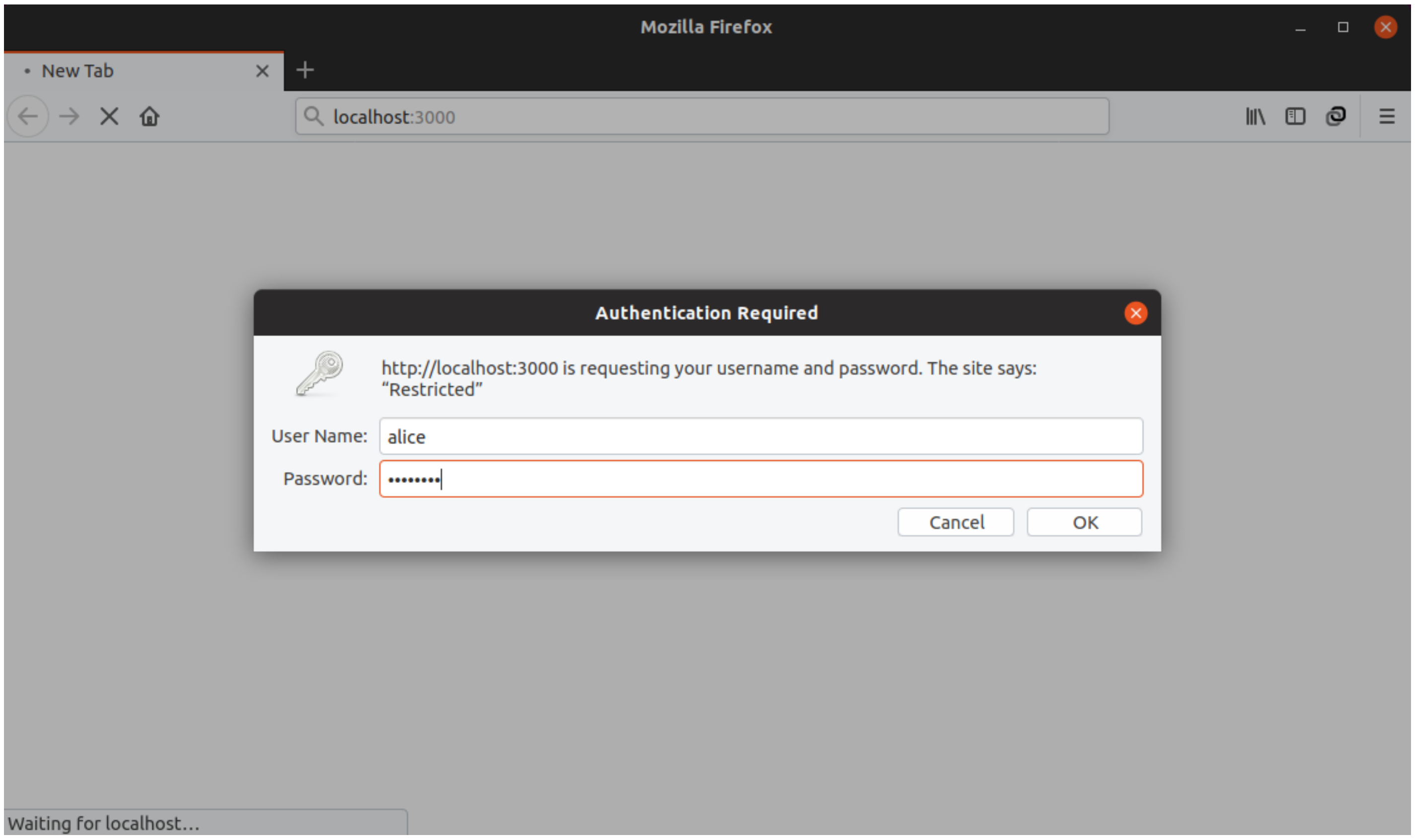
The first third-party middleware we'll demonstrate is `goji/httpauth`, which provides HTTP Basic Authentication functionality.

When using this package you call a *helper function* in order to setup the chainable middleware. Specifically, you call the `httpauth.SimpleBasicAuth()` function, and this returns a middleware function with the signature `func(http.Handler) http.Handler` — which you can then use in exactly the same way as any custom-built middleware.

`main.go`

```go
package main

import (
    "log"
    "net/http"

    "github.com/goji/httpauth"
)

func main() {
    authHandler := httpauth.SimpleBasicAuth("alice", "pa$$word")

    mux := http.NewServeMux()

    finalHandler := http.HandlerFunc(final)
    mux.Handle("/", authHandler(finalHandler))

    log.Print("Listening on :3000...")
    err := http.ListenAndServe(":3000", mux)
    log.Fatal(err)
}

func final(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("OK"))
}
```

If you run this code then visit `http://localhost:3000` in your browser you should get a username and password prompt like so:



http://localhost:3000 is requesting your username and password. The site says: "Restricted"

User Name: alice

Password: ••••••••

Entering the wrong username and password should result in the prompt being redisplayed, clicking 'Cancel' should result in a plain-text `"Unauthorized"` response, and using the correct username (`alice`) and password (`pa$$word`) should result in an `"OK"` response.

That was pretty straightforward and easy to integrate. Let's now look at a different example using the `LoggingHandler` middleware from the `gorilla/handlers` package, which records request logs using the Apache Common Log Format.

Instead of using the standard middleware signature that we've seen so far throughout this post, this middleware has the signature `func(out io.Writer, h http.Handler) http.Handler`, so it takes not only the next handler but also the `io.Writer` that the log will be written to.

Here's a simple example of using it in which we write logs to a `server.log` file in the current directory:

main.go

```go
package main

import (
    "log"
    "net/http"
    "os"

    "github.com/gorilla/handlers"
)

func main() {
    logFile, err := os.OpenFile("server.log", os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0664)
    if err != nil {
        log.Fatal(err)
    }

    mux := http.NewServeMux()

    finalHandler := http.HandlerFunc(final)
    mux.Handle("/", handlers.LoggingHandler(logFile, finalHandler))

    log.Print("Listening on :3000...")
    err = http.ListenAndServe(":3000", mux)
    log.Fatal(err)
}

func final(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("OK"))
}
```

In a trivial case like this our code is fairly clear. But what happens if we want to use this as part of a larger middleware chain? We could easily end up with a declaration looking something like this...

```
http.Handle("/", handlers.LoggingHandler(logFile, authHandler(enforceJSONHandler(finalHandler
```

... And that's pretty confusing!

To help tidy this up it's possible to create a *constructor function* which wraps the `LoggingHandler()` middleware and returns a standard `func(http.Handler) http.Handler` function that we can nest neatly with other middleware. Like so:

```
package main

import (
    "io"
    "log"
    "net/http"
    "os"

    "github.com/gorilla/handlers"
)

func newLoggingHandler(dst io.Writer) func(http.Handler) http.Handler {
    return func(h http.Handler) http.Handler {
        return handlers.LoggingHandler(dst, h)
    }
}
```

```
func main() {
    logFile, err := os.OpenFile("server.log", os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0664)
    if err != nil {
        log.Fatal(err)
    }

    loggingHandler := newLoggingHandler(logFile)

    mux := http.NewServeMux()

    finalHandler := http.HandlerFunc(final)
    mux.Handle("/", loggingHandler(finalHandler))

    log.Print("Listening on :3000...")
    err = http.ListenAndServe(":3000", mux)
    log.Fatal(err)
}

func final(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("OK"))
}
```

If you run this application and make a few requests, your `server.log` file should look something like this:

```
$ cat server.log
127.0.0.1 - - [10/May/2020:15:11:30 +0200] "GET / HTTP/1.1" 200 2
127.0.0.1 - - [10/May/2020:15:11:31 +0200] "POST / HTTP/1.1" 200 2
127.0.0.1 - - [10/May/2020:15:11:33 +0200] "PUT / HTTP/1.1" 200 2
```

If you're interested, here's a gist of the three middleware handlers from this post combined in one example.

Additional Tools

The justinas/alice package is very lightweight tool which provides some syntactic sugar for chaining middleware handlers. At it's most basic, it lets you rewrite this:

```
mux.Handle("/", loggingHandler(authHandler(enforceJSONHandler(finalHandler))))
```

As this:

```
mux.Handle("/", alice.New(loggingHandler, authHandler, enforceJSONHandler).Then(finalHandler
```

To my eyes at least, that code is slightly clearer to understand at a glance. However, the real benefit of Alice is that it lets you to specify a handler chain once and reuse it for multiple routes. Like so:

```
stdChain := alice.New(loggingHandler, authHandler, enforceJSONHandler)
```

```
mux.Handle("/foo", stdChain.Then(fooHandler))
mux.Handle("/bar", stdChain.Then(barHandler))
```

ⓘ *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