3 Handling requests

This chapter covers

- Using the Go net/http library
- Serving out HTTP using the Go net/http library
- Understanding handlers and handler functions
- Working with multiplexers

Chapter 2 showed the steps for creating a simple internet forum web application. The chapter mapped out the various parts of a Go web application, and you saw the big picture of how a Go web application is structured. But there's little depth in each of those parts. In the next few chapters, we'll delve into the details of each of these parts and explore in depth how they can be put together.

In this and the next chapter, we'll focus on the brains of the web application: the handlers that receive and process requests from the client. In this chapter, you'll learn how to create a web server with Go, and then we'll move on to handling requests from the client.

3.1 The Go net/http library

Although using a mature and sophisticated web application framework to write web applications is usually easy and fast, the same frameworks often impose their own conventions and patterns. Many assume that these conventions and patterns are best practices, but best practices have a way of growing into *cargo cult programming* when they aren't understood properly. Programmers following these conventions without understanding why they're used often follow them blindly and reuse them when it's unnecessary or even harmful.

Cargo cult programming

During World War II, the Allied forces set up air bases on islands in the Pacific to help with the war efforts. Large amounts of supplies and military equipment were air-dropped to troops and islanders supporting the troops, drastically changing their lives. For the first time, the islanders saw manufactured clothes, canned food, and other goods. When the war ended, the bases were abandoned and the cargo stopped arriving. So the islanders did a very natural thing—they dressed themselves up as air traffic controllers, soldiers, and sailors, waved landing signals using sticks on the airfields, and performed parade ground drills in an attempt to get cargo to continue falling by parachute from planes.

These cargo cultists gave their names to the practice of cargo cult programming. While not exactly waving landing signals, cargo cult programmers copy and paste code they either inherit or find on the internet (often, StackOverflow) without understanding why it works, only that it works. As a result, they're often unable to extend or make changes to code. Similarly, cargo cult programmers often use web frameworks without understanding why the framework uses certain patterns or conventions, as well as the trade-offs that are being made.

The reason data is persisted as cookies in the client and sessions in the server is because HTTP is a connection-less protocol, and each call to the server has no stored knowledge of the previous call. Without this understanding, using cookies and sessions seems a convoluted way of persisting information between connections. Using a framework to get around this complexity is smart because a framework normally hides the complexity and presents a uniform interface for persistence between connections. As a result, a new programmer would simply assume all it takes to persist data between connections is to use this interface. This uniform interface is based on the conventions of a specific framework, though, and such practices might or might not be consistent across all frameworks. What's worse, the same interface name might be used in different frameworks, with different implementations and different names, adding to the confusion. This means that the web application that's developed is now tied to the

framework; moving it to another framework or even extending the application or adding new features requires deep knowledge of the framework (or customized versions of the framework).

This book isn't about rejecting frameworks or conventions or patterns. A good web application framework is often the best way to build scalable and robust web applications quickly. But it's important to understand the underlying concepts infrastructure that these frameworks are built on. In the case of the Go programming language, using the standard libraries typically means using the net/http and html/template libraries. With proper understanding, it becomes easier to see why certain conventions and patterns are what they are. This helps us to avoid pitfalls, gives clarity, and stops us from following patterns blindly.

In this and the next chapter, we'll be focusing on net/http; chapter 5 covers html/template.

The net/http library is divided into two parts, with various structs and functions supporting either one or both (see figure 3.1):

- *Client*—Client, Response, Header, Request, Cookie
- *Server*—Server, ServeMux, Handler/HandleFunc, ResponseWriter, Header, Request, Cookie

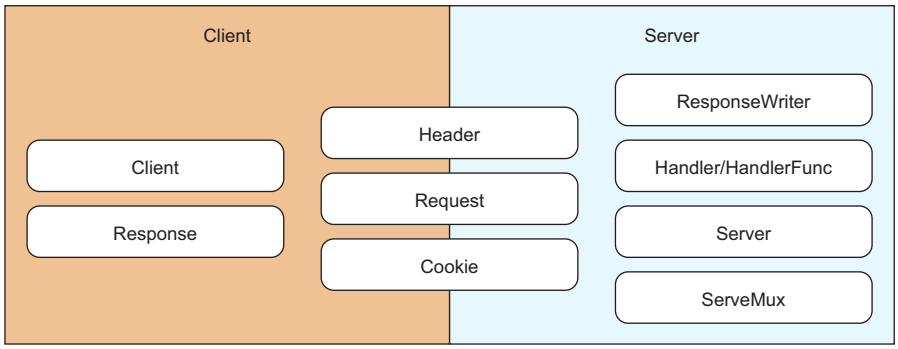

Figure 3.1 Chaining handlers

We'll start by using the net/http library as the server, and in this chapter we'll talk about how Go handles requests from the client. In the next chapter, we'll continue with the net/http library but focus on using it to process the request.

In this book, we'll focus on using the net/http library's server capabilities and not its client capabilities.

3.2 Serving Go

The net/http library provides capabilities for starting up an HTTP server that handles requests and sends responses to those requests (see figure 3.2). It also provides an interface for a multiplexer and a default multiplexer.

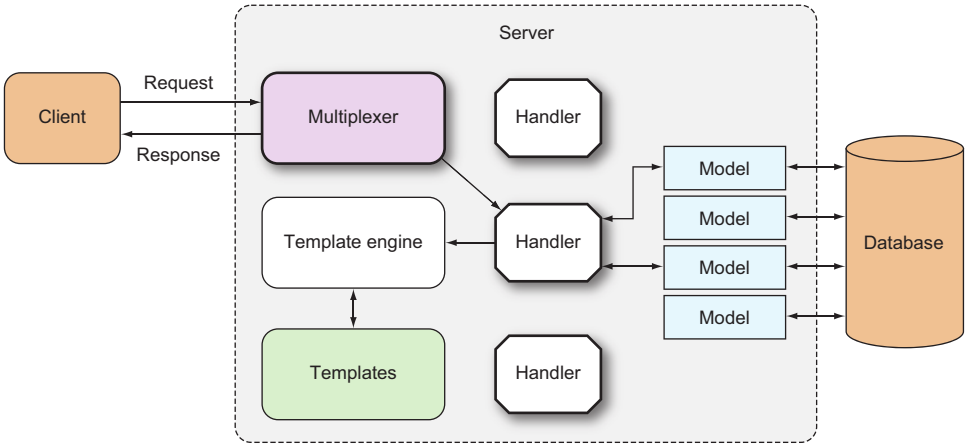

Figure 3.2 Handling requests with the Go server

3.2.1 The Go web server

Unlike most standard libraries in other programming languages, Go provides a set of libraries to create a web server. Creating a server is trivial and can be done with a call to `ListenAndServe`, with the network address as the first parameter and the handler that takes care of the requests the second parameter, as shown in the following listing. If the network address is an empty string, the default is all network interfaces at port 80. If the handler parameter is `nil`, the default multiplexer, `DefaultServeMux`, is used.

Listing 3.1 The simplest web server

```
package main

import (
     "net/http"
)

func main() {
     http.ListenAndServe("", nil)
}
```

This simple server doesn't allow much configuration, but Go also provides a `Server` struct that's essentially a server configuration.

Listing 3.2 Web server with additional configuration

```
package main

import (
    "net/http"
)

func main() {
    server := http.Server{
        Addr:    "127.0.0.1:8080",
        Handler: nil,
    }
    server.ListenAndServe()
}
```

The following listing does almost the same thing as the previous code but now allows more configurations. Configurations include setting the timeout for reading the request and writing the response and setting an error logger for the `Server` struct.

Listing 3.3 The `Server` struct configuration

```
type Server struct {
    Addr           string
    Handler        Handler
    ReadTimeout    time.Duration
    WriteTimeout   time.Duration
    MaxHeaderBytes int
    TLSConfig      *tls.Config
    TLSNextProto   map[string]func(*Server, *tls.Conn, Handler)
    ConnState      func(net.Conn, ConnState)
    ErrorLog       *log.Logger
}
```

3.2.2 Serving through HTTPS

Most major websites use HTTPS to encrypt and protect the communications between the client and the server when confidential information like passwords and credit card information is shared. In some cases, this protection is mandated. If you accept credit card payments, you need to be compliant with the Payment Card Industry (PCI) Data Security Standard, and to be compliant you need to encrypt the communications between the client and the server. Some sites like Gmail and Facebook use HTTPS throughout their entire site. If you're planning to run a site that requires the user to log in, you'll need to use HTTPS.

HTTPS is nothing more than layering HTTP on top of SSL (actually, Transport Security Layer [TLS]). To serve our simple web application through HTTPS, we'll use the `ListenAndServeTLS` function, shown in listing 3.4.

SSL, TLS, and HTTPS

SSL (Secure Socket Layer) is a protocol that provides data encryption and authentication between two parties, usually a client and a server, using Public Key Infrastructure (PKI). SSL was originally developed by Netscape and was later taken over by the Internet Engineering Task Force (IETF), which renamed it TLS. HTTPS, or HTTP over SSL, is essentially just that—HTTP layered over an SSL/TLS connection.

An SSL/TLS certificate (I'll use the term SSL certificate as it's more widely known) is used to provide data encryption and authentication. An SSL certificate is an X.509-formatted piece of data that contains some information, as well as a public key, stored at a web server. SSL certificates are usually signed by a certificate authority (CA), which assures the authenticity of the certificate. When the client makes a request to the server, it returns with the certificate. If the client is satisfied that the certificate is authentic, it will generate a random key and use the certificate (or more specifically the public key in the certificate) to encrypt it. This symmetric key is the actual key used to encrypt the data between the client and the server.

Listing 3.4 Serving through HTTPS

```
package main

import (
    "net/http"
)

func main() {
    server := http.Server{
        Addr:    "127.0.0.1:8080",
        Handler: nil,
    }
    server.ListenAndServeTLS("cert.pem", "key.pem")
}
```

In the previous listing, the cert.pem file is the *SSL certificate* whereas key.pem is the *private key* for the server. In a production scenario you'll need to get the SSL certificate from a CA like VeriSign, Thawte, or Comodo SSL. But if you need a certificate and private key only to try things out, you can generate your own certificates. There are many ways of generating them, including using Go standard libraries, mostly under the crypto library group.

Although you won't use them (the certificate and private key created here) in a production server, it's useful to understand how an SSL certificate and private key

can be generated for development and testing purposes. This listing shows how we can do this.

Listing 3.5 Generating your own SSL certificate and server private key

```
package main

import (
    "crypto/rand"
    "crypto/rsa"
    "crypto/x509"
    "crypto/x509/pkix"
    "encoding/pem"
    "math/big"
    "net"
    "os"
    "time"
)

func main() {
    max := new(big.Int).Lsh(big.NewInt(1), 128)
    serialNumber, _ := rand.Int(rand.Reader, max)
    subject := pkix.Name{
        Organization:       []string{"Manning Publications Co."},
        OrganizationalUnit: []string{"Books"},
        CommonName:         "Go Web Programming",
    }

    template := x509.Certificate{
        SerialNumber: serialNumber,
        Subject:      subject,
        NotBefore:    time.Now(),
        NotAfter:     time.Now().Add(365 * 24 * time.Hour),
        KeyUsage:     x509.KeyUsageKeyEncipherment | x509.KeyUsageDigitalSig-
  nature,
        ExtKeyUsage:  []x509.ExtKeyUsage{x509.ExtKeyUsageServerAuth},
        IPAddresses:  []net.IP{net.ParseIP("127.0.0.1")},
    }

    pk, _ := rsa.GenerateKey(rand.Reader, 2048)

    derBytes, _ := x509.CreateCertificate(rand.Reader, &template,
    ➥ &template, &pk.PublicKey, pk)
    certOut, _ := os.Create("cert.pem")
    pem.Encode(certOut, &pem.Block{Type: "CERTIFICATE", Bytes: derBytes})
    certOut.Close()

    keyOut, _ := os.Create("key.pem")
    pem.Encode(keyOut, &pem.Block{Type: "RSA PRIVATE KEY", Bytes:
    ➥ x509.MarshalPKCS1PrivateKey(pk)})
    keyOut.Close()
}
```

Generating the SSL certificate and private key is relatively easy. An SSL certificate is essentially an X.509 certificate with the extended key usage set to server authentication, so we'll be using the crypto/x509 library to create the certificate. The private key is required to create the certificate, so we simply take the private key we created for the certificate and save it into a file for the server private key file.

Let's go through the code. First, we need to have a `Certificate` struct, which allows us to set the configuration for our certificate:

```
template := x509.Certificate{
  SerialNumber: serialNumber,
  Subject: subject,
  NotBefore: time.Now(),
  NotAfter:  time.Now().Add(365*24*time.Hour),
  KeyUsage: x509.KeyUsageKeyEncipherment | x509.KeyUsageDigitalSignature,
  ExtKeyUsage: []x509.ExtKeyUsage{x509.ExtKeyUsageServerAuth},
  IPAddresses: []net.IP{net.ParseIP("127.0.0.1")},
}
```

We need a certificate serial number, which is a unique number issued by the CA. For our purposes, it's good enough to use a very large integer that's randomly generated. Next, we create the distinguished name and set it up as the subject for the certificate, and we also set up the validity period to last for one year from the day the certificate is created. The KeyUsage and ExtKeyUsage fields are used to indicate that this X.509 certificate is used for server authentication. Finally, we set up the certificate to run from the IP 127.0.0.1 only.

SSL certificates

X.509 is an ITU-T (International Telecommunication Union Telecommunication Standardization Sector) standard for a Public Key Infrastructure (PKI). X.509 includes standard formats for public key certificates.

An X.509 certificate (also colloquially called an SSL certificate) is a digital document expressed in ASN.1 (Abstract Syntax Notation One) that has been encoded. ASN.1 is a standard and notation that describes rules and structures for representing data in telecommunications and computer networking.

X.509 certificates can be encoded in various formats, including BER (Basic Encoding Rules). The BER format specifies a self-describing and self-delimiting format for encoding ASN.1 data structures. DER is a subset of BER, providing for exactly one way to encode an ASN.1 value, and is widely used in cryptography, especially X.509 certificates.

In SSL, the certificates can be saved in files of different formats. One of them is PEM (Privacy Enhanced Email, which doesn't have much relevance here except as the name of the file format used), which is a Base64-encoded DER X.509 certificate enclosed between "—BEGIN CERTIFICATE—" and "—END CERTIFICATE—".

Next, we need to generate a private key. We use the crypto/rsa library and call the `GenerateKey` function to create an RSA private key:

```
pk, _ := rsa.GenerateKey(rand.Reader, 2048)
```

The RSA private key struct that's created has a public key that we can access, useful when we use the `x509.CreateCertificate` function to create our SSL certificate:

```
derBytes, _ := x509.CreateCertificate(rand.Reader, &template, &template,
➥ &pk.PublicKey, pk)
```

The `CreateCertificate` function takes a number of parameters, including the `Certificate` struct and the public and private keys, to create a slice of DER-formatted bytes. The rest is relatively straightforward: we use the encoding/pem library to encode the certificate into the cert.pem file:

```
certOut, _ := os.Create("cert.pem")
pem.Encode(certOut, &pem.Block{Type: "CERTIFICATE", Bytes: derBytes})
certOut.Close()
```

We also PEM encode and save the key we generated earlier into the key.pem file:

```
keyOut, _ := os.Create("key.pem")
pem.Encode(keyOut, &pem.Block{Type: "RSA PRIVATE KEY", Bytes:
➥ x509.MarshalPKCS1PrivateKey(pk)})
keyOut.Close()
```

Note that if the certificate is signed by a CA, the certificate file should be the concatenation of the server's certificate followed by the CA's certificate.

3.3 Handlers and handler functions

Starting up a server is easy, but it doesn't do anything. If you access the server, you'll get only a 404 HTTP response code. The default multiplexer that will be used if the handler parameter is nil can't find any handlers (because we haven't written any) and will respond with the 404. To do any work, we need to have handlers.

3.3.1 Handling requests

So what exactly is a handler? We talked briefly about handlers and handler functions in chapters 1 and 2, so let's elaborate here. In Go, a handler is an interface that has a method named `ServeHTTP` with two parameters: an `HTTPResponseWriter` interface and a pointer to a `Request` struct. In other words, anything that has a method called `ServeHTTP` with this method signature is a handler:

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

Let me digress and answer a question that might have occurred to you as you're reading this chapter. If the second parameter for `ListenAndServe` is a handler, then why is the default value a multiplexer, `DefaultServeMux`?

That's because `ServeMux` (which is what `DefaultServeMux` is an instance of) has a method named `ServeHTTP` with the same signature! In other words, a `ServeMux` is also an instance of the `Handler` struct. `DefaultServeMux` is an instance of `ServeMux`, so it is also an instance of the `Handler` struct. It's a special type of handler, though, because the only thing it does is redirect your requests to different handlers depending on the URL that's provided. If we use a handler instead of the default multiplexer, we'll be able to respond, as shown in this listing.

Listing 3.6 Handling requests

```
package main

import (
    "fmt"
    "net/http"
)

type MyHandler struct{}

func (h *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
}

func main() {
    handler := MyHandler{}
    server := http.Server{
        Addr:    "127.0.0.1:8080",
        Handler: &handler,
    }
    server.ListenAndServe()
}
```

Now let's start the server (if you're a bit hazy on how to do this, please flip to section 2.7). If you go to http://localhost:8080 in your browser you'll see Hello World!

Here's the tricky bit: if you go to http://localhost:8080/anything/at/all you'll still get the same response! Why this is so should be quite obvious. We just created a handler and attached it to our server, so we're no longer using any multiplexers. This means there's no longer any URL matching to route the request to a particular handler, so all requests going into the server will go to this handler.

In our handler, the `ServeHTTP` method does all the processing. It doesn't do anything except return Hello World!, so that's what it does for all requests into the server.

This is the reason why we'd normally use a multiplexer. Most of the time we want the server to respond to more than one request, depending on the request URL. Naturally if you're writing a very specialized server for a very specialized purpose, simply creating one handler will do the job marvelously.

3.3.2 More handlers

Most of the time, we don't want to have a single handler to handle all the requests like in listing 3.6; instead we want to use different handlers instead for different URLs. To

do this, we don't specify the Handler field in the `Server` struct (which means it will use the `DefaultServeMux` as the handler); we use the `http.Handle` function to attach a handler to `DefaultServeMux`. Notice that some of the functions like `Handle` are functions for the `http` package and also methods for `ServeMux`. These functions are actually convenience functions; calling them simply calls `DefaultServeMux`'s corresponding functions. If you call `http.Handle` you're actually calling `DefaultServeMux`'s `Handle` method.

In the following listing, we create two handlers and then attach the handler to the respective URL. If you now go to http://localhost:8080/hello you'll get Hello! whereas if you go to http://localhost:8080/world, you'll get World!.

Listing 3.7 Handling requests with multiple handlers

```
package main

import (
    "fmt"
    "net/http"
)

type HelloHandler struct{}

func (h *HelloHandler) ServeHTTP (w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello!")
}

type WorldHandler struct{}

func (h *WorldHandler) ServeHTTP (w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "World!")
}

func main() {
    hello := HelloHandler{}
    world := WorldHandler{}

    server := http.Server{
        Addr: "127.0.0.1:8080",
    }

    http.Handle("/hello", &hello)
    http.Handle("/world", &world)

    server.ListenAndServe()
}
```

3.3.3 Handler functions

We talked about handlers, but what are handler functions? Handler functions are functions that behave like handlers. Handler functions have the same signature as the `ServeHTTP` method; that is, they accept a `ResponseWriter` and a pointer to a `Request`. The following listing shows how this works with our server.

Listing 3.8 Handling requests with handler functions

```
package main

import (
    "fmt"
    "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello!")
}

func world(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "World!")
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/hello", hello)
    http.HandleFunc("/world", world)

    server.ListenAndServe()
}
```

How does this work? Go has a function type named `HandlerFunc`, which will adapt a function `f` with the appropriate signature into a `Handler` with a method `f`. For example, take the `hello` function:

```
func hello(w http.ResponseWriter, r *http.Request) {
      fmt.Fprintf(w, "Hello!")
}
```

If we do this:

```
helloHandler := HandlerFunc(hello)
```

then `helloHandler` becomes a `Handler`. Confused? Let's go back to our earlier server, which accepts handlers.

```
type MyHandler struct{}

func (h *MyHandler) ServeHTTP (w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
}

func main() {
    handler := MyHandler{}
    server := http.Server{
        Addr:    "127.0.0.1:8080",
        Handler: &handler,
    }
    server.ListenAndServe()
}
```

The line that registers the `hello` function to the URL */hello* is

```
http.Handle("/hello", &hello)
```

This shows us how the `Handle` function registers a pointer to a `Handler` to a URL. To simplify things, the `HandleFunc` function converts the `hello` function into a `Handler` and registers it to `DefaultServeMux`. In other words, handler functions are merely convenient ways of creating handlers. The following listing shows the code for the `http.HandleFunc` function.

Listing 3.9 `http.HandleFunc` source code

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
     DefaultServeMux.HandleFunc(pattern, handler)
}
```

Here's the source code for the `HandleFunc` function:

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter,
     *Request)) {
    mux.Handle(pattern, HandlerFunc(handler))
}
```

Notice that `handler`, a function, is converted into an actual handler by `HandlerFunc`.

Because using handler functions is cleaner and it does the job just as well, why use handlers at all? It all boils down to design. If you have an existing interface or if you want a type that can also be used as a handler, simply add a `ServeHTTP` method to that interface and you'll get a handler that you can assign to a URL. It can also allow you to build web applications that are more modular.

3.3.4 Chaining handlers and handler functions

Although Go isn't considered a functional language, it has some features that are common to functional languages, including function types, anonymous functions, and closures. As you noticed earlier, we passed a function into another function and we referred to a named function by its identifier. This means we can pass a function f1 into another function f2 for f2 to do its processing, and then call f1 (see figure 3.3).

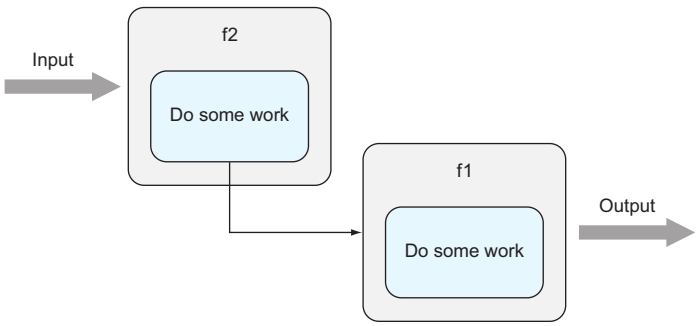

Figure 3.3 Chaining handlers

Let's work through an example. Say every time we call a handler we want to log it down somewhere that it was called. We can always add this code into the handler, or we can refactor a utility function (as we did in chapter 2) that can be called by every function. Doing this can be intrusive, though; we usually want our handler to contain logic for processing the request only.

Logging, along with a number of similar functions like security and error handling, is what's commonly known as a *cross-cutting concern*. These functions are common and we want to avoid adding them everywhere, which causes code duplication and dependencies. A common way of cleanly separating cross-cutting concerns away from your other logic is *chaining*. This listing shows how we can chain handlers.

Listing 3.10 Chaining two handler functions

```
package main

import (
    "fmt"
    "net/http"
    "reflect"
    "runtime"
)

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello!")
}

func log(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        name := runtime.FuncForPC(reflect.ValueOf(h).Pointer()).Name()
        fmt.Println("Handler function called - " + name)
        h(w, r)
    }
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/hello", log(hello))
    server.ListenAndServe()
}
```

We have our usual `hello` handler function. We also have a `log` function, which takes in a `HandlerFunc` and returns a `HandlerFunc`. Remember that `hello` is a `HandlerFunc`, so this sends the `hello` function into the `log` function; in other words it chains the `log` and the `hello` functions.

```
log(hello)
```

The `log` function returns an anonymous function that takes a `ResponseWriter` and a pointer to a `Request`, which means that the anonymous function is a `HandlerFunc`. Inside the anonymous function, we print out the name of the `HandlerFunc` (in this

case it's `main.hello`), and then call it. As a result, we'll get hello! in the browser and a printed statement on the console that says this:

```
Handler function called - main.hello
```

Naturally if we can chain together two handler functions, we can chain more. The same principle allows us to stack handlers to perform multiple actions, like Lego bricks. This is sometimes called *pipeline processing* (see figure 3.4).

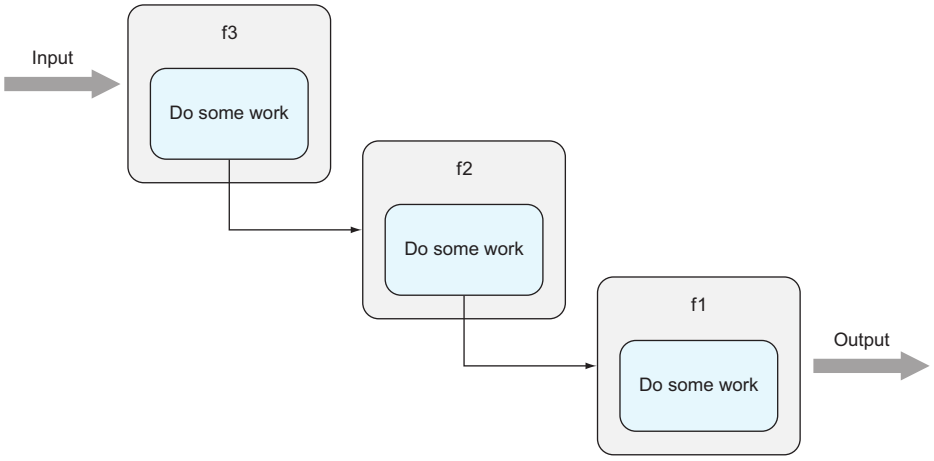

Figure 3.4 Chaining more handlers

Say we have another function named `protect`, which checks for the user's authorization before executing the handler:

```
func protect(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        . . .
        h(w, r)
    }
}
```

Code, omitted for brevity, to make sure the user is authorized.

Then to use `protect`, we simply chain them together:

```
http.HandleFunc("/hello", protect(log(hello)))
```

You might have noticed that while I mentioned earlier that we're chaining handlers, the code in listing 3.10 actually shows chaining handler functions. The mechanisms for both chaining handlers and handler functions are very similar, as shown next.

Listing 3.11 Chaining handlers

```
package main

import (
    "fmt"
```

```
    "net/http"
)

type HelloHandler struct{}

func (h HelloHandler) ServeHTTP (w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello!")
}

func log(h http.Handler) http.Handler {
    return http.HandlerFunc (func(w http.ResponseWriter, r *http.Request) {
        fmt.Printf("Handler called - %T\n", h)
        h.ServeHTTP (w, r)
    })
}

func protect(h http.Handler) http.Handler {
    return http.HandlerFunc (func(w http.ResponseWriter, r *http.Request) {
        . . .
        h.ServeHTTP (w, r)
    })
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    hello := HelloHandler{}
    http.Handle("/hello", protect(log(hello)))
    server.ListenAndServe()
}
```

Code, omitted for brevity, to make sure the user is authorized.

Let's see what's different. We have our `HelloHandler` from earlier, which is the last handler in the chain, as before. The `log` function, instead of taking in a `HandlerFunc` and returning a `HandlerFunc`, takes in a `Handler` and returns a `Handler`:

```
func log(h http.Handler) http.Handler {
    return http.HandlerFunc (func(w http.ResponseWriter, r *http.Request) {
        fmt.Printf("Handler called - %T\n", h)
        h.ServeHTTP (w, r)
    })
}
```

Instead of returning an anonymous function, now we adapt that anonymous function using `HandlerFunc`, which, if you remember from earlier, returns a `Handler`. Also, instead of executing the handler function, we now take the handler and call its `ServeHTTP` function. Everything else remains mostly the same, except that instead of registering a handler function, we register the handler:

```
hello := HelloHandler{}
http.Handle("/hello", protect(log(hello)))
```

Chaining handlers or handler functions is a common idiom you'll find in many web application frameworks.

3.3.5 ServeMux and DefaultServeMux

We discussed `ServeMux` and `DefaultServeMux` earlier in this chapter and in the previous chapter. `ServeMux` is an HTTP request multiplexer. It accepts an HTTP request and redirects it to the correct handler according to the URL in the request, illustrated in figure 3.5.

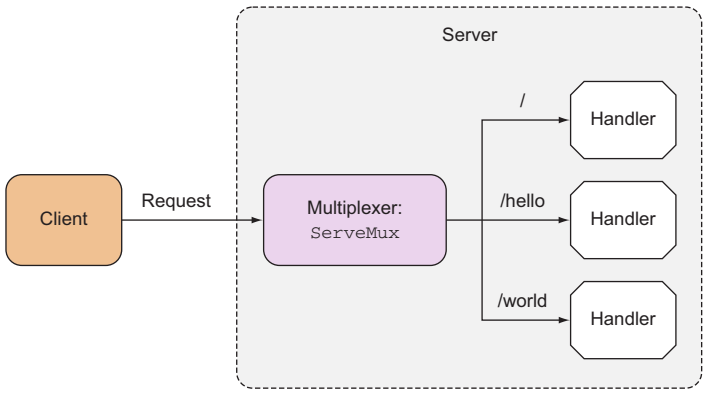

Figure 3.5 Multiplexing requests to handlers

`ServeMux` is a struct with a map of entries that map a URL to a handler. It's also a handler because it has a `ServeHTTP` method. `ServeMux`'s `ServeHTTP` method finds the URL most closely matching the requested one and calls the corresponding handler's `ServeHTTP` (see figure 3.6).

So what is `DefaultServeMux`? `ServeMux` isn't an interface, so `DefaultServeMux` isn't an implementation of `ServeMux`. `DefaultServeMux` is an instance of `ServeMux` that's

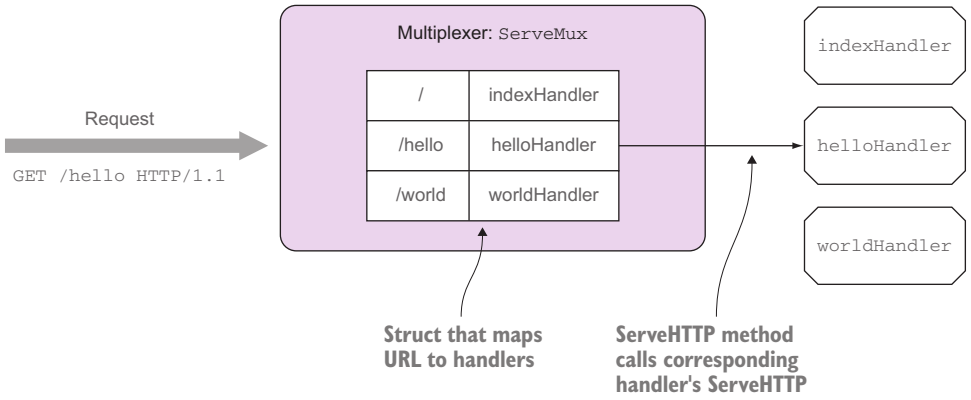

Figure 3.6 Inside a multiplexer

publicly available to the application that imports the net/http library. It's also the instance of `ServeMux` that's used when no handler is provided to the `Server` struct.

Stepping back a bit, you'll also come to a realization that `ServeMux` is also another take on chaining handlers, because `ServeMux` is a handler.

In these examples the requested URL /hello matches nicely with the registered URL in the multiplexer. What happens if we call the URL /random? Or if we call the URL /hello/there?

It all depends on how we register the URLs. If we register the root URL (/) as in figure 3.6, any URLs that don't match will fall through the hierarchy and land on the root URL. If we now call /random and don't have the handler for this URL, the root URL's handler (in this case indexHandler) will be called.

How about /hello/there then? The *Principle of Least Surprise* would dictate that because we have the URL /hello registered we should default to that URL and `helloHandler` should be called. But in figure 3.6, `indexHandler` is called instead. Why is that so?

> **The Principle of Least Surprise**
>
> The Principle of Least Surprise, also known as the Principle of Least Astonishment, is a general principle in the design of all things (including software) that says that when designing, we should do the least surprising thing. The results of doing something should be obvious, consistent, and predictable.
>
> If we place a button next to a door, we'd expect the button to do something with the door (ring the doorbell or open the door). If the button turns off the corridor lights instead, that would be against the Principle of Least Surprise because it's doing something that a user of that button wouldn't be expecting.

The reason is because we registered the `helloHandler` to the URL /hello instead of /hello/. For any registered URLs that don't end with a slash (/), `ServeMux` will try to match the exact URL pattern. If the URL ends with a slash (/), `ServeMux` will see if the requested URL starts with any registered URL.

If we'd registered the URL /hello/ instead, then when /hello/there is requested, if `ServeMux` can't find an exact match, it'll start looking for URLs that start with /hello/. There's a match, so `helloHandler` will be called.

3.3.6 Other multiplexers

Because what it takes to be a handler or even a multiplexer is to implement the `ServeHTTP`, it's possible to create alternative multiplexers to net/http's `ServeMux`. Sure enough, a number of third-party multiplexers are available, including the excellent Gorilla Toolkit (www.gorillatoolkit.org). The Gorilla Toolkit has two different multiplexers that work quite differently: mux and pat. In this section, we'll go through a lightweight but effective third-party multiplexer called HttpRouter.

One of the main complaints about `ServeMux` is that it doesn't support variables in its pattern matching against the URL. `ServeMux` handles /threads pretty well to retrieve and display all threads in the forum, but it's difficult to handle /thread/123 for retrieving and displaying the thread with id 123. To process such URLs, your handler will need to parse the request path and extract the relevant sections. Also, because of the way `ServeMux` does pattern matching for the URLs, you can't use something like /thread/123/post/456 if you want to retrieve the post with id 456 from the thread with id 123 (at least not with a lot of unnecessary parsing complexity).

The HttpRouter library overcomes some of these limitations. In this section, we'll explore some of the more important features of this library, but you can always look up the rest of the documentation at https://github.com/julienschmidt/httprouter. This listing shows an implementation using HttpRouter.

Listing 3.12 Using HttpRouter

```
package main

import (
    "fmt"
    "github.com/julienschmidt/httprouter"
    "net/http"
)

func hello(w http.ResponseWriter, r *http.Request, p httprouter.Params) {
    fmt.Fprintf(w, "hello, %s!\n", p.ByName("name"))
}

func main() {
    mux := httprouter.New()
    mux.GET("/hello/:name", hello)

    server := http.Server{
        Addr:    "127.0.0.1:8080",
        Handler: mux,
    }
    server.ListenAndServe()
}
```

Most of the code should look familiar to you now. We create the multiplexer by calling the `New` function.

```
mux := httprouter.New()
```

Instead of using `HandleFunc` to register the handler functions, we use the method names:

```
mux.GET("/hello/:name", hello)
```

In this case we're registering a URL for the GET method to the `hello` function. If we send a GET request, the `hello` function will be called; if we send any other HTTP

requests it won't. Notice that the URL now has something called a *named parameter*. These named parameters can be replaced by any values and can be retrieved later by the handler.

```
func hello(w http.ResponseWriter, r *http.Request, p httprouter.Params) {
     fmt.Fprintf(w, "hello, %s!\n", p.ByName("name"))
}
```

The handler function has changed too; instead of taking two parameters, we now take a third, a `Params` type. `Params` contain the named parameters, which we can retrieve using the `ByName` method.

Finally, instead of using `DefaultServeMux`, we pass our multiplexer into the `Server` struct and let Go use that instead:

```
  server := http.Server{
    Addr:    "127.0.0.1:8080",
    Handler: mux,
}
server.ListenAndServe()
```

But wait. How exactly do we include the third-party library? If we do what we did for the other examples and run `go build` at the console, we'll get something like this:

```
$ go build
server.go:5:5: cannot find package "github.com/julienschmidt/httprouter" in
   any of:
    /usr/local/go/src/github.com/julienschmidt/httprouter (from $GOROOT)
    /Users/sausheong/gws/src/github.com/julienschmidt/httprouter (from $GOPATH)
```

A simple package management system is one of the strengths of Go. We simply need to run

```
$ go get github.com/julienschmidt/httprouter
```

at the console and if we're connected to the internet, it'll download the code from the HttpRouter repository (in GitHub) and store it in the $GOPATH/src directory. Then when we run `go build`, it'll import the code and compile our server.

3.4 Using HTTP/2

Before leaving this chapter, let me show you how you can use HTTP/2 in Go with what you have learned in this chapter.

In chapter 1, you learned about HTTP/2 and how Go 1.6 includes HTTP/2 by default when you start up a server with HTTPS. For older versions of Go, you can enable this manually through the `golang.org/x/net/http2` package.

If you're using a Go version prior to 1.6, the http2 package is not installed by default, so you need to get it using go get:

```
go get "golang.org/x/net/http2"
```

Modify the code from listing 3.6 by importing the http2 package and also adding a line to set up the server to use HTTP/2.

In the following listing you can see by calling the `ConfigureServer` method in the `http2` package, and passing it the server configuration, you have set up the server to run in HTTP/2.

Listing 3.13 Using HTTP/2

```
package main

import (
    "fmt"
    "golang.org/x/net/http2"
    "net/http"
)

type MyHandler struct{}

func (h *MyHandler) ServeHTTP (w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World!")
}

func main() {
    handler := MyHandler{}
    server := http.Server{
        Addr:    "127.0.0.1:8080",
        Handler: &handler,
    }
    http2.ConfigureServer(&server, &http2.Server{})
    server.ListenAndServeTLS("cert.pem", "key.pem")
}
```

Now, run the server:

```
go run server.go
```

To check whether the server is running in HTTP/2, you can use cURL. You will be using cURL quite a bit in this book, because it's widely available on most platforms, so it's a good time to get familiar with it.

cURL

cURL is a command-line tool that allows users to get or send files through a URL. It supports a large number of common internet protocols, including HTTP and HTTPS. cURL is installed by default in many variants of Unix, including OS X, but is also available in Windows. To download and install cURL manually, go to http://curl.haxx.se/download.html.

Starting from version 7.43.0, cURL supports HTTP/2. You can perform a request using the HTTP/2 protocol passing the `--http2` flag. To use cURL with HTTP/2, you need to link it to nghttp2, a C library that provides support for HTTP/2. As of this writing,

many default cURL implementations don't yet support HTTP/2 (including the one shipped with OS X), so if you need to recompile cURL, link it with nghttp2 and replace the previous cURL version with the one you just built.

Once you have done that, you can use cURL to check your HTTP/2 web application:

```
curl -I --http2 --insecure https://localhost:8080/
```

Remember, you need to run it against HTTPS. Because you created your own certificate and private key, by default cURL will not proceed as it will try to verify the certificate. To force cURL to accept your certificate, you need to set the insecure flag.

You should get an output similar to this:

```
HTTP/2.0 200
content-type:text/plain; charset=utf-8
content-length:12
date:Mon, 15 Feb 2016 05:33:01 GMT
```

We've discussed how to handle requests, but we mostly glossed over how to process the incoming request and send responses to the client. Handlers and handler functions are the key to writing web applications in Go, but processing requests and sending responses is the real reason why web applications exist. In the next chapter, we'll turn to the details on requests and responses and you'll see how to extract information from requests and pass on information through responses.

3.5 Summary

- Go has full-fledged standard libraries for building web applications, with net/http and html/template.
- Although using good web frameworks is often easier and saves time, it is important to learn the basics of web programming before using them.
- Go's `net/http` package allows HTTP to be layered on top of SSL to be more secured, creating HTTPS.
- Go handlers can be any struct that has a method named `ServeHTTP` with two parameters: an `HTTPResponseWriter` interface and a pointer to a `Request` struct.
- Handler functions are functions that behave like handlers. Handler functions have the same signature as the `ServeHTTP` method and are used to process requests.
- Handlers and handler functions can be chained to allow modular processing of requests through separation of concerns.
- Multiplexers are also handlers. `ServeMux` is an HTTP request multiplexer. It accepts an HTTP request and redirects it to the correct handler according to the URL in the request. `DefaultServeMux` is a publicly available instance of `ServeMux` that is used as the default multiplexer.
- In Go 1.6 and later, `net/http` supports HTTP/2 by default. Before 1.6, HTTP/2 support can be added manually by using the `http2` package.