4 Processing requests

This chapter covers

- Using Go requests and responses
- Processing HTML forms with Go
- Sending responses back to the client with `ResponseWriter`
- Working with cookies
- Implementing flash messages with cookies

In the previous chapter we explored serving web applications with the built-in net/http library. You also learned about handlers, handler functions, and multiplexers. Now that you know about receiving and handing off the request to the correct set of functions, in this chapter we'll investigate the tools that Go provides to programmers to process requests and send responses back to the client.

4.1 Requests and responses

In chapter 1 we went through quite a bit of information about HTTP messages. If that chapter was a blur to you, now would be a great time to revisit it. HTTP messages are messages sent between the client and the server. There are two types of HTTP messages: HTTP request and HTTP response.

Both requests and responses have basically the same structure:

1. Request or response line
2. Zero or more headers
3. An empty line, followed by …
4. … an optional message body

Here's an example of a GET request:

```
GET /Protocols/rfc2616/rfc2616.html HTTP/1.1
Host: www.w3.org
User-Agent: Mozilla/5.0
(empty line)
```

The net/http library provides structures for HTTP messages, and you need to know these structures to understand how to process requests and send responses. Let's start with the request.

4.1.1 Request

The `Request` struct represents an HTTP request message sent from the client. The representation isn't literal because HTTP requests are only lines of text. The struct contains the parsed information that's considered to be important, as well as a number of useful methods. Some important parts of `Request` are

- URL
- Header
- Body
- Form, PostForm, and MultipartForm

You can also get access to the cookies in the request and the referring URL and the user agent from methods in `Request`. `Request` can either be requests sent to the server, or requests sent from the client, when the net/http library is used as an HTTP client.

4.1.2 Request URL

The URL field of a `Request` is a representation of the URL that's sent as part of the request line (the first line of the HTTP request). The URL field is a pointer to the `url.URL` type, which is a struct with a number of fields, as shown here.

Listing 4.1 The `URL` struct

```
type URL struct {
    Scheme   string
    Opaque   string
    User     *Userinfo
    Host     string
    Path     string
    RawQuery string
    Fragment string
}
```

The general form is

```
scheme://[userinfo@]host/path[?query][#fragment]
```

URLs that don't start with a slash after the scheme are interpreted as

```
scheme:opaque[?query][#fragment]
```

When programming web applications, we often pass information from the client to the server using the URL query. The RawQuery field provides the actual query string that's passed through. For example, if we send a request to the URL http://www.example.com/post?id=123&thread_id=456 RawQuery will contain `id=123&thread_id=456` and we'll need to parse it to get the key-value pairs. There's a convenient way of getting these key-value pairs: through the Form field in the `Request`. We'll get to the Form, PostForm, and MultipartForm fields in a bit.

It's interesting to note that you can't extract the Fragment field out of a `URL` struct if you're getting a request from the browser. Recall from chapter 1 that fragments are stripped by the browser before being sent to the server, so it's not the Go libraries being annoying—it's because the fragment you see on the browser never gets sent to the server. So why have it at all? It's because not all requests come from browsers; you can get requests from HTTP client libraries or other tools, or even client frameworks like Angular. Also, `Request` isn't used only at the server—it can also be used as part of the client library.

4.1.3 Request header

Request and response headers are described in the `Header` type, which is a map representing the key-value pairs in an HTTP header. There are four basic methods on `Header`, which allow you to add, delete, get, and set values, given a key. Both the key and the value are strings.

The difference between adding and setting a value is straightforward but tells us quite a lot about the structure of the `Header` type. A header is a map with keys that are strings and values that are slices of strings. Setting a key creates a blank slice of strings as the value, with the first element in the slice being the new header value. To add a new header value to a given key, you append the new element to the existing slice of string (see figure 4.1).

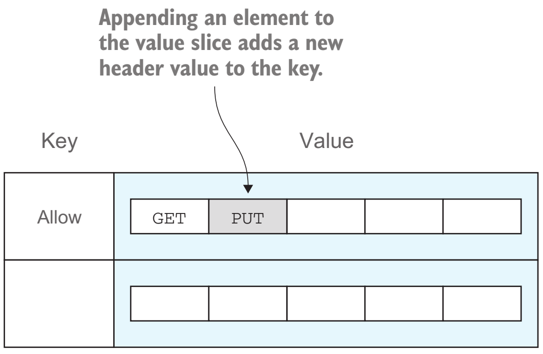

Appending an element to the value slice adds a new header value to the key.
Key
Value
Allow
GET
PUT

Figure 4.1 A header is a map, with the key a string and the value a slice of strings.

This listing shows how you'd read headers from a request.

Listing 4.2 Reading headers from a request

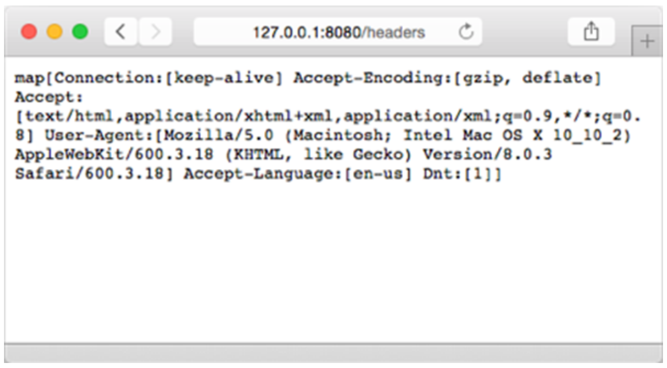
```
package main

import (
    "fmt"
    "net/http"
)

func headers(w http.ResponseWriter, r *http.Request) {
    h := r.Header
    fmt.Fprintln(w, h)
}

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

The previous listing shows the simple server from chapter 3, but this time the handler prints out the header. Figure 4.2 shows what you'll see in your browser (I used Safari on my OS X machine).



```
map[Connection:[keep-alive] Accept-Encoding:[gzip, deflate]
Accept:
[text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
8] User-Agent:[Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2)
AppleWebKit/600.3.18 (KHTML, like Gecko) Version/8.0.3
Safari/600.3.18] Accept-Language:[en-us] Dnt:[1]]
```

Figure 4.2 Header output as viewed in the browser

To get just one particular header, instead of using

```
h := r.Header
```

you'd use

```
h := r.Header["Accept-Encoding"]
```

and you'd get

```
[gzip, deflate]
```

You can also use

```
h := r.Header.Get("Accept-Encoding")
```

which will give you

```
gzip, deflate
```

Notice the difference. If you refer to the `Header` directly, you'll get a map of strings; if you use the `Get` method on the `Header`, then you'll get the comma-delimited list of values (which is the actual string in the header).

4.1.4 Request body

Both request and response bodies are represented by the Body field, which is an `io.ReadCloser` interface. In turn the Body field consists of a `Reader` interface and a `Closer` interface. A `Reader` is an interface that has a `Read` method that takes in a slice of bytes and returns the number of bytes read and an optional error. A `Closer` is an interface that has a `Close` method, which takes in nothing and returns an optional error. What this really means is that you can call on the `Read` and `Close` methods of the Body field. To read the contents of a request body, you can call the Body field's `Read` method, as shown in this listing.

Listing 4.3 Reading data from a request body

```go
package main

import (
    "fmt"
    "net/http"
)

func body(w http.ResponseWriter, r *http.Request) {
    len := r.ContentLength
    body := make([]byte, len)
    r.Body.Read(body)
    fmt.Fprintln(w, string(body))
}

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

Notice that you first need to determine how much to read; then you create a byte array of the content length, and call the `Read` method to read into the byte array.

If you want to test this, you'll need to send a POST request to the server with the appropriate message body, because GET requests don't have message bodies. Remember that you can't normally send POST requests through a browser—you need an HTTP client. There are plenty of choices. You can use a desktop graphical HTTP client, a browser plug-in or extension, or even cURL from the command line.

Type this on your console:

```
$ curl -id "first_name=sausheong&last_name=chang" 127.0.0.1:8080/body
```

cURL will display the full, raw HTTP response, with the HTTP body after the blank line. This is what you should be getting:

```
HTTP/1.1 200 OK
Date: Tue, 13 Jan 2015 16:11:58 GMT
Content-Length: 37
Content-Type: text/plain; charset=utf-8

first_name=sausheong&last_name=chang
```

Normally you wouldn't need to read the raw form of the body, though, because Go provides methods such as `FormValue` and `FormFile` to extract the values from a POST form.

4.2 HTML forms and Go

Before we delve into getting form data from a POST request, let's take a deeper look at HTML forms. Most often, POST requests come in the form (pun intended) of an HTML form and often look like this:

```
<form action="/process" method="post">
  <input type="text" name="first_name"/>
  <input type="text" name="last_name"/>
  <input type="submit"/>
</form>
```

Within the `<form>` tag, we place a number of HTML form elements including text input, text area, radio buttons, checkboxes, and file uploads. These elements allow users to enter data to be submitted to the server. Data is submitted to the server when the user clicks a button or somehow triggers the form submission.

We know the data is sent to the server through an HTTP POST request and is placed in the body of the request. But how is the data formatted? The HTML form data is always sent as name-value pairs, but how are these name-value pairs formatted in the POST body? It's important for us to know this because as we receive the POST request from the browser, we need to be able to parse the data and extract the name-value pairs.

The format of the name-value pairs sent through a POST request is specified by the content type of the HTML form. This is defined using the `enctype` attribute like this:

```
<form action="/process" method="post" enctype="application/x-www-
➥ form-urlencoded">
  <input type="text" name="first_name"/>
  <input type="text" name="last_name"/>
  <input type="submit"/>
</form>
```

The default value for enctype is application/x-www-form-urlencoded. Browsers are required to support at least application/x-www-form-urlencoded and multipart/form-data (HTML5 also supports a text/plain value).

If we set enctype to application/x-www-form-urlencoded, the browser will encode in the HTML form data a long query string, with the name-value pairs separated by an ampersand (&) and the name separated from the values by an equal sign (=). That's the same as URL encoding, hence the name (see chapter 1). In other words, the HTTP body will look something like this:

```
first_name=sau%20sheong&last_name=chang
```

If you set enctype to multipart/form-data, each name-value pair will be converted into a MIME message part, each with its own content type and content disposition. Our form data will now look something like this:

```
------WebKitFormBoundaryMPNjKpeO9cLiocMw
 Content-Disposition: form-data; name="first_name"

sau sheong
 ------WebKitFormBoundaryMPNjKpeO9cLiocMw
 Content-Disposition: form-data; name="last_name"

 chang
 ------WebKitFormBoundaryMPNjKpeO9cLiocMw--
```

When would you use one or the other? If you're sending simple text data, the URL encoded form is better—it's simpler and more efficient and less processing is needed. If you're sending large amounts of data, such as uploading files, the multipart-MIME form is better. You can even specify that you want to do Base64 encoding to send binary data as text.

So far we've only talked about POST requests—what about GET requests in an HTML form? HTML allows the method attribute to be either POST or GET, so this is also a valid format:

```
<form action="/process" method="get">
  <input type="text" name="first_name"/>
  <input type="text" name="last_name"/>
  <input type="submit"/>
</form>
```

In this case, there's no request body (GET requests have no request body), and all the data is set in the URL as name-value pairs.

Now that you know how data is sent from an HTML form to the server, let's go back to the server and see how you can use net/http to process the request.

4.2.1 Form

In the previous sections, we talked about extracting data from the URL and the body in the raw form, which requires us to parse the data ourselves. In fact, we normally don't need to, because the net/http library includes a rather comprehensive set of functions that normally provides us with all we need. Let's talk about each in turn.

The functions in `Request` that allow us to extract data from the URL and/or the body revolve around the Form, PostForm, and MultipartForm fields. The data is in the form of key-value pairs (which is what we normally get from a POST request anyway). The general algorithm is:

1. Call `ParseForm` or `ParseMultipartForm` to parse the request.
2. Access the Form, PostForm, or MultipartForm field accordingly.

This listing shows parsing forms.

Listing 4.4 Parsing forms

```
package main

import (
    "fmt"
    "net/http"
)

func process(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    fmt.Fprintln(w, r.Form)
}

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

The focus of the server in listing 4.4 is on these two lines:

```
r.ParseForm()
fmt.Fprintln(w, r.Form)
```

As mentioned earlier, you need to first parse the request using `ParseForm`, and then access the Form field.

Let's look at the client that's going to call this server. You'll create a simple, minimal HTML form to send the request to the server. Place the code in a file named client.html.

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Go Web Programming</title>
  </head>
  <body>
    <form action=http://127.0.0.1:8080/process?hello=world&thread=123
    ➥ method="post" enctype="application/x-www-form-urlencoded">
```

```
      <input type="text" name="hello" value="sau sheong"/>
      <input type="text" name="post" value="456"/>
      <input type="submit"/>
    </form>
  </body>
</html>
```

In this form, you are

- Sending the URL http://localhost:8080/process?hello=world&thread=123 to the server using the POST method
- Specifying the content type (in the enctype field) to be `application/x-www-form-urlencoded`
- Sending two HTML form key-value pairs—`hello=sau sheong` and `post=456`—to the server

Note that you have two values for the key `hello`. One of them is `world` in the URL and the other is `sau sheong` in the HTML form.

Open the client.html file directly in your browser (you don't need to serve it out from a web server—just running it locally on your browser is fine) and click the submit button. You'll see the following in the browser:

```
map[thread:[123] hello:[sau sheong world] post:[456]]
```

This is the raw string version of the `Form` struct in the POST request, after the request has been parsed. The `Form` struct is a map, whose keys are strings and values are a slice of strings. Notice that the map isn't sorted, so you might get a different sorting of the returned values. Nonetheless, what we get is the combination of the query values `hello=world` and `thread=123` as well as the form values `hello=sau sheong` and `post=456`. As you can see, the values are URL decoded (there's a space between `sau` and `sheong`).

4.2.2 PostForm

Of course, if you only want to get the value to the key `post`, you can use `r.Form["post"]`, which will give you a map with one element: `[456]`. If the form and the URL have the same key, both of them will be placed in a slice, with the form value always prioritized before the URL value.

What if you need just the form key-value pairs and want to totally ignore the URL key-value pairs? For that you can use the PostForm field, which provides key-value pairs only for the form and not the URL. If you change from using `r.Form` to using `r.PostForm` in the code, this is what you get:

```
map[post:[456] hello:[sau sheong]]
```

This example used `application/x-www-form-urlencoded` for the content type. What happens if you use `multipart/form-data`? Make the change to the client HTML form, switch back to using `r.Form`, and let's find out:

```
map[hello:[world] thread:[123]]
```

What happened here? You only get the URL query key-value pairs this time and not the form key-value pairs, because the PostForm field only supports `application/x-www-form-urlencoded`. To get multipart key-value pairs from the body, you must use the `MultipartForm` field.

4.2.3 MultipartForm

Instead of using the `ParseForm` method on the `Request` struct and then using the Form field on the request, you have to use the `ParseMultipartForm` method and then use the MultipartForm field on the request. The `ParseMultipartForm` method also calls the `ParseForm` method when necessary.

```
r.ParseMultipartForm(1024)
fmt.Fprintln(w, r.MultipartForm)
```

You need to tell the `ParseMultipartForm` method how much data you want to extract from the multipart form, in bytes. Now let's see what happens:

```
&{map[hello:[sau sheong] post:[456]] map[]}
```

This time you see the form key-value pairs but not the URL key-value pairs. This is because the MultipartForm field contains only the form key-value pairs. Notice that the returned value is no longer a map but a struct that contains two maps. The first map has keys that are strings and values that are slices of string; the second map is empty. It's empty because it's a map with keys that are strings but values that are files, which we're going to talk about in the next section.

There's one last set of methods on `Request` that allows you to access the key-value pairs even more easily. The `FormValue` method lets you access the key-value pairs just like in the Form field, except that it's for a specific key and you don't need to call the `ParseForm` or `ParseMultipartForm` methods beforehand—the `FormValue` method does that for you.

Taking our previous example, this means if you do this in your handler function:

```
fmt.Fprintln(w, r.FormValue("hello"))
```

and set the client's form `enctype` attribute to `application/x-www-form-urlencoded`, you'll get this:

```
sau sheong
```

That's because the `FormValue` method retrieves only the first value, even though you actually have both values in the `Form` struct. To prove this, let's add another line below the earlier line of code, like this:

```
fmt.Fprintln(w, r.FormValue("hello"))
fmt.Fprintln(w, r.Form)
```

This time you'll see

```
sau sheong
map[post:[456] hello:[sau sheong world] thread:[123]]
```

The PostFormValue method does the same thing, except that it's for the PostForm field instead of the Form field. Let's make some changes to the code to use the PostFormValue method:

```
fmt.Fprintln(w, r.PostFormValue("hello"))
fmt.Fprintln(w, r.PostForm)
```

This time you get this instead:

```
sau sheong
map[hello:[sau sheong] post:[456]]
```

As you can see, you get only the form key-value pairs.

Both the FormValue and PostFormValue methods call the ParseMultipartForm method for you so you don't need to call it yourself, but there's a slightly confusing gotcha that you should be careful with (at least as of Go 1.4). If you set the client form's enctype to multipart/form-data and try to get the value using either the FormValue or the PostFormValue method, you won't be able to get it even though the MultipartForm method has been called!

To help clarify, let's make changes to the server's handler function again:

```
fmt.Fprintln(w, "(1)", r.FormValue("hello"))
fmt.Fprintln(w, "(2)", r.PostFormValue("hello"))
fmt.Fprintln(w, "(3)", r.PostForm)
fmt.Fprintln(w, "(4)", r.MultipartForm)
```

Here's the result from using our form with enctype set to multipart/form-data:

```
(1) world
(2)
(3) map[]
(4) &{map[hello:[sau sheong] post:[456]] map[]}
```

The first line in the results gives you the value for hello that's found in the URL and not the form. The second and third lines tell you why, because if you just take the form key-value pairs, you get nothing. That's because the FormValue and PostFormValue methods correspond to the Form and PostForm fields, and not the MultipartForm field. The last line in the results proves that the ParseMultipartForm method was actually called—that's why if you try to access the MultipartForm field you'll get the data there.

We covered quite a bit in these sections, so let's recap, in table 4.1, how these functions are different. The table shows the methods that should be called if you're looking for values in the corresponding fields. The table also shows where the data comes from and what type of data you'll get. For example, in the first row, if you're looking for data in the Form field, you should be calling the ParseForm method (either directly or indirectly). You'll then get both the URL data and the form data from the request and the data will be URL-encoded. Undoubtedly the naming convention leaves much to be desired!

Table 4.1 Comparing Form, PostForm, and MultipartForm fields

		Key-value pairs from		Content type	
Field	**Should call method**	**URL**	**Form**	**URL encoded**	**Multipart**
Form	`ParseForm`	✓	✓	✓	-
PostForm	Form	-	✓	✓	-
MultipartForm	`ParseMultipartForm`	-	✓	-	✓
FormValue	NA	✓	✓	✓	-
PostFormValue	NA	-	✓	✓	-

4.2.4 *Files*

Probably the most common use for `multipart/form-data` is for uploading files. This mostly means the file HTML tag, so let's make some changes, shown in bold in the following listing, to our client form.

Listing 4.5 Uploading files

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Go Web Programming</title>
  </head>
  <body>
    <form action="http://localhost:8080/process?hello=world&thread=123"
  method="post" enctype="multipart/form-data">
      <input type="text" name="hello" value="sau sheong"/>
      <input type="text" name="post" value="456"/>
      <input type="file" name="uploaded">
      <input type="submit">
    </form>
  </body>
</html>
```

To receive files, we'll make changes, shown next, in our handler function.

Listing 4.6 Receiving uploaded files using the MultipartForm field

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func process(w http.ResponseWriter, r *http.Request) {
    r.ParseMultipartForm(1024)
    fileHeader := r.MultipartForm.File["uploaded"][0]
    file, err := fileHeader.Open()
    if err == nil {
        data, err := ioutil.ReadAll(file)
        if err == nil {
            fmt.Fprintln(w, string(data))
        }
    }
}

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

As mentioned earlier, you need to call the `ParseMultipartForm` method first. After that you get a FileHeader from the File field of the MultipartForm field and call its `Open` method to get the file. If you upload a text file and send it across to the server, the handler will get the file, read it into a byte array, and then print it out to the browser.

As with the `FormValue` and `PostFormValue` methods, there's a shorter way of getting an uploaded file using the `FormFile` method, shown in the following listing. The `FormFile` method returns the first value given the key, so this approach is normally faster if you have only one file to be uploaded.

Listing 4.7 Retrieving uploaded files using FormFile

```
func process(w http.ResponseWriter, r *http.Request) {
    file, _, err := r.FormFile("uploaded")
    if err == nil {
        data, err := ioutil.ReadAll(file)
        if err == nil {
            fmt.Fprintln(w, string(data))
        }
    }
}
```

As you can see, you no longer have to call the `ParseMultipartForm` method, and the `FormFile` method returns both the file and the file header at the same time. You simply need to process the file that's returned.

4.2.5 Processing POST requests with JSON body

So far in our discussion we've focused on name-value pairs in the request body. This is because we've been focusing on HTML forms only. But not all POST requests will come from HTML forms. Sending POST requests is increasingly common with client libraries such as JQuery as well as client-side frameworks such as Angular or Ember, or even the older Adobe Flash or Microsoft Silverlight technologies.

If you're trying to get the JSON data from a POST request sent by an Angular client and you're calling the `ParseForm` method to get the data, you won't be able to. At the same time, other JavaScript libraries like JQuery allow you to do so. What gives?

Client frameworks encode their POST requests differently. JQuery encodes POST requests like an HTML form with `application/x-www-form-urlencoded` (that is, it sets the request header `Content-Type` to `application/x-www-form-urlencoded`); Angular encodes POST requests with `application/json`. Go's `ParseForm` method only parses forms and so doesn't accept `application/json`. If you call the `ParseForm` method, you won't get any data at all!

The problem doesn't lie with the implementation of any of the libraries. It lies in the lack of sufficient documentation (although there will arguably never be enough documentation) and the programmer making certain assumptions based on their dependency on frameworks.

Frameworks help programmers by hiding the underlying complexities and implementation details. As a programmer you should be using frameworks. But it's also important to understand how things work and what the framework simplifies for you because eventually there will be cases where the framework's seams show at the joints.

We've covered quite a lot on processing requests. Now let's look at sending responses to the user.

4.3 ResponseWriter

If you were thinking that sending a response to the client would involve creating a `Response` struct, setting up the data in it, and sending it out, then you'd be wrong. The correct interface to use when sending a response from the server to the client is `ResponseWriter`.

`ResponseWriter` is an interface that a handler uses to create an HTTP response. The actual struct backing up `ResponseWriter` is the nonexported struct `http.response`. Because it's nonexported, you can't use it directly; you can only use it through the `ResponseWriter` interface.

Why do we pass ResponseWriter into ServeHTTP by value?

Having read the earlier part of this chapter, you might wonder why the ServeHTTP function takes two parameters—the `ResponseWriter` interface and a pointer to a `Request` struct. The reason why it's a pointer to `Request` is simple: changes to `Request` by the handler need to be visible to the server, so we're only passing it by reference instead of by value. But why are we passing in a `ResponseWriter` by value? The server needs to know the changes to `ResponseWriter` too, doesn't it?

If you dig into the net/http library code, you'll find that `ResponseWriter` is an interface to a nonexported struct response, and we're passing the struct by reference (we're passing in a pointer to response) and not by value.

In other words, both the parameters are passed in by reference; it's just that the method signature takes a `ResponseWriter` that's an interface to a pointer to a struct, so it looks as if it's passed in by value.

The `ResponseWriter` interface has three methods:

- `Write`
- `WriteHeader`
- `Header`

4.3.1 Writing to the ResponseWriter

The `Write` method takes in an array of bytes, and this gets written into the body of the HTTP response. If the header doesn't have a content type by the time `Write` is called, the first 512 bytes of the data are used to detect the content type. This listing shows how to use the `Write` method.

Listing 4.8 `Write` to send responses to the client

```
package main

import (
    "net/http"
)

func writeExample(w http.ResponseWriter, r *http.Request) {
    str := `<html>
<head><title>Go Web Programming</title></head>
<body><h1>Hello World</h1></body>
</html>`
    w.Write([]byte(str))
}

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

In listing 4.8 you're writing an HTML string to the HTTP response body using `ResponseWriter`. You send this command through cURL:

```
curl -i 127.0.0.1:8080/write
HTTP/1.1 200 OK
Date: Tue, 13 Jan 2015 16:16:13 GMT
Content-Length: 95
Content-Type: text/html; charset=utf-8

<html>
<head><title>Go Web Programming</title></head>
<body><h1>Hello World</h1></body>
</html>
```

Notice that you didn't set the content type, but it was detected and set correctly.

The `WriteHeader` method's name is a bit misleading. It doesn't allow you to write any headers (you use `Header` for that), but it takes an integer that represents the status code of the HTTP response and writes it as the return status code for the HTTP response. After calling this method, you can still write to the `ResponseWriter`, though you can no longer write to the header. If you don't call this method, by default when you call the `Write` method, 200 OK will be sent as the response code.

The `WriteHeader` method is pretty useful if you want to return error codes. Let's say you're writing an API and though you defined the interface, you haven't fleshed it out, so you want to return a 501 Not Implemented status code. Let's see how this works by adding a new handler function to our existing server, shown in the following listing. Remember to register this to DefaultServeMux by calling the `HandleFunc` function!

Listing 4.9 Writing headers to responses using `WriteHeader`

```
package main

import (
    "fmt"
    "net/http"
)

func writeExample(w http.ResponseWriter, r *http.Request) {
    str := `<html>
<head><title>Go Web Programming</title></head>
<body><h1>Hello World</h1></body>
</html>`
    w.Write([]byte(str))
}

func writeHeaderExample(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(501)
    fmt.Fprintln(w, "No such service, try next door")
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/write", writeExample)
    http.HandleFunc("/writeheader", writeHeaderExample)
    server.ListenAndServe()
}
```

Call the URL through cURL:

```
curl -i 127.0.0.1:8080/writeheader
HTTP/1.1 501 Not Implemented
Date: Tue, 13 Jan 2015 16:20:29 GMT
Content-Length: 31
Content-Type: text/plain; charset=utf-8
```

```
No such service, try next door
```

Finally the `Header` method returns a map of headers that you can modify (refer to section 4.1.3). The modified headers will be in the HTTP response that's sent to the client.

Listing 4.10 Writing headers to redirect the client

```go
package main

import (
    "fmt"
    "net/http"
)

func writeExample(w http.ResponseWriter, r *http.Request) {
    str := `<html>
<head><title>Go Web Programming</title></head>
<body><h1>Hello World</h1></body>
</html>`
    w.Write([]byte(str))
}

func writeHeaderExample(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(501)
    fmt.Fprintln(w, "No such service, try next door")
}

func headerExample(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Location", "http://google.com")
    w.WriteHeader(302)
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/write", writeExample)
    http.HandleFunc("/writeheader", writeHeaderExample)
    http.HandleFunc("/redirect", headerExample)
    server.ListenAndServe()
}
```

The previous listing shows how a redirect works—it's simple to set the status code to 302 and then add a header named `Location` with the value of the location you want the user to be redirected to. Note that you must add the `Location` header before writing the status code because `WriteHeader` prevents the header from being modified after it's called. When you call the URL from the browser, you'll be redirected to Google.

If you use cURL, you will see this:

```
curl -i 127.0.0.1:8080/redirect
HTTP/1.1 302 Found
Location: http://google.com
```

```
Date: Tue, 13 Jan 2015 16:22:16 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Let's look at one last example of how to use `ResponseWriter` directly. This time, you want to return JSON to the client. Assuming that you have a struct named `Post`, the following listing shows the handler function.

Listing 4.11 Writing JSON output

```
package main

import (
    "fmt"
    "encoding/json"
    "net/http"
)

type Post struct {
    User    string
    Threads []string
}

func writeExample(w http.ResponseWriter, r *http.Request) {
    str := `<html>
<head><title>Go Web Programming</title></head>
<body><h1>Hello World</h1></body>
</html>`
    w.Write([]byte(str))
}

func writeHeaderExample(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(501)
    fmt.Fprintln(w, "No such service, try next door")
}

func headerExample(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Location", "http://google.com")
    w.WriteHeader(302)
}

func jsonExample(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    post := &Post{
        User:    "Sau Sheong",
        Threads: []string{"first", "second", "third"},
    }
    json, _ := json.Marshal(post)
    w.Write(json)
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/write", writeExample)
    http.HandleFunc("/writeheader", writeHeaderExample)
    http.HandleFunc("/redirect", headerExample)
    http.HandleFunc("/json", headerExample)
    server.ListenAndServe()
}
```

Focus only on the `ResponseWriter`. It's okay if you don't understand the JSON bits yet—we'll be covering JSON in chapter 7. Just know that the variable `json` is a JSON string that's marshaled from a `Post` struct.

First you set the content type to `application/json` using `Header`; then you write the JSON string to the `ResponseWriter`. If you call this using cURL, you will see:

```
curl -i 127.0.0.1:8080/json
HTTP/1.1 200 OK
Content-Type: application/json
Date: Tue, 13 Jan 2015 16:27:01 GMT
Content-Length: 58

{"User":"Sau Sheong","Threads":["first","second","third"]}
```

4.4 Cookies

In chapter 2, you saw how to use cookies to create sessions for authentication. In this section, we'll delve into the details of using cookies not just for sessions but for persistence at the client in general.

A cookie is a small piece of information that's stored at the client, originally sent from the server through an HTTP response message. Every time the client sends an HTTP request to the server, the cookie is sent along with it. Cookies are designed to overcome the stateless-ness of HTTP. Although it's not the only mechanism that can be used, it's one of the most common and popular methods. Entire industries' revenues depend on it, especially in the internet advertising domain.

There are a number of types of cookies, including interestingly named ones like super cookies, third-party cookies, and zombie cookies. But generally there are only two classes of cookies: session cookies and persistent cookies. Most other types of cookies are variants of the persistent cookies.

4.4.1 Cookies with Go

The `Cookie` struct, shown in this listing, is the representation of a cookie in Go.

Listing 4.12 The `Cookie` struct

```
type Cookie struct {
  Name       string
  Value      string
  Path       string
  Domain     string
  Expires    time.Time
  RawExpires string
  MaxAge     int
  Secure     bool
  HttpOnly   bool
  Raw        string
  Unparsed   []string
}
```

If the Expires field isn't set, then the cookie is a session or temporary cookie. Session cookies are removed from the browser when the browser is closed. Otherwise, the cookie is a persistent cookie that'll last as long as it isn't expired or removed.

There are two ways of specifying the expiry time: the Expires field and the MaxAge field. Expires tells us exactly when the cookie will expire, and MaxAge tells us how long the cookie should last (in seconds), starting from the time it's created in the browser. This isn't a design issue with Go, but rather results from the inconsistent implementation differences of cookies in various browsers. Expires was deprecated in favor of MaxAge in HTTP 1.1, but almost all browsers still support it. MaxAge isn't supported by Microsoft Internet Explorer 6, 7, and 8. The pragmatic solution is to use only Expires or to use both in order to support all browsers.

4.4.2 Sending cookies to the browser

Cookie has a `String` method that returns a serialized version of the cookie for use in a `Set-Cookie` response header. Let's take a closer look.

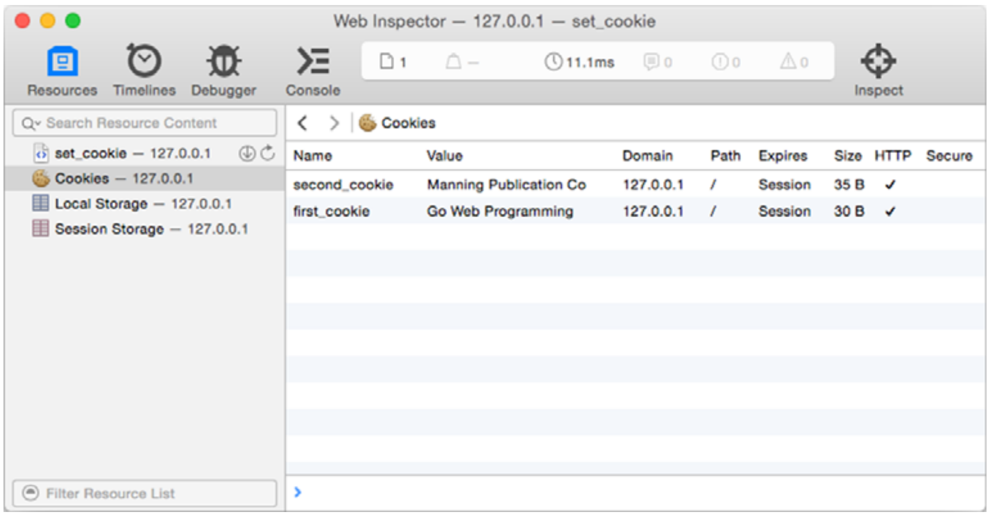
Listing 4.13 Sending cookies to the browser

```
package main

import (
    "net/http"
)

func setCookie(w http.ResponseWriter, r *http.Request) {
    c1 := http.Cookie{
        Name:     "first_cookie",
        Value:    "Go Web Programming",
        HttpOnly: true,
    }
    c2 := http.Cookie{
        Name:     "second_cookie",
        Value:    "Manning Publications Co",
        HttpOnly: true,
    }
    w.Header().Set("Set-Cookie", c1.String())
    w.Header().Add("Set-Cookie", c2.String())
}

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



Name	Value	Domain	Path	Expires	Size	HTTP	Secure
second_cookie	Manning Publication Co	127.0.0.1	/	Session	35 B	✓	
first_cookie	Go Web Programming	127.0.0.1	/	Session	30 B	✓	

Figure 4.3 Viewing cookies on Web Inspector (Safari)

```
    http.HandleFunc("/set_cookie", setCookie)
    server.ListenAndServe()
}
```

You use the `Set` method to add the first cookie and then the `Add` method to add a second cookie. Go to http://127.0.0.1:8080/set_cookie in your browser, and then inspect the list of cookies set on the browser. (In figure 4.3 I'm using Web Inspector in Safari, but any corresponding tool on other browsers should show the same thing.)

Go provides a simpler shortcut to setting the cookie: using the `SetCookie` function in the net/http library. Taking the example from listing 4.13, let's make a change (shown in bold in the following listing) to the response header.

Listing 4.14 Sending cookies to the browser using `SetCookie`

```
func setCookie(w http.ResponseWriter, r *http.Request) {
    c1 := http.Cookie{
        Name:      "first_cookie",
        Value:     "Go Web Programming",
        HttpOnly: true,
    }
    c2 := http.Cookie{
        Name:      "second_cookie",
        Value:     "Manning Publications Co",
        HttpOnly: true,
    }
    http.SetCookie(w, &c1)
    http.SetCookie(w, &c2)
}
```

It doesn't make too much of a difference, though you should take note that you need to pass in the cookies by reference instead.

4.4.3 Getting cookies from the browser

Now that you can set a cookie, you want to also retrieve that cookie from the client. This listing demonstrates how.

Listing 4.15 Getting cookies from the header

```
package main

import (
    "fmt"
    "net/http"
)

func setCookie(w http.ResponseWriter, r *http.Request) {
    c1 := http.Cookie{
        Name:     "first_cookie",
        Value:    "Go Web Programming",
        HttpOnly: true,
    }
    c2 := http.Cookie{
        Name:     "second_cookie",
        Value:    "Manning Publications Co",
        HttpOnly: true,
    }
    http.SetCookie(w, &c1)
    http.SetCookie(w, &c2)
}

func getCookie(w http.ResponseWriter, r *http.Request) {
    h := r.Header["Cookie"]
    fmt.Fprintln(w, h)
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/set_cookie", setCookie)
    http.HandleFunc("/get_cookie", getCookie)
    server.ListenAndServe()
}
```

After you recompile and start the server, when you go to http://127.0.0.1:8080/get_cookie you'll see this in the browser:

```
[first_cookie=Go Web Programming; second_cookie=Manning Publications Co]
```

This is a slice, with a single string. If you want to get the individual name-value pairs you'll have to parse the string yourself. But as you can see in the following listing, Go provides a couple of easy ways to get cookies.

Listing 4.16 Using the `Cookie` and `Cookies` methods

```
package main

import (
    "fmt"
    "net/http"
)

func setCookie(w http.ResponseWriter, r *http.Request) {
    c1 := http.Cookie{
        Name:     "first_cookie",
        Value:    "Go Web Programming",
        HttpOnly: true,
    }
    c2 := http.Cookie{
        Name:     "second_cookie",
        Value:    "Manning Publications Co",
        HttpOnly: true,
    }
    http.SetCookie(w, &c1)
    http.SetCookie(w, &c2)
}

func getCookie(w http.ResponseWriter, r *http.Request) {
    c1, err := r.Cookie("first_cookie")
    if err != nil {
        fmt.Fprintln(w, "Cannot get the first cookie")
    }
    cs := r.Cookies()
    fmt.Fprintln(w, c1)
    fmt.Fprintln(w, cs)
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/set_cookie", setCookie)
    http.HandleFunc("/get_cookie", getCookie)
    server.ListenAndServe()
}
```

Go provides a `Cookie` method on `Request` (shown in bold text in listing 4.16) that allows you to retrieve a named cookie. If the cookie doesn't exist, it'll throw an error. This is a single value, though, so if you want to get multiple cookies you can use the `Cookies` method on `Request`. That way, you retrieve all cookies into a Go slice; in fact, it's the same as getting it through the `Header` yourself. If you recompile, restart the server, and go to http://127.0.0.1:8080/get_cookie now, you'll see this in your browser:

```
first_cookie=Go Web Programming
[first_cookie=Go Web Programming second_cookie=Manning Publications Co]
```

We didn't set the Expires or the MaxAge fields when we set the cookie, so what was returned are session cookies. To prove the point, quit your browser (don't just close the tab or window; completely quit your browser). Then go to http://127.0.0.1:8080/get_cookie again and you'll see that the cookies are gone.

4.4.4 Using cookies for flash messages

In chapter 2 we looked at using cookies for managing sessions, so let's try out our newfound cookie skills on something else.

Sometimes it's necessary to show users a short informational message telling them whether or not they've successfully completed an action. If the user submits a post to a forum and his posting fails, you'll want to show him a message that tells him that the post didn't go through. Following the Principle of Least Surprise from the previous chapter, you want to show the message on the same page. But this page doesn't normally show any messages, so you want the message to show on certain conditions and it must be *transient* (which means it doesn't show again when the page is refreshed). These transient messages are commonly known as *flash messages*.

There are many ways to implement flash messages, but one of the most common is to store them in session cookies that are removed when the page is refreshed. This listing shows how you can do this in Go.

Listing 4.17 Implementing flash messaging using Go cookies

```go
package main

import (
    "encoding/base64"
    "fmt"
    "net/http"
    "time"
)

func setMessage(w http.ResponseWriter, r *http.Request) {
    msg := []byte("Hello World!")
    c := http.Cookie{
        Name:  "flash",
        Value: base64.URLEncoding.EncodeToString(msg),
    }
    http.SetCookie(w, &c)
}

func showMessage(w http.ResponseWriter, r *http.Request) {
    c, err := r.Cookie("flash")
    if err != nil {
        if err == http.ErrNoCookie {
            fmt.Fprintln(w, "No message found")
        }
    } else {
        rc := http.Cookie{
            Name:    "flash",
            MaxAge:  -1,
            Expires: time.Unix(1, 0),
        }
        http.SetCookie(w, &rc)
        val, _ := base64.URLEncoding.DecodeString(c.Value)
        fmt.Fprintln(w, string(val))
    }
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/set_message", setMessage)
    http.HandleFunc("/show_message", showMessage)
    server.ListenAndServe()
}
```

You create two handler functions, `setMessage` and `showMessage`, and attach them to `/set_message` and `/show_message`, respectively. Let's start with `setMessage`, which is straightforward.

Listing 4.18 Setting the message

```
func setMessage(w http.ResponseWriter, r *http.Request) {
    msg := []byte("Hello World!")
    c := http.Cookie{
        Name:  "flash",
        Value: base64.URLEncoding.EncodeToString(msg),
    }
    http.SetCookie(w, &c)
}
```

This isn't much different from the `setCookie` handler function from earlier, except this time you do a Base64 URL encoding of the message. You do so because the cookie values need to be URL encoded in the header. You managed to get away with it earlier because you didn't have any special characters like a space or the percentage sign, but you can't get away with here because messages will eventually need to have them.

Now let's look at the `showMessage` function:

```
func showMessage(w http.ResponseWriter, r *http.Request) {
    c, err := r.Cookie("flash")
    if err != nil {
        if err == http.ErrNoCookie {
            fmt.Fprintln(w, "No message found")
        }
    } else {
        rc := http.Cookie{
            Name:    "flash",
            MaxAge:  -1,
            Expires: time.Unix(1, 0),
        }
        http.SetCookie(w, &rc)
        val, _ := base64.URLEncoding.DecodeString(c.Value)
        fmt.Fprintln(w, string(val))
    }
}
```

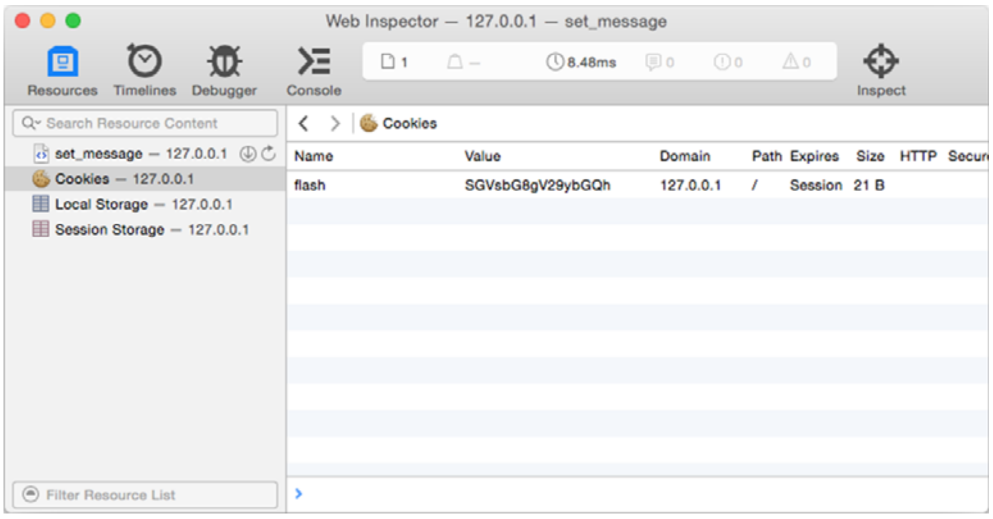
First, you get the cookie. If you can't find the cookie (`err` will have a value of `http.ErrNoCookie`), you'll show the message "No message found."

If you find the message, you have to do two things:

1. Create a cookie with the same name, but with MaxAge set to a negative number and an Expires value that's in the past.
2. Send the cookie to the browser with `SetCookie`.

Here you're replacing the existing cookie, essentially removing it altogether because the MaxAge field is a negative number and the Expires field is in the past. Once you do that, you can decode your string and show the value.

Now let's see that in action. Start up your browser and go to http://localhost:8080/set_message. Figure 4.4 shows what you'll see in Web Inspector.



Name	Value	Domain	Path	Expires	Size	HTTP	Secur
flash	SGVsbG8gV29ybGQh	127.0.0.1	/	Session	21 B		

Figure 4.4 The flash message cookie in Web Inspector (Safari)

Notice that the value is Base64 URL encoded. Now, using the same browser window, go to http://localhost:8080/show_message. This is what you should see in the browser:

```
Hello World!
```

Go back to the Web Inspector and look at the cookies. Your cookie is gone! Setting a cookie with the same name to the browser will replace the old cookie with the new cookie of the same name. Because the new cookie has a negative number for MaxAge and expires in some time in the past, this tells the browser to remove the cookie, which means the earlier cookie you set is removed.

This is what you'll see in the browser:

```
No message found
```

This chapter wraps up our two-part tour of what net/http offers for web application development on the server. In the next chapter, we move on to the next big component in a web application: templates. I will cover template engines and templates in Go and show you how they can be used to generate responses to the client.

4.5 Summary

- Go provides a representation of the HTTP requests through various structs, which can be used to extract data from the requests.
- The Go `Request` struct has three fields, Form, PostForm, and MultipartForm, that allow easy extraction of different data from a request. To get data from these fields, call `ParseForm` or `ParseMultipartForm` to parse the request and then access the Form, PostForm, or MultipartForm field accordingly.
- Form is used for URL-encoded data from the URL and HTML form, PostForm is used for URL-encoded data from the HTML form only, and MultipartForm is used for multi-part data from the URL and HTML form.
- To send data back to the client, write header and body data to `ResponseWriter`.
- To persist data at the client, send cookies in the `ResponseWriter`.
- Cookies can be used for implementing flash messages.