**Creating HTTP Servers**

**What You’ll Learn in This Hour:**

Image[Announcing the presence of your web server](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev1sec1)

Image[Examining requests and responses](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev1sec2)

Image[Working with handler functions](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev1sec3)

Image[Handling 404s](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev1sec4)

Image[Setting a header](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev1sec5)

Image[Responding with different content types](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev2sec3)

Image[Responding to different types of requests](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev1sec6)

Image[Receiving data from GET and Post requests](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18lev1sec7)

Go offers strong support for creating web servers that can serve web pages, web services, and files. During this hour, you will learn how to create web servers that can respond to different routes, different types of requests, and different content types. By taking advantage of Go’s approach to concurrency, writing web servers in Go is a great option.

**Announcing Your Presence with the “Hello World” Web Server**

The net/http standard library package provides multiple methods for creating HTTP servers, and comes with a basic router. It is traditional to create a Hello World program to announce a basic presence to the world. The most basic HTTP server in Go is shown in [Listing 18.1](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ex01).

**LISTING 18.1** Basic HTTP Server in Go

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p18ex01a)

 1:  package main  
 2:  
 3:  import (  
 4:      "net/http"  
 5:  )  
 6:  
 7:  func helloWorld(w http.ResponseWriter, r \*http.Request) {  
 8:      w.Write([]byte("Hello World\n"))  
 9:  }  
10:  
11:  func main() {  
12:      http.HandleFunc("/", helloWorld)  
13:      http.ListenAndServe(":8000", nil)  
14:  }

Although the program is just 14 lines, plenty is going on.

ImageThe net/http package is imported.

ImageWithin the main function, a route / is created using the HandleFunc method. This takes a pattern describing a path, followed by a function that defines how to respond to a request to that path.

ImageThe helloWorld function takes a http.ResponseWriter and a pointer to the request. This means that within the function, the request can be examined or manipulated before returning a response to the client. In this case, the Write method is used to write the response. This writes the HTTP response including status, headers, and the body. The usage of []byte initializes a byte slice and converts the string value into bytes. This means it can be used by the Write method, which expects a slice of bytes.

ImageThe ListenAndServe method is used to start a server to respond to a client that listens on localhost and port 8000.

Although this is a short example, if you can begin to understand how a web server operates in Go, you will be well on your way to creating more complex programs.

Try It Yourself: Running a “Hello World” Web Server

In this example, you run a Hello World web server:

**1.** Open the file hour18/example01.go in a text editor and try to understand what the example is doing. If you need to, refer to the previous bullet points and step through the code.

**2.** From the terminal, run the program with go run example01.go.

**3.** Open a web browser at http://localhost:8000.

**4.** You should see “Hello World” in your web browser.

**5.** Congratulations! You just ran your first web server in Go.

**Examining Requests and Responses**

As this hour progresses, we will examine requests and responses that are being sent and received. The tool we use for this is curl.

Curl is a command-line tool for making HTTP requests, and is generally available on all platforms. On macOS, curl is pre-installed. On Linux, curl is often installed and is available through package managers. For Windows, curl is not pre-installed. For Windows users, installing GIT for Windows was recommended in an earlier chapter. If you have not done this, visit <https://git-scm.com/download/win>, open the download file, and install it. Once installed, you will have a new Start menu item: Git Bash. Open it.

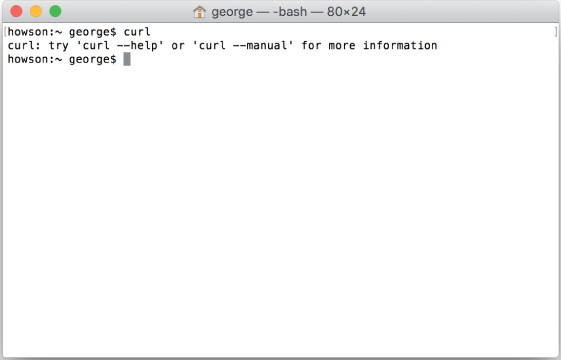
To verify everything is installed correctly, follow these steps:

**1a.** On macOS or Linux, open a terminal.

**1b.** On Windows, open “Git Bash” from the Start menu.

**2.** Type curl and hit Return.

**3.** If curl is installed successfully, you will see the output shown in [Figure 18.1](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18fig01).



**FIGURE 18.1** Verifying curl is correctly installed.

**Making a Request with curl**

With curl installed, you can use it for developing and debugging web servers. Rather than using a browser, you can use curl to send a variety of requests to a web server and to examine the response. To make a request to the Hello World web server, open a terminal and run the server.

go run example01.go

On macOS or Linux, open another terminal tab or window. On Windows, switch to Git Bash. Run the following command. The -is options mean that headers are printed and that some unwanted output is ignored.

curl -is http://localhost:8000

If the command was successful, you will see a response from the web server that includes the headers and the response body.

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p246pro03a)

HTTP/1.1 200 OK  
Date: Wed, 16 Nov 2016 16:45:51 GMT  
Content-Length: 12  
Content-Type: text/plain; charset=utf-8  
  
Hello World

The output may be explained as follows:

ImageThe response uses the HTTP 1.1 protocol, and a 200 response was received.

ImageThe Date header details when the response was sent.

ImageThe Content-Length header details the length of the response. In this case, it is 12 bytes.

ImageThe Content-Type header details the type of content and the encoding used. In this case, the response is text/plain encoded using utf-8.

ImageFinally, the response body is outputted. In this case, it is Hello World.

**Routing in More Detail**

The HandleFunc registers functions to respond to URL address mappings. In simplistic terms, HandleFunc sets up a routing table that allows the HTTP server to respond correctly.

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p247pro01a)

http.HandleFunc("/", helloWorld)  
http.HandleFunc("/users/", usersHandler)  
http.HandleFunc("/projects/", projectsHandler)

In this example, whenever a request is made to /, the helloWorld function will be called. Whenever a request is made to /users/, the usersHandler function will be called, and so on.

Note the following points about the behavior of the router:

ImageThe default router directs any request that it does not have a handler for to /.

ImageThe route must match exactly; for example, a request to /users will go to /, as it is missing a trailing slash.

ImageThe router has no concern over the type of request. It simply passes a request that matches a route to the handler.

**Working with Handler Functions**

While the Go router maps routes to functions, it is the handler functions that define how a request is handled and the response that is returned to the client. Many programming languages and web frameworks follow the pattern of passing a request and response through functions before returning the response. Go is similar in this respect. Handler functions are responsible for these common tasks:

ImageReading or writing headers

ImageExamining the type of a request

ImageFetching data from a database

ImageParsing request data

ImageAuthentication

Handler functions have access to the Request and the Response, so a common pattern is to complete everything needed for the request before writing the response back to the client. Once the response is written, no further processing on the response can take place. In the following example, the response is sent using the Write method. On the next line, a header is set on the response to be written. As the response has already been written, this will have no effect, but the code will compile. The key point: Write the response last.

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p248pro01a)

func helloWorld(w http.ResponseWriter, r \*http.Request) {  
     w.Write([]byte("Hello World\n"))  
     // This has no effect, as the response is already written  
     w.Header().Set("X-My-Header", "I am setting a header!")  
}

**Handling 404s**

The behavior of the default Handler is to pass any request that does not have a handler function defined to /. Returning to the first example, if a request is made to a non-existent page, the handler function for / is called, and a 200 along with the “Hello World” response is returned.

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p248pro02a)

curl -is http://localhost:8000/asdfa  
HTTP/1.1 200 OK  
Date: Thu, 17 Nov 2016 09:07:51 GMT  
Content-Length: 12  
Content-Type: text/plain; charset=utf-8

As the route does not exist, a 404 Page Not Found should be returned. On the default route, a check can be added to return a 404 if the path is not / (see [Listing 18.2](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ex02)).

**LISTING 18.2** Adding a 404 Response

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p18ex02a)

 1:  package main  
 2:  
 3:  import (  
 4:     "net/http"  
 5:  )  
 6:  
 7:  func helloWorld(w http.ResponseWriter, r \*http.Request) {  
 8:     if r.URL.Path != "/" {  
 9:              http.NotFound(w, r)  
10:              return  
11:     }  
12:     w.Write([]byte("Hello World\n"))  
13:  }  
14:  
15:  func main() {  
16:      http.HandleFunc("/", helloWorld)  
17:      http.ListenAndServe(":8000", nil)  
18:  }

The modifications to the initial Hello World web server may be explained as follows:

ImageIn the helloWorld handler function, the path is checked to see if it is /.

ImageIf it is not, the NotFound method from the http package is called, passing the response and request. This writes a 404 response to the client.

ImageIf the path does match /, then the if statement is ignored and the Hello World response is sent.

TRY IT YOURSELF: Adding a 404 Response

In this example, you will understand how to add a 404 response.

**1.** Open the file hour18/example02.go in a text editor and try to understand what the example is doing. If you need to, refer to the previous bullet points and step through the code.

**2.** From the terminal, run the program with go run example02.go.

**3.** Using curl, make a request to a non-existent page. For example:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p249pro01a)

curl -is http://localhost:8000/asdfa

**4.** You should see that the response is now a 404:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p249pro02a)

HTTP/1.1 404 Not Found  
Content-Type: text/plain; charset=utf-8  
X-Content-Type-Options: nosniff  
Date: Thu, 17 Nov 2016 09:15:23 GMT  
Content-Length: 19  
  
404 page not found

**Setting a Header**

A common requirement when creating HTTP servers is to be able to set headers on a response. Go offers great support for creating, reading, updating, and deleting headers. In the following example, suppose that the 250server will send some JSON. By setting the Content-Type header, the server can inform the client that JSON data is being sent. Through the ResponseWriter, a handler function can add a header as follows:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p249pro03a)

w.Header().Set("Content-Type", "application/json; charset=utf-8")

Provided this is before the response is written to the client, the header will be added to the response. In [Listing 18.3](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ex03), the header is added before the JSON content is sent. Note that for simplicity the JSON is set as a string, but normally data would be read from somewhere and then encoded to JSON.

**LISTING 18.3** Adding a Header to a Response

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p18ex03a)

 1:  package main  
 2:  
 3:  import (  
 4:      "net/http"  
 5:  )  
 6:  
 7:  func helloWorld(w http.ResponseWriter, r \*http.Request) {  
 8:      if r.URL.Path != "/" {  
 9:              http.NotFound(w, r)  
10:              return  
11:      }  
12:      w.Header().Set("Content-Type", "application/json; charset=utf-8")  
13:      w.Write([]byte(`{"hello": "world"}`))  
14:  }  
15:  
16:  func main() {  
17:      http.HandleFunc("/", helloWorld)  
18:      http.ListenAndServe(":8000", nil)  
19:  }

TRY IT YOURSELF: Including an HTTP Header in a Response

In this example, you will understand how to add a HTTP header.

**1.** Open the file hour18/example03.go in a text editor and try to understand what the example is doing.

**2.** From the terminal, run the program with go run example03.go.

**3.** Using curl, make a request to the web server:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p250pro01a)

curl -is http://localhost:8000/

**4.** You should see that the response contains a header setting the content type to: application/json.

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p250pro02a)

HTTP/1.1 200 OK  
Content-Type: application/json; charset=utf-8  
Date: Thu, 17 Nov 2016 09:28:44 GMT  
Content-Length: 18  
  
{"hello": "world"}

**5.** Congratulations! You just served JSON using Go.

**Responding with Different Content Types**

HTTP servers typically respond to clients with multiple content types. Some content types in common usage include text/plain, text/html, application/json, and application/xml. If a server supports multiple content types, a client may request a content type using an Accept header. This means the same URL can serve a browser with HTML or an API client with JSON. With a small modification, the example that you have been working through can now respond with multiple content types by examining the Accept header sent by the client, as shown in [Listing 18.4](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ex04).

**LISTING 18.4** Responding with Different Content Types

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p18ex04a)

 1:  package main  
 2:  
 3:  import (  
 4:      "net/http"  
 5:  )  
 6:  
 7:  func helloWorld(w http.ResponseWriter, r \*http.Request) {  
 8:     if r.URL.Path != "/" {  
 9:              http.NotFound(w, r)  
10:              return  
11:      }  
12:      switch r.Header.Get("Accept") {  
13:      case "application/json":  
14:              w.Header().Set("Content-Type", "application/json; charset=utf-8")  
15:              w.Write([]byte(`{"message": "Hello World"}`))  
16:     case "application/xml":  
17:              w.Header().Set("Content-Type", "application/xml; charset=utf-8")  
18:              w.Write([]byte(`<?xml version="1.0" encoding="utf-  
    8"?><Message>Hello World</Message>`)  
19:     default:  
20:              w.Header().Set("Content-Type", "text/plain; charset=utf-8")  
21:              w.Write([]byte("Hello World\n"))  
22:      }  
23:  
24:  }  
25:  
26:  func main() {  
27:      http.HandleFunc("/", helloWorld)  
28:      http.ListenAndServe(":8000", nil)  
29:  }

The amendments to the example may be explained as follows:

ImageIn the helloWorld function, a switch statement is added that examines the Accept header from the client.

ImageDepending on the contents of the Accept header, a switch statement is used to set the response accordingly.

ImageIf no header is found, the server defaults to sending a plain text response.

TRY IT YOURSELF: Serving Different Content Types

In this example, you will understand how to serve different content types and how to request them using a client.

**1.** Open the file hour18/example04.go in a text editor and try to understand what the example is doing. If you need to, refer to the previous bullet points and step through the code.

**2.** From the terminal, run the program with go run example04.go.

**3.** Using curl, make a request to the web server that requests the content type application/json . Note that the -H option is used to set a header:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p252pro01a)

curl -si -H 'Accept: application/json' http://localhost:8000

**4.** You should see that the response is of content type application/json:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p252pro02a)

HTTP/1.1 200 OK  
Content-Type: application/json; charset=utf-8  
Date: Thu, 17 Nov 2016 09:28:44 GMT  
Content-Length: 18  
  
{"hello": "world"}

**5.** Using curl, make a second request to the web server that requests the content type application/xml:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p252pro03a)

curl -si -H 'Accept: application/xml' http://localhost:8000

**6.** You should see that the response is of content type application/xml:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p252pro04a)

HTTP/1.1 200 OK  
Content-Type: application/xml; charset=utf-8  
Date: Thu, 17 Nov 2016 09:45:24 GMT  
Content-Length: 68  
  
<?xml version="1.0" encoding="utf-8"?><Message>Hello World</Message>

**Responding to Different Types of Requests**

As well as being able to respond to requests for different content types, HTTP servers typically need to be able to respond to different types of requests. The types of requests that a client may make are defined in the HTTP specification and include GET, POST, PUT, and DELETE. To create a HTTP server in Go that responds to different types of requests, a similar technique to serve multiple content types may be used, as shown in [Listing 18.5](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ex05). In the handler function for a route, the request type can be checked and then a switch can determine how to handle the request.

**LISTING 18.5** Responding to Different Types of Requests

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p18ex05a)

 1:  package main  
 2:  
 3:  import (  
 4:      "net/http"  
 5:  )  
 6:  
 7:  func helloWorld(w http.ResponseWriter, r \*http.Request) {  
 8:      if r.URL.Path != "/" {  
 9:              http.NotFound(w, r)  
10:              return  
11:      }  
12:      switch r.Method {  
13:      case "GET":  
14:              w.Write([]byte("Received a GET request\n"))  
15:      case "POST":  
16:              w.Write([]byte("Received a POST request\n"))  
17:      default:  
18:              w.WriteHeader(http.StatusNotImplemented)  
19:              w.Write([]byte(http.StatusText(http.StatusNotImplemented)) + "\n")  
20:      }  
21:  
22:  }  
23:  
24:  func main() {  
25:      http.HandleFunc("/", helloWorld)  
26:      http.ListenAndServe(":8000", nil)  
27:  }

The amendments to the server can be explained as follows:

ImageInstead of using content type to switch the response, the server uses the request method.

ImageThe switch statement sends a response depending on the type of request.

ImageIn this example, a plain text response is sent to indicate the type of request.

ImageIf the method is not a GET or a POST, it falls through to the default. This sends a 501 Not Implemented HTTP response. The 501 code means the server does not understand or does not support the HTTP method sent by the client.

Running the server shows that both GET and POST requests may now be made. To change the request type using curl, the -X option is used.

TRY IT YOURSELF: Understanding Different Types of Requests

In this example, you will understand how to respond to different types of requests, such as GET and POST.

**1.** Open the file hour18/example05.go in a text editor and try to understand what the example is doing. If you need to, refer to the explanation that follows [Listing 18.5](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ex05).

**2.** From the terminal, run the program with go run example05.go.

**3.** Using curl, make a request to the web server using a GET request. Note that the -X option is used to set the type of request:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p254pro01a)

curl -si -X GET http://localhost:8000

**4.** You should see the server responding that it has received a GET request:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p254pro02a)

HTTP/1.1 200 OK  
Date: Thu, 17 Nov 2016 10:02:49 GMT  
Content-Length: 23  
Content-Type: text/plain; charset=utf-8  
  
Received a GET request

**5.** Using curl, make a second request to the web server, this time making a POST request:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p254pro03a)

curl -si -X POST http://localhost:8000

**6.** You should see the server responding that it has received a POST request:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p254pro04a)

HTTP/1.1 200 OK  
Date: Thu, 17 Nov 2016 10:03:27 GMT  
Content-Length: 24  
Content-Type: text/plain; charset=utf-8  
  
Received a POST request

**Receiving Data from GET and POST Requests**

An HTTP client can send data to an HTTP server along with an HTTP request. Typical examples of this include:

ImageSubmitting a form

ImageSetting options on data to be returned

ImageManaging data through an API interface

Getting data from a client request is simple in Go, but depending on the type of request it is accessed in different ways. For a GET, request data is usually set through a query string. An example of sending data through a GET request is making a search on Google. Here, the URL includes a search term as a query string:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p255pro01a)

https://www.google.com/?q=golang

A web server may then read in the query string data, using it to do something like fetch some data from a database before returning it to the client. In Go, the query string parameters for a request are available as a map of strings, and these can be iterated over using a range clause.

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p255pro02a)

func queryParams(w http.ResponseWriter, r \*http.Request) {  
    for k, v := range r.URL.Query() {  
        fmt.Printf("%s: %s\n", k, v)  
    }  
}

For a POST request, data is usually sent as the body of a request. This data may be read and used as follows:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p255pro03a)

func queryParams(w http.ResponseWriter, r \*http.Request) {  
    reqBody, err := ioutil.ReadAll(r.Body)  
    if err != nil {  
        log.Fatal(err)  
    }  
    fmt.Printf("%s", reqBody)  
}

Warning: Do Not Trust User Input

An aside on security for a moment: Data received on a server should be considered untrusted. An attacker may send a request that attempts to steal information, gain access to a server, or delete a database. All data coming into a server should be considered unsafe and should be filtered before use. For the purposes of these examples, data is used unfiltered, but when writing code for production use, ensure that the incoming data is sanitized before use.

A full code example can now be created to demonstrate handling data from different requests, which appears in [Listing 18.6](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ex06). This server builds on the previous example to show the data that is sent to the server. Running this example shows that data can be received for different types of requests. Of course, the server will probably want to do something more interesting with the data other than return it to the client.

**LISTING 18.6** Handling Data from Different Requests

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p18ex06a)

 1:  package main  
 2:  
 3:  import (  
 4:      "fmt"  
 5:      "io/ioutil"  
 6:      "log"  
 7:      "net/http"  
 8:  )  
 9:  
10:  func helloWorld(w http.ResponseWriter, r \*http.Request) {  
11:      if r.URL.Path != "/" {  
12:              http.NotFound(w, r)  
13:              return  
14:      }  
15:      switch r.Method {  
16:      case "GET":  
17:              for k, v := range r.URL.Query() {  
18:                     fmt.Printf("%s: %s\n", k, v)  
19:              }  
20:              w.Write([]byte("Received a GET request\n"))  
21:     case "POST":  
22:              reqBody, err := ioutil.ReadAll(r.Body)  
23:              if err != nil {  
24:                       log.Fatal(err)  
25:              }  
26:  
27:              fmt.Printf("%s\n", reqBody)  
28:              w.Write([]byte("Received a POST request\n"))  
29:     default:  
30:              w.WriteHeader(http.StatusNotImplemented)  
31:              w.Write([]byte(http.StatusText(http.StatusNotImplemented)))  
32:     }  
33:  
34:  }  
35:  
36:  func main() {  
37:      http.HandleFunc("/", helloWorld)  
38:      http.ListenAndServe(":8000", nil)  
39:  }

Try It Yourself: Receiving Data from GET and POST Requests

In this example, you will understand how to receive data from GET and POST requests.

**1.** Open the file hour18/example06.go in a text editor and try to understand what the example is doing.

**2.** From the terminal, run the program with go run example06.go.

**3.** Using curl, make a request to the web server using a GET request that includes some query parameters:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p257pro01a)

curl -si "http://localhost:8000/?foo=1&bar=2"

**4.** In the terminal that is running the server, you should see the data has been received:

foo: [1]  
  
bar: [2]

**5.** Using curl, make a second request to the web server, this time making a POST request:

[Click here to view code image](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18_images.xhtml#p257pro02a)

curl -si -X POST -d "some data to send" http://localhost:8000/

**6.** In the terminal that is running the server, you should see the data has been received:

some data to send

**Summary**

This hour introduced you to creating HTTP servers with Go. You were introduced to how routing works with the HTTP package and understood how handler functions may be used to handle requests. You learned how to set a header on an HTTP response and then progressed to being able to respond to different types of requests. Finally, you learned how to receive data from HTTP client requests.

**Q&A**

[**Q**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ans1a)**. Is it possible to set variables in the routing pattern? I want to set something like /products/:id, where :id is a variable.**

[**A**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ans1)**.** The http package by default uses ServeMux to handle routing, and neither variables nor regular expressions are supported. Some popular community-created routers offer variables and other features like request and content types. Generally, they integrate with the http package.

[**Q**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ans2a)**. I used framework libraries in other languages to create servers. Is anything similar available in Go?**

[**A**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ans2)**.** Yes. There are framework libraries available in Go. For many cases, however, the http package provides everything needed.

[**Q**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ans3a)**. How do I create a HTTPS server?**

[**A**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18ans3)**.** The http package supports creating a server served over HTTPS (TLS) through the ListenAndServeTLS method. This works in the same way as ListenAndServe, but expects certificate and key files to be passed to it.

**Workshop**

The workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the “Answers” section that follows.

**Quiz**

[**1.**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18que1a) What is the difference between an HTTP GET and a POST request?

[**2.**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18que2a) In a handler function, what does w.Write do in terms of processing the response where w is a ResponseWriter?

[**3.**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18que3a) Should you trust data submitted to an HTTP server?

**Answers**

[**1.**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18que1) A GET request requests data from a specified resource. A POST request submits data to a specified resource. A GET request may set data through query string parameters. A POST request sends data to the server as the message body.

[**2.**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18que2) Calling Write on a ResponseWriter causes the response to be sent to the client. This includes headers and the body content. Once sent, it is not possible to modify the response.

[**3.**](https://learning.oreilly.com/library/view/go-in-24/9780134771922/ch18.xhtml#ch18que3) No. You should never trust data submitted by a client to a server. Data should be sanitized before use.

**Exercises**

**1.** Modify example04.go to be able to respond to a request for HTML with a simple HTML document. The HTML content type is "text/html; charset=utf-8".

**2.** Modify example05.go to allow the server to support DELETE requests.

**3.** Read the documentation for the http package and try to understand some of the modifications you can make to a request and a response when writing a server.

* [Support](https://learning.oreilly.com/public/support/)
* [Sign Out](https://learning.oreilly.com/accounts/logout/)

© 2021 [O'Reilly Media, Inc](https://learning.oreilly.com/). [Terms of Service](https://www.oreilly.com/terms/) / [Privacy Policy](https://learning.oreilly.com/privacy)