

George Ornbo

Sams **Teach Yourself**

Go

Next Generation Systems
Programming with **Golang**

in **24**
Hours

 Pearson

George Ornbo

Sams **Teach Yourself**

Go

Next Generation Systems
Programming with **Golang**

in **24**
Hours

SAMS

800 East 96th Street, Indianapolis, Indiana 46240 USA

Sams Teach Yourself Go in 24 Hours

Next Generation Systems Programming with Golang

Copyright © 2018 by Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33803-8

ISBN-10: 0-672-33803-3

Library of Congress Control Number: 2017955706

First Printing December 2017

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark. Go™ programming language is a trademark of Google, Inc.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided “as is” without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screenshots may be viewed in full within the software version specified.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. Screenshots and icons reprinted with permission from the Microsoft Corporation. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Editor-in-Chief

Mark Taub

Executive Editor

Laura Lewin

Development Editor

Sheri Replin

Managing Editor

Sandra Schroeder

Project Editor

Prathiba.R/
Pearson CSC

Copy Editor

Pearson CSC

Indexer

Pearson CSC

Proofreader

Pearson CSC

Technical Editors

Bala Natarajan
Yoshiki Shibata

Editorial Assistant

Courtney Martin

Cover Designer

Chuti Prasertsith

Cover Image

Ryan McVay/
Getty Images

Compositor

Pearson CSC

Contents at a Glance

Introduction	1
HOURL 1 Getting Started	3
HOURL 2 Understanding Types	13
HOURL 3 Understanding Variables	27
HOURL 4 Using Functions in Go	41
HOURL 5 Using Control Flow	53
HOURL 6 Working with Arrays, Slices, and Maps	71
HOURL 7 Using Structs and Pointers	83
HOURL 8 Creating Methods and Interfaces	101
HOURL 9 Working with Strings	115
HOURL 10 Handling Errors	129
HOURL 11 Working with Goroutines	139
HOURL 12 Introducing Channels	155
HOURL 13 Using Packages for Code Reuse	173
HOURL 14 Naming Conventions in Go	185
HOURL 15 Testing and Performance	199
HOURL 16 Debugging	213
HOURL 17 Using Command-Line Programs	227
HOURL 18 Creating HTTP Servers	243
HOURL 19 Creating HTTP Clients with Go	259
HOURL 20 Working with JSON	273
HOURL 21 Working with Files	289
HOURL 22 Introducing Regular Expressions	305
HOURL 23 Programming Time in Go	317
HOURL 24 Deploying Go Code	329
Index	341

APPENDIX

Online ancillaries: Bonus Hours 25 and 26

Table of Contents

Introduction	1
HOURL 1: Getting Started	3
Introducing Go	3
Installing Go	4
Setting Up Your Environment	8
Writing Your First Go Program: Hello World	8
Summary	10
Q&A	10
Workshop	11
Exercises	11
HOURL 2: Understanding Types	13
What Is a Data Type?	13
Differentiating Between Static and Dynamic Typing	13
Implementing Booleans	17
Understanding Numeric Types	18
Checking the Type of a Variable	22
Converting Between Types	23
Summary	24
Q&A	25
Workshop	25
Exercises	26
HOURL 3: Understanding Variables	27
What Is a Variable?	27
Declaring Shorthand Variables	29
Understanding Variables and Zero Values	30
Writing a Short Variable Declaration	32
Styles of Variable Declaration	32

Using Variable Scope	33
Using Pointers	35
Declaring Variables with Constants	38
Summary	39
Q&A	39
Workshop	39
Exercises	40
HOURL 4: Using Functions in Go	41
What Is a Function?	41
Defining Variadic Functions	45
Using Named Return Values	46
Using Recursive Functions	48
Passing Functions as Values	49
Summary	51
Q&A	51
Workshop	51
Exercises	52
HOURL 5: Using Control Flow	53
Using the if Statement	53
Using the else Statement	55
Using the else if Statement	57
Using Comparison Operators	58
Using Arithmetic Operators	59
Using Logical Operators	59
Using the switch Statement	60
Looping with the for Statement	62
Using the defer Statement	66
Summary	68
Q&A	68
Workshop	69
Exercises	69

HOURL 6: Working with Arrays, Slices, and Maps	71
Working with Arrays	71
Working with Slices	73
Working with Maps	78
Summary	80
Q&A	80
Workshop	81
Exercises	81
HOURL 7: Using Structs and Pointers	83
What Is a Struct?	83
Initializing a Struct	85
Nesting Structs	89
Creating Default Values for Structs	91
Comparing Structs	93
Understanding Public and Private Values	95
Differentiating Between Pointer and Value References	96
Summary	99
Q&A	99
Workshop	100
Exercises	100
HOURL 8: Creating Methods and Interfaces	101
Using Methods	101
Creating Method Sets	103
Working with Methods and Pointers	105
Using Interfaces	109
Summary	113
Q&A	113
Workshop	114
Exercises	114
HOURL 9: Working with Strings	115
Creating String Literals	115
Understanding Rune Literals	116
Concatenating Strings	118

Summary	126
Q&A	126
Workshop	127
Exercises	127
HOURL 10: Handling Errors	129
Handling Errors and Idiomatic Go	130
Understanding the Error Type	132
Creating Errors	132
Formatting Errors	133
Returning an Error from a Function	134
Errors and Usability	135
Don't panic	136
Summary	137
Q&A	137
Workshop	138
Exercises	138
HOURL 11: Working with Goroutines	139
Understanding Concurrency	139
Concurrency Versus Parallelism	141
Understanding Concurrency Through a Web Browser	142
Blocking and Non-Blocking Code	144
Handling Concurrent Operations with Goroutines	145
Using Goroutines to Manage Latency	147
Defining Goroutines	152
Summary	153
Q&A	153
Workshop	153
Exercises	154
HOURL 12: Introducing Channels	155
Using Channels	155
Using Buffered Channels	159
Blocking and Flow Control	161
Using Channels as Function Arguments	164

Employing the select Statement	164
Quitting Channels	167
Summary	170
Q&A	170
Workshop	171
Exercises	171
HOURL 13: Using Packages for Code Reuse	173
Importing Packages	173
Understanding Package Usage	175
Using Third-Party Packages	175
Installing a Third-Party Package	176
Managing Third-Party Dependencies	178
Summary	182
Q&A	182
Workshop	183
Exercises	184
HOURL 14: Naming Conventions in Go	185
Formatting Code in Go	185
Using gofmt	186
Configuring Text Editors	188
Naming Conventions	188
Using golint	190
Using godoc	192
Automating Workflow	196
Summary	197
Q&A	197
Workshop	198
Exercises	198
HOURL 15: Testing and Performance	199
Testing: The Most Important Aspect of Software Development	199
testing Package	201
Running Table Tests	204
Benchmarking in Go	207

Providing Test Coverage	209
Summary	210
Q&A	210
Workshop	211
Exercises	211
HOURL 16: Debugging	213
Logging	213
Printing Data	218
Using the <code>fmt</code> Package	219
Using Delve	221
Using <code>gdb</code>	224
Summary	225
Q&A	225
Workshop	226
Exercises	226
HOURL 17: Using Command-Line Programs	227
Operating with Inputs and Outputs	228
Accessing Raw Command-Line Arguments	229
Parsing Command-Line Flags	230
Working with Types	232
Customizing Help Text	234
Creating Subcommands	235
POSIX Compliance	239
Installing and Sharing Command-Line Programs	240
Summary	241
Q&A	241
Workshop	242
Exercises	242
HOURL 18: Creating HTTP Servers	243
Announcing Your Presence with the “Hello World” Web Server	243
Examining Requests and Responses	245
Working with Handler Functions	247
Handling 404s	248

Setting a Header	249
Responding to Different Types of Requests	253
Receiving Data from GET and POST Requests	255
Summary	257
Q&A	257
Workshop	258
Exercises	258
HOURL 19: Creating HTTP Clients with Go	259
Understanding HTTP	259
Making a GET Request	261
Making a POST Request	262
Gaining Further Control over HTTP Requests	264
Debugging HTTP Requests	265
Dealing with Timeouts	269
Summary	270
Q&A	271
Workshop	271
Exercises	272
HOURL 20: Working with JSON	273
Introducing JSON	273
Using JSON APIs	275
Using JSON with Go	275
Decoding JSON	281
Mapping Data Types	283
Working with JSON Received over HTTP	285
Summary	286
Q&A	286
Workshop	287
Exercises	287
HOURL 21: Working with Files	289
Importance of Files	289
Reading and Writing Files with the io/ioutil Package	290
Writing to a File	294

Listing a Directory	295
Copying a File	296
Deleting Files	298
Using Files to Manage Configuration	299
Summary	302
Q&A	303
Workshop	303
Exercises	304
HOURL 22: Introducing Regular Expressions	305
Defining Regular Expressions	305
Getting Familiar with Regular Expression Syntax	307
Using Regular Expressions for Validation	308
Using Regular Expressions to Transform Data	309
Parsing Data with Regular Expressions	311
Summary	315
Q&A	315
Workshop	316
Exercises	316
HOURL 23: Programming Time in Go	317
Programming the Element of Time	317
Putting Your Program to Sleep	319
Setting a Timeout	320
Using a Ticker	321
Representing Time in a String Format	322
Working with Time Structs	323
Adding and Subtracting Time	324
Comparing Different Time Structs	324
Summary	326
Q&A	326
Workshop	326
Exercises	327

HOURL 24: Deploying Go Code	329
Understanding Targets	329
Reducing the Size of Binaries	332
Using Docker	334
Downloading Binary Files	336
Using <code>go get</code>	338
Releasing Code with Package Managers	339
Summary	339
Q&A	339
Workshop	340
Exercises	340
Index	341

APPENDIX

Online ancillaries: Bonus Hours 25 and 26

Follow these steps to access the online chapters:

1. Register your book by going to www.informit.com/register, and log in or create a new account.
2. On the Register a Product page, enter this book's ISBN (9780672338038), and click Submit.
3. Answer the challenge question as proof of book ownership.
4. On the Registered Products tab of your account page, click on the Access Bonus Content link to go to the page where your downloadable content is available.

About the Author

George Ornbo is a software engineer, blogger, and author with 14 years of experience delivering software to startups and enterprise clients. He has experience with a broad range of programming languages, UNIX, and the underlying protocols of the web. He is currently working at a Blockchain startup in London.

Dedication

For Bea and Fin. I won a Golden Ticket with you two!

Acknowledgments

Thanks to Laura Lewin and the team at Pearson for giving me the chance to write a second book in the Sams Teach Yourself in 24 Hours Series. Thanks also to Sheri Replin for your amazing work!

Thanks to Bala Natarajan and Yoshiki Shibata for being the technical reviewers of this book. You suggested many excellent improvements throughout the reviews. Any mistakes left in the book are, of course, my own.

Thanks to Robert Griesemer, Rob Pike, and Ken Thompson for thinking again about language design.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@sampublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Register your copy of *Sams Teach Yourself Go in 24 Hours* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account*. Enter the product ISBN, 9780672338038, and click Submit. Once the process is complete, you will find any available bonus content under Registered Products.

* Be sure to check the box that you would like to hear from us in order to receive exclusive discounts on future editions of this product.

HOUR 18

Creating HTTP Servers

What You'll Learn in This Hour:

- ▶ Announcing the presence of your web server
- ▶ Examining requests and responses
- ▶ Working with handler functions
- ▶ Handling 404s
- ▶ Setting a header
- ▶ Responding with different content types
- ▶ Responding to different types of requests
- ▶ Receiving data from GET and Post requests

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.

LISTING 18.1 Basic HTTP Server in Go

```
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.

- ▶ The `net/http` package is imported.
- ▶ Within 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.
- ▶ The `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.
- ▶ The `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.

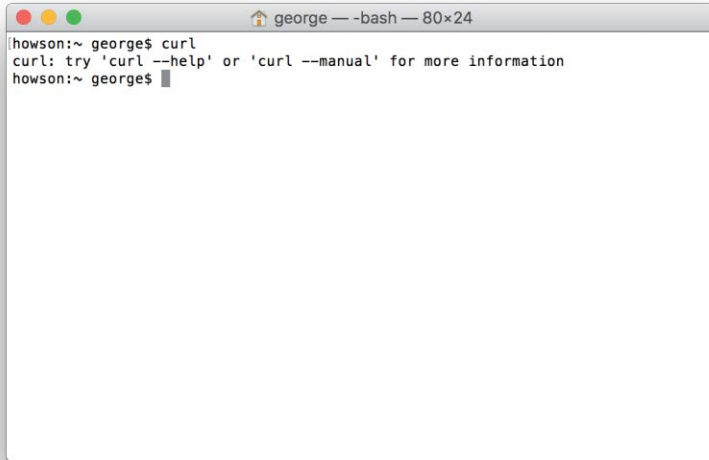
Examining Requests and Responses

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.

A terminal window titled 'george — -bash — 80x24' with a home icon. The prompt is 'howson:~ george\$'. The user has entered 'curl', and the terminal displays the help text: 'curl: try 'curl --help' or 'curl --manual' for more information'. The prompt returns to 'howson:~ george\$' with a cursor.

```
howson:~ george$ curl
curl: try 'curl --help' or 'curl --manual' for more information
howson:~ george$
```

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.

```
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:

- ▶ The response uses the HTTP 1.1 protocol, and a 200 response was received.
- ▶ The `Date` header details when the response was sent.
- ▶ The `Content-Length` header details the length of the response. In this case, it is 12 bytes.
- ▶ The `Content-Type` header details the type of content and the encoding used. In this case, the response is `text/plain` encoded using `utf-8`.
- ▶ Finally, 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.

```
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:

- ▶ The default router directs any request that it does not have a handler for to `/`.
- ▶ The route must match exactly; for example, a request to `/users` will go to `/`, as it is missing a trailing slash.
- ▶ The 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:

- ▶ Reading or writing headers
- ▶ Examining the type of a request
- ▶ Fetching data from a database
- ▶ Parsing request data
- ▶ Authentication

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.

```
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.

```
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).

LISTING 18.2 Adding a 404 Response

```
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:

- ▶ In the `helloWorld` handler function, the path is checked to see if it is `/`.
- ▶ If 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.
- ▶ If 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:

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

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

```
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 server 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:

```
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, 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

```
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:

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

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

```
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.

LISTING 18.4 Responding with Different Content Types

```

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-
19:8"?><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:

- ▶ In the `helloWorld` function, a `switch` statement is added that examines the `Accept` header from the client.
- ▶ Depending on the contents of the `Accept` header, a `switch` statement is used to set the response accordingly.
- ▶ If 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:
4. You should see that the response is of content type `application/json`:

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

```
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. 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

```

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:

- ▶ Instead of using content type to switch the response, the server uses the request method.
- ▶ The `switch` statement sends a response depending on the type of request.

- ▶ In this example, a plain text response is sent to indicate the type of request.
- ▶ If 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.
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:

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

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

```
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:

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

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

```
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:

- ▶ Submitting a form
- ▶ Setting options on data to be returned
- ▶ Managing 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:

```
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.

```
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:

```
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. 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

```
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:

```
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:

```
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.** Is it possible to set variables in the routing pattern? I want to set something like `/products/:id`, where `:id` is a variable.
- A.** 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.** I used framework libraries in other languages to create servers. Is anything similar available in Go?
- A.** Yes. There are framework libraries available in Go. For many cases, however, the `http` package provides everything needed.

Q. How do I create a HTTPS server?

- A.** 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. What is the difference between an HTTP `GET` and a `POST` request?
2. In a handler function, what does `w.Write` do in terms of processing the response where `w` is a `ResponseWriter`?
3. Should you trust data submitted to an HTTP server?

Answers

1. 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. 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. 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.