**Go Web applications**

**Introduction**

Go is a battery included programming language and has a webserver already built in. The net/http package from the standard library contains all functionalities about the HTTP protocol. This includes (among many other things) an HTTP client and an HTTP server. In this example you will figure out how simple it is, to create a webserver that you can view in your browser.

## Registering a Request Handler

First, create a Handler which receives all incomming HTTP connections from browsers, HTTP clients or API requests. A handler in Go is a function with this signature:

**func** (w http.ResponseWriter, r **\***http.Request)

The function receives two parameters:

1. An http.ResponseWriter which is where you write your text/html response to.
2. An http.Request which contains all information about this HTTP request including things like the URL or header fields.

Registering a request handler to the default HTTP Server is as simple as this:

http.HandleFunc("/", **func** (w http.ResponseWriter, r **\***http.Request) {

fmt.Fprintf(w, "Hello, you've requested: %s\n", r.URL.Path)

})

## Listen for HTTP Connections

The request handler alone can not accept any HTTP connections from the outside. An HTTP server has to listen on a port to pass connections on to the request handler. Because port 80 is in most cases the default port for HTTP traffic, this server will also listen on it.

The following code will start Go’s default HTTP server and listen for connections on port 80. You can navigate your browser to http://localhost/ and see your server handing your request.

http.ListenAndServe(":80", **nil**)

## The Code (for copy/paste)

This is the complete code that you can use to try out the things you’ve learned in this example.

**package** main

**import** (

"fmt"

"net/http"

)

**func** main() {

http.HandleFunc("/", **func**(w http.ResponseWriter, r **\***http.Request) {

fmt.Fprintf(w, "Hello, you've requested: %s\n", r.URL.Path)

})

http.ListenAndServe(":80", **nil**)

}

# HTTP Server

## Introduction

In this example you will learn how to create a basic HTTP server in Go. First, let’s talk about what our HTTP server should be capable of. A basic HTTP server has a few key jobs to take care of.

* Process dynamic requests: Process incoming requests from users who browse the website, log into their accounts or post images.
* Serve static assets: Serve JavaScript, CSS and images to browsers to create a dynamic experience for the user.
* Accept connections: The HTTP Server must listen on a specific port to be able to accept connections from the internet.

## Process dynamic requests

The net/http package contains all utilities needed to accept requests and handle them dynamically. We can register a new handler with the http.HandleFunc function. It’s first parameter takes a path to match and a function to execute as a second. In this example: When someone browses your websites (http://example.com/), he or she will be greeted with a nice message.

http.HandleFunc("/", **func** (w http.ResponseWriter, r **\***http.Request) {

fmt.Fprint(w, "Welcome to my website!")

})

For the dynamic aspect, the http.Request contains all information about the request and it’s parameters. You can read GET parameters with r.URL.Query().Get("token") or POST parameters (fields from an HTML form) with r.FormValue("email").

## Serving static assets

To serve static assets like JavaScript, CSS and images, we use the inbuilt http.FileServer and point it to a url path. For the file server to work properly it needs to know, where to serve files from. We can do this like so:

fs **:=** http.FileServer(http.Dir("static/"))

Once our file server is in place, we just need to point a url path at it, just like we did with the dynamic requests. One thing to note: In order to serve files correctly, we need to strip away a part of the url path. Usually this is the name of the directory our files live in.

http.Handle("/static/", http.StripPrefix("/static/", fs))

## Accept connections

The last thing to finish off our basic HTTP server is, to listen on a port to accept connections from the internet. As you can guess, Go has also an inbuilt HTTP server, we can start faily quickly. Once started, you can view your HTTP server in your [browser.](http://localhost/)

http.ListenAndServe(":80", **nil**)

## The Code (for copy/paste)

This is the complete code that you can use to try out the things you’ve learned in this example.

**package** main

**import** (

"fmt"

"net/http"

)

**func** main() {

http.HandleFunc("/", **func** (w http.ResponseWriter, r **\***http.Request) {

fmt.Fprintf(w, "Welcome to my website!")

})

fs **:=** http.FileServer(http.Dir("static/"))

http.Handle("/static/", http.StripPrefix("/static/", fs))

http.ListenAndServe(":80", **nil**)

}

# Routing (using gorilla/mux)

## Introduction

Go’s net/http package provides a lot of functionalities for the HTTP protocol. One thing it doesn’t do very well is complex request routing like segmenting a request url into single parameters. Fortunately there is a very popular package for this, which is well known for the good code quality in the Go community. In this example you will see how to use the gorilla/mux package to create routes with named parameters, GET/POST handlers and domain restrictions.

## Installing the gorilla/mux package

gorilla/mux is a package which adapts to Go’s default HTTP router. It comes with a lot of features to increase the productivity when writing web applications. It is also compliant to Go’s default request handler signature func (w http.ResponseWriter, r \*http.Request), so the package can be mixed and machted with other HTTP libraries like middleware or exisiting applications. Use the go get command to install the package from GitHub like so:

go get -u github.com/gorilla/mux

## Creating a new Router

First create a new request router. The router is the main router for your web application and will later be passed as parameter to the server. It will receive all HTTP connections and pass it on to the request handlers you will register on it. You can create a new router like so:

r **:=** mux.NewRouter()

## Registering a Request Handler

Once you have a new router you can register request handlers like usual. The only difference is, that instead of calling http.HandleFunc(...), you call HandleFunc on your router like this: r.HandleFunc(...).

## URL Parameters

The biggest strength of the gorilla/mux Router is the ability to extract segments from the request URL. As an example, this is a URL in your application:

/books/go-programming-blueprint/page/10

This URL has two dynamic segments:

1. Book title slug (go-programming-blueprint)
2. Page (10)

To have a request handler match the URL mentioned above you replace the dynamic segments of with placeholders in your URL pattern like so:

r.HandleFunc("/books/{title}/page/{page}", **func**(w http.ResponseWriter, r **\***http.Request) {

*// get the book*

*// navigate to the page*

})

The last thing is to get the data from these segments. The package comes with the function mux.Vars(r) which takes the http.Request as parameter and returns a map of the segments.

**func**(w http.ResponseWriter, r **\***http.Request) {

vars **:=** mux.Vars(r)

vars["title"] *// the book title slug*

vars["page"] *// the page*

}

## Setting the HTTP server’s router

Ever wondered what the nil in http.ListenAndServe(":80", nil) ment? It is the parameter for the main router of the HTTP server. By default it’s nil, which means to use the default router of the net/http package. To make use of your own router, replace the nil with the variable of your router r.

http.ListenAndServe(":80", r)

## The Code (for copy/paste)

This is the complete code that you can use to try out the things you’ve learned in this example.

**package** main

**import** (

"fmt"

"net/http"

"github.com/gorilla/mux"

)

**func** main() {

r **:=** mux.NewRouter()

r.HandleFunc("/books/{title}/page/{page}", **func**(w http.ResponseWriter, r **\***http.Request) {

vars **:=** mux.Vars(r)

title **:=** vars["title"]

page **:=** vars["page"]

fmt.Fprintf(w, "You've requested the book: %s on page %s\n", title, page)

})

http.ListenAndServe(":80", r)

}

## Features of the gorilla/mux Router

### Methods

Restrict the request handler to specific HTTP methods.

r.HandleFunc("/books/{title}", CreateBook).Methods("POST")

r.HandleFunc("/books/{title}", ReadBook).Methods("GET")

r.HandleFunc("/books/{title}", UpdateBook).Methods("PUT")

r.HandleFunc("/books/{title}", DeleteBook).Methods("DELETE")

### Hostnames & Subdomains

Restrict the request handler to specific hostnames or subdomains.

r.HandleFunc("/books/{title}", BookHandler).Host("www.mybookstore.com")

### Schemes

Restrict the request handler to http/https.

r.HandleFunc("/secure", SecureHandler).Schemes("https")

r.HandleFunc("/insecure", InsecureHandler).Schemes("http")

### Path Prefixes & Subrouters

Restrict the request handler to specific path prefixes.

bookrouter **:=** r.PathPrefix("/books").Subrouter()

bookrouter.HandleFunc("/", AllBooks)

bookrouter.HandleFunc("/{title}", GetBook)

# Templates

## Introduction

Go’s html/template package provides a rich templating language for HTML templates. It is mostly used in web applications to display data in a structured way in a client’s browser. One great benefit of Go’s templating language is the automatic escaping of data. There is no need to worry about about XSS attacks as Go parses the HTML template and escapes all inputs before displaying it to the browser.

## First Template

Writing a template in Go is very simple. This example shows a TODO list, written as an unordered list (ul) in HTML. When rendering templates, the data passed in can be any kind of Go’s data structures. It may be a simple string or a number, it can even be nested data structure as in the example below. To access the data in a template the top most variable is access by {{.}}. The dot inside the curly braces is called the pipeline and the root element of the data.

data **:=** TodoPageData{

PageTitle: "My TODO list",

Todos: []Todo{

{Title: "Task 1", Done: **false**},

{Title: "Task 2", Done: **true**},

{Title: "Task 3", Done: **true**},

},

}

<h1>{{.PageTitle}}<h1>

<ul>

{{range .Todos}}

{{if .Done}}

<li class**=**"done">{{.Title}}</li>

{{else}}

<li>{{.Title}}</li>

{{end}}

{{end}}

</ul>

## Control Structures

The templating language contains a rich set of control structures to render your HTML. Here you will get an overview of the most commonly used ones. To get a detailed list of all possible structures visit: [text/template](https://golang.org/pkg/text/template/#hdr-Actions)

| **Control Structure** | **Definition** |
| --- | --- |
| {{/\* a comment \*/}} | Defines a comment |
| {{.}} | Renders the root element |
| {{.Title}} | Renders the “Title”-field in a nested element |
| {{if .Done}} {{else}} {{end}} | Defines an if-Statement |
| {{range .Todos}} {{.}} {{end}} | Loops over all “Todos” and renders each using {{.}} |
| {{block "content" .}} {{end}} | Defines a block with the name “content” |

## Parsing Templates from Files

Template can either be parsed from a string or a file on disk. As it is usually the case, that templates are pares from disk, this example shows how to do so. In this example there is a template file in the same directory as the Go program called layout.html.

tmpl, err **:=** template.ParseFiles("layout.html")

*// or*

tmpl **:=** template.Must(template.ParseFiles("layout.html"))

## Execute a Template in a Request Handler

Once the template is parsed from disk it’s ready to be used in the request handler. The Execute function accepts an io.Writer for writing out the template and an interface{} to pass data into the template. When the function is called on an http.ResponseWriter the Content-Type is header is automatically set in the HTTP response to Content-Type: text/html; charset=utf-8.

**func**(w http.ResponseWriter, r **\***http.Request) {

tmpl.Execute(w, "data goes here")

}

## The Code (for copy/paste)

This is the complete code that you can use to try out the things you’ve learned in this example.

**package** main

**import** (

"html/template"

"net/http"

)

**type** Todo **struct** {

Title **string**

Done **bool**

}

**type** TodoPageData **struct** {

PageTitle **string**

Todos []Todo

}

**func** main() {

tmpl **:=** template.Must(template.ParseFiles("layout.html"))

http.HandleFunc("/", **func**(w http.ResponseWriter, r **\***http.Request) {

data **:=** TodoPageData{

PageTitle: "My TODO list",

Todos: []Todo{

{Title: "Task 1", Done: **false**},

{Title: "Task 2", Done: **true**},

{Title: "Task 3", Done: **true**},

},

}

tmpl.Execute(w, data)

})

http.ListenAndServe(":80", **nil**)

}

<h1>{{.PageTitle}}<h1>

<ul>

{{range .Todos}}

{{if .Done}}

<li class**=**"done">{{.Title}}</li>

{{else}}

<li>{{.Title}}</li>

{{end}}

{{end}}

</ul>

# Assets and Files

This example will show how to serve static files like CSSs, JavaScripts or images from a specific directory.

*// static-files.go*

**package** main

**import** "net/http"

**func** main() {

fs **:=** http.FileServer(http.Dir("assets/"))

http.Handle("/static/", http.StripPrefix("/static/", fs))

http.ListenAndServe(":8080", **nil**)

}

$ tree assets/

assets/

└── css

└── styles.css

$ go run static-files.go

$ curl -s http://localhost:8080/static/css/styles.css

body {

background-color: black;

}

# Forms

This example will show how to simulate a contact form and parse the message into a struct.

*// forms.go*

**package** main

**import** (

"html/template"

"net/http"

)

**type** ContactDetails **struct** {

Email **string**

Subject **string**

Message **string**

}

**func** main() {

tmpl **:=** template.Must(template.ParseFiles("forms.html"))

http.HandleFunc("/", **func**(w http.ResponseWriter, r **\***http.Request) {

**if** r.Method **!=** http.MethodPost {

tmpl.Execute(w, **nil**)

**return**

}

details **:=** ContactDetails{

Email: r.FormValue("email"),

Subject: r.FormValue("subject"),

Message: r.FormValue("message"),

}

*// do something with details*

\_ = details

tmpl.Execute(w, **struct**{ Success **bool** }{**true**})

})

http.ListenAndServe(":8080", **nil**)

}

*<!-- forms.html -->*

{{if .Success}}

<h1>Thanks for your message!</h1>

{{else}}

<h1>Contact</h1>

<form method**=**"POST">

<label>Email:</label><br />

<input type**=**"text" name**=**"email"><br />

<label>Subject:</label><br />

<input type**=**"text" name**=**"subject"><br />

<label>Message:</label><br />

<textarea name**=**"message"></textarea><br />

<input type**=**"submit">

</form>

{{end}}

$ go run forms.go

# Contact

Top of Form

Email:  
  
Subject:  
  
Message:

# Middleware (Basic)

This example will show how to create basic logging middleware in Go.

A middleware simply takes a http.HandlerFunc as one of its parameters, wraps it and returns a new http.HandlerFunc for the server to call.

*// basic-middleware.go*

**package** main

**import** (

"fmt"

"log"

"net/http"

)

**func** logging(f http.HandlerFunc) http.HandlerFunc {

**return** **func**(w http.ResponseWriter, r **\***http.Request) {

log.Println(r.URL.Path)

f(w, r)

}

}

**func** foo(w http.ResponseWriter, r **\***http.Request) {

fmt.Fprintln(w, "foo")

}

**func** bar(w http.ResponseWriter, r **\***http.Request) {

fmt.Fprintln(w, "bar")

}

**func** main() {

http.HandleFunc("/foo", logging(foo))

http.HandleFunc("/bar", logging(bar))

http.ListenAndServe(":8080", **nil**)

}

$ go run basic-middleware.go

2017/02/10 23:59:34 /foo

2017/02/10 23:59:35 /bar

2017/02/10 23:59:36 /foo?bar

$ curl -s http://localhost:8080/foo

$ curl -s http://localhost:8080/bar

$ curl -s http://localhost:8080/foo?bar

# Middleware (Advanced)

This example will show how to create a more advanced version of middleware in Go.

A middleware in itself simply takes a http.HandlerFunc as one of its parameters, wraps it and returns a new http.HandlerFunc for the server to call.

Here we define a new type Middleware which makes it eventually easier to chain multiple middlewares together. This idea is inspired by Mat Ryers’ talk about Building APIs. You can find a more detailed explaination including the talk [here](https://medium.com/@matryer/writing-middleware-in-golang-and-how-go-makes-it-so-much-fun-4375c1246e81).

This snippet explains in detail how a new middleware is created. In the full example below, we reduce this version by some boilerplate code.

**func** createNewMiddleware() Middleware {

*// Create a new Middleware*

middleware **:=** **func**(next http.HandlerFunc) http.HandlerFunc {

*// Define the http.HandlerFunc which is called by the server eventually*

handler **:=** **func**(w http.ResponseWriter, r **\***http.Request) {

*// ... do middleware things*

*// Call the next middleware/handler in chain*

next(w, r)

}

*// Return newly created handler*

**return** handler

}

*// Return newly created middleware*

**return** middleware

}

This is the full example:

*// advanced-middleware.go*

**package** main

**import** (

"fmt"

"log"

"net/http"

"time"

)

**type** Middleware **func**(http.HandlerFunc) http.HandlerFunc

*// Logging logs all requests with its path and the time it took to process*

**func** Logging() Middleware {

*// Create a new Middleware*

**return** **func**(f http.HandlerFunc) http.HandlerFunc {

*// Define the http.HandlerFunc*

**return** **func**(w http.ResponseWriter, r **\***http.Request) {

*// Do middleware things*

start **:=** time.Now()

**defer** **func**() { log.Println(r.URL.Path, time.Since(start)) }()

*// Call the next middleware/handler in chain*

f(w, r)

}

}

}

*// Method ensures that url can only be requested with a specific method, else returns a 400 Bad Request*

**func** Method(m **string**) Middleware {

*// Create a new Middleware*

**return** **func**(f http.HandlerFunc) http.HandlerFunc {

*// Define the http.HandlerFunc*

**return** **func**(w http.ResponseWriter, r **\***http.Request) {

*// Do middleware things*

**if** r.Method **!=** m {

http.Error(w, http.StatusText(http.StatusBadRequest), http.StatusBadRequest)

**return**

}

*// Call the next middleware/handler in chain*

f(w, r)

}

}

}

*// Chain applies middlewares to a http.HandlerFunc*

**func** Chain(f http.HandlerFunc, middlewares **...**Middleware) http.HandlerFunc {

**for** \_, m **:=** **range** middlewares {

f = m(f)

}

**return** f

}

**func** Hello(w http.ResponseWriter, r **\***http.Request) {

fmt.Fprintln(w, "hello world")

}

**func** main() {

http.HandleFunc("/", Chain(Hello, Method("GET"), Logging()))

http.ListenAndServe(":8080", **nil**)

}

$ go run advanced-middleware.go

2017/02/11 00:34:53 / 0s

$ curl -s http://localhost:8080/

hello world

$ curl -s -XPOST http://localhost:8080/

Bad Request

# Sessions

This example will show how to store data in session cookies using the popular[gorilla/sessions](https://github.com/gorilla/sessions) package in Go.

Cookies are small pieces of data stored in the browser of a user and are sent to our server on each request. In them, we can store e.g. whether or not a user is logged in into our website and figure out who he actually is (in our system).

In this example we will only allow authenticated users to view our secret message on the /secret page. To get access to it, the will first have to visit /login to get a valid session cookie, which logs him in. Additionally he can visit /logout to revoke his access to our secret message.

*// sessions.go*

**package** main

**import** (

"fmt"

"net/http"

"github.com/gorilla/sessions"

)

**var** (

*// key must be 16, 24 or 32 bytes long (AES-128, AES-192 or AES-256)*

key = []byte("super-secret-key")

store = sessions.NewCookieStore(key)

)

**func** secret(w http.ResponseWriter, r **\***http.Request) {

session, \_ **:=** store.Get(r, "cookie-name")

*// Check if user is authenticated*

**if** auth, ok **:=** session.Values["authenticated"].(**bool**); !ok **||** !auth {

http.Error(w, "Forbidden", http.StatusForbidden)

**return**

}

*// Print secret message*

fmt.Fprintln(w, "The cake is a lie!")

}

**func** login(w http.ResponseWriter, r **\***http.Request) {

session, \_ **:=** store.Get(r, "cookie-name")

*// Authentication goes here*

*// ...*

*// Set user as authenticated*

session.Values["authenticated"] = **true**

session.Save(r, w)

}

**func** logout(w http.ResponseWriter, r **\***http.Request) {

session, \_ **:=** store.Get(r, "cookie-name")

*// Revoke users authentication*

session.Values["authenticated"] = **false**

session.Save(r, w)

}

**func** main() {

http.HandleFunc("/secret", secret)

http.HandleFunc("/login", login)

http.HandleFunc("/logout", logout)

http.ListenAndServe(":8080", **nil**)

}

$ go run sessions.go

$ curl -s http://localhost:8080/secret

Forbidden

$ curl -s -I http://localhost:8080/login

Set-Cookie: cookie-name=MTQ4NzE5Mz...

$ curl -s --cookie "cookie-name=MTQ4NzE5Mz..." http://localhost:8080/secret

The cake is a lie!

# JSON

This example will show how to encode and decode JSON data using the encoding/json package.

*// json.go*

**package** main

**import** (

"encoding/json"

"fmt"

"net/http"

)

**type** User **struct** {

Firstname **string** `json:"firstname"`

Lastname **string** `json:"lastname"`

Age **int** `json:"age"`

}

**func** main() {

http.HandleFunc("/decode", **func**(w http.ResponseWriter, r **\***http.Request) {

**var** user User

json.NewDecoder(r.Body).Decode(**&**user)

fmt.Fprintf(w, "%s %s is %d years old!", user.Firstname, user.Lastname, user.Age)

})

http.HandleFunc("/encode", **func**(w http.ResponseWriter, r **\***http.Request) {

peter **:=** User{

Firstname: "John",

Lastname: "Doe",

Age: 25,

}

json.NewEncoder(w).Encode(peter)

})

http.ListenAndServe(":8080", **nil**)

}

$ go run json.go

$ curl -s -XPOST -d'{"firstname":"Donald","lastname":"Trump","age":70}' http://localhost:8080/decode

Donald Trump is 70 years old!

$ curl -s http://localhost:8080/encode

{"firstname":"John","lastname":"Doe","age":25}

# Websockets

This example will show how to work with websockets in Go. We will build a simple server which echoes back everything we send to it. For this we have to go get the popular [gorilla/websocket](https://github.com/gorilla/websocket) library like so:

$ go get github.com/gorilla/websocket

From now on, every application we write will be able to make use of this library.

*// websockets.go*

**package** main

**import** (

"fmt"

"net/http"

"github.com/gorilla/websocket"

)

**var** upgrader = websocket.Upgrader{

ReadBufferSize: 1024,

WriteBufferSize: 1024,

}

**func** main() {

http.HandleFunc("/echo", **func**(w http.ResponseWriter, r **\***http.Request) {

conn, \_ **:=** upgrader.Upgrade(w, r, **nil**) *// error ignored for sake of simplicity*

**for** {

*// Read message from browser*

msgType, msg, err **:=** conn.ReadMessage()

**if** err **!=** **nil** {

**return**

}

*// Print the message to the console*

fmt.Printf("%s sent: %s\n", conn.RemoteAddr(), string(msg))

*// Write message back to browser*

**if** err = conn.WriteMessage(msgType, msg); err **!=** **nil** {

**return**

}

}

})

http.HandleFunc("/", **func**(w http.ResponseWriter, r **\***http.Request) {

http.ServeFile(w, r, "websockets.html")

})

http.ListenAndServe(":8080", **nil**)

}

*<!-- websockets.html -->*

<input id**=**"input" type**=**"text" />

<button onclick**=**"send()">Send</button>

<pre id**=**"output"></pre>

<script>

**var** input **=** document.getElementById("input");

**var** output **=** document.getElementById("output");

**var** socket **=** **new** WebSocket("ws://localhost:8080/echo");

socket.onopen **=** **function** () {

output.innerHTML **+=** "Status: Connected\n";

};

socket.onmessage **=** **function** (e) {

output.innerHTML **+=** "Server: " **+** e.data **+** "\n";

};

**function** send() {

socket.send(input.value);

input.value **=** "";

}

</script>

$ go run websockets.go

[127.0.0.1]:53403 sent: Hello Go Web Examples, you're doing great!

 Send

Status: Connected

Server: Hello Go Web Examples, you’re doing great!

Bottom of Form