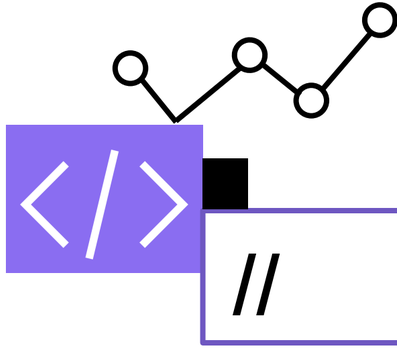
DIGITALHOUSE

Servidor web en Go

Certified Tech Developer
The Ultimate Degree

Índice

01 Package net/http



Package net/http

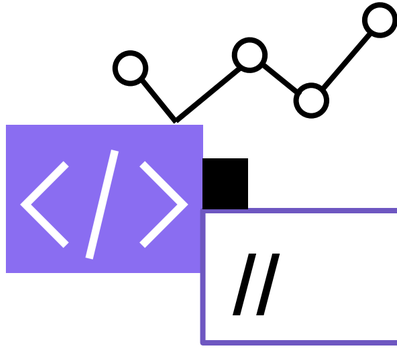
Este paquete permite generar servidores web de una manera simple. Un concepto fundamental en los servidores **net/http** son los **handlers**. Un handler es un objeto que implementa la interfaz **http.Handler**. Una forma común de escribir un handler es usar el adaptador **http.HandlerFunc** en funciones con la firma adecuada.

Las funciones que sirven como handler toman un `http.ResponseWriter` y un `http.Request` como argumentos. El **ResponseWriter** se utiliza para devolver la respuesta HTTP.

Aquí nuestra respuesta es simplemente "holaHandler".

```
{}
```

```go
func holaHandler(w http.ResponseWriter, req *http.Request) {
   fmt.Fprintf(w, "hola\n")
}
```

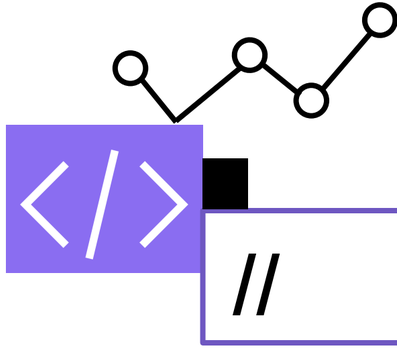


Package net/http

Registramos nuestros handlers en las rutas del servidor utilizando la función **http.HandleFunc**. Configuramos el router predeterminado en el paquete net/http y tomamos una función como argumento.

Finalmente, llamamos a **ListenAndServe** con el puerto “:8080” y un controlador. El **nil** le dice que use el router predeterminado que acabamos de configurar.

```
{}
func main() {
   http.HandleFunc("/hola", holaHandler)
   http.ListenAndServe(":8080", nil)
}
```



Package net/http

Corremos el servidor web de la siguiente manera:

```
{}  go run main.go
```

Para probar nuestro endpoint entramos a la siguiente URL http://localhost:8080/hola.

Deberíamos obtener como respuesta el siguiente texto: “hola”.

Ejemplo completo

```go
package main

import (
    "fmt"
    "net/http"
)

func holaHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "hola\n")
}

func main() {
    http.HandleFunc("/hola", holaHandler)
    http.ListenAndServe(":8080", nil)
}
```

02 Gin web framework

¿Qué es?

Así como podemos generar un servidor con el package net/http, también existen otros frameworks que nos permiten crear un web server.

Gin es un **microframework de alto rendimiento** que se puede utilizar para crear aplicaciones web y microservicios en Go.

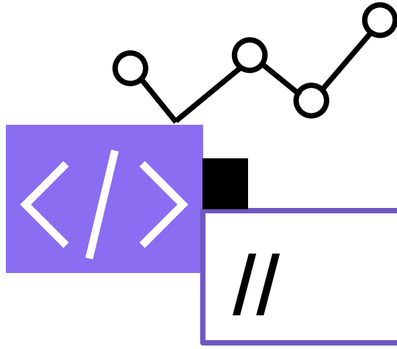
Este contiene un conjunto de funcionalidades (por ejemplo: routing, middleware, rendering, etc.) que reducen el código repetitivo y simplifican la creación de aplicaciones web y microservicios.

¿Cómo funciona?

Veamos rápidamente cómo Gin procesa una solicitud. El flujo de control para una aplicación web típica, un servidor API o un microservicio, es el siguiente:

Cuando llega una solicitud de un cliente, Gin primero analiza la ruta. Si se encuentra una definición de ruta coincidente, Gin invoca los middlewares (son opcionales) en un orden definido por la definición de ruta (si es que tienen) y el handler de ruta.

03 Crear un servidor web



1. Crear repositorio

Crear repos en GitHub:

https://github.com/new

Luego de haber creado el repositorio, debemos clonarlo:

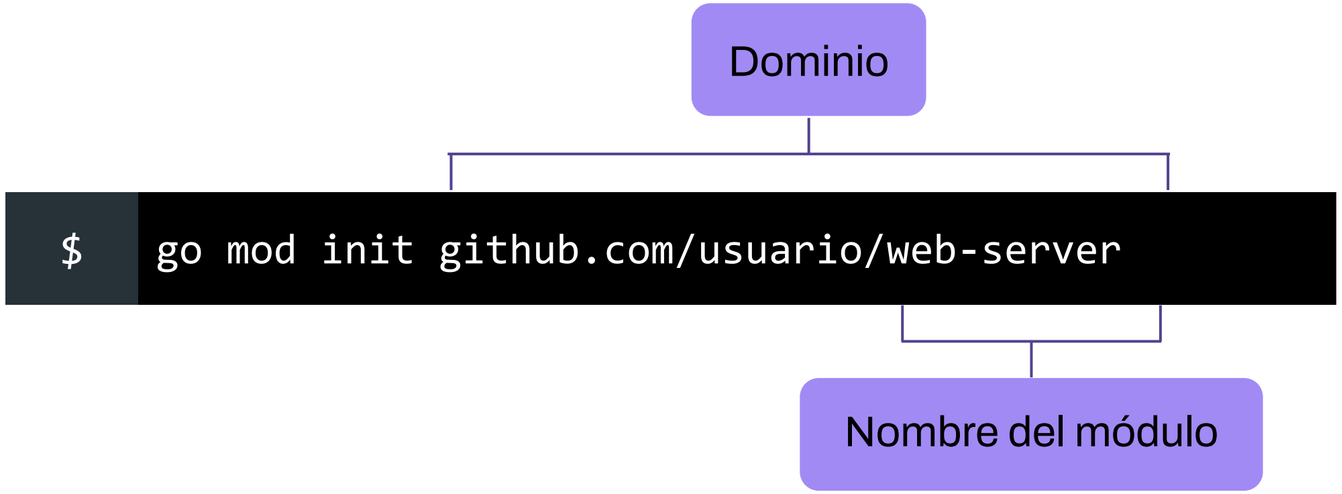
```
$ git clone github.com/usuario/web-server
```

Abrimos nuestra carpeta con el Visual Studio Code:

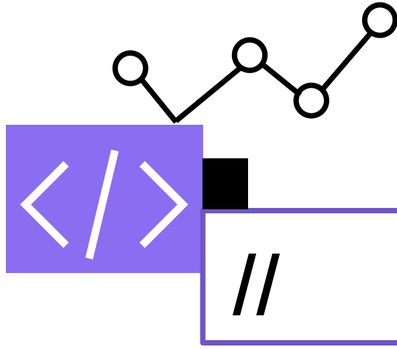
```
$ code ./web-server/
```

2. Inicializar el módulo

Luego de haber creado y clonado el repositorio, debemos inicializar nuestro módulo.
Para esto, utilizamos el siguiente comando:



El dominio y el nombre del módulo deben coincidir con el nombre del repositorio que ya se creó.



3. Obtener Gin

Para utilizar Gin se requiere la versión 1.13+ de Go. Una vez instalada, utilizamos el siguiente comando para instalar Gin:

```
$ go get -u github.com/gin-gonic/gin
```

Luego lo importamos a nuestro código:

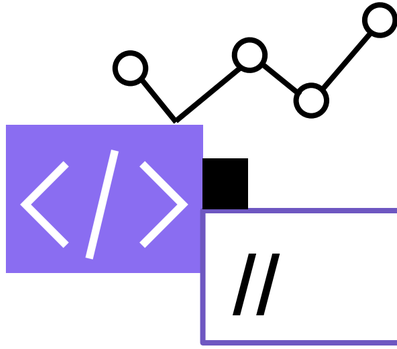
```
{} import "github.com/gin-gonic/gin"
```

4. Crear nuestro router con Gin

Ya teniendo instalado Gin, creamos un servidor web simple. `gin.Default()` crea un router de Gin con dos middlewares por defecto: **logger** y **recovery**.

```
{}
// Crea un router con Gin
router  := gin.Default()
```

Ya teniendo definido nuestro router, este nos permite ir agregando los distintos endpoints que tendrá nuestra aplicación. Para ello debemos agregar, al router, distintos handlers.



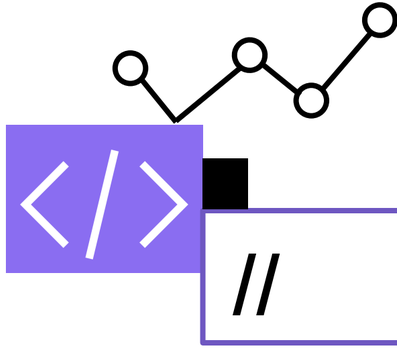
5. Crear nuestro handler

A continuación, creamos un handler utilizando la función `router.GET("endpoint", Handler)` donde endpoint es la ruta relativa y handler es la función que toma `*gin.Context` como argumento.

En el siguiente ejemplo, la función de handler sirve una respuesta JSON con un estado de 200:

{}

```
// Captura la solicitud GET “/hello-world”
router.GET("/hello-world", func(c *gin.Context) {
    c.JSON(200, gin.H{
        "message": "Hello World!",
    })
})
```



6. Correr nuestro servidor

Por último, iniciamos el router usando **router.Run()** que, por defecto, escucha en el puerto 8080.

```
{}  router.Run() // Corremos nuestro servidor sobre el puerto 8080
```

Para correr nuestra aplicación, lo que hacemos es correr el siguiente comando:

```
$  go run main.go
```

Para probar nuestro endpoint entramos a la siguiente URL: http://localhost:8080/hello-world.
Deberíamos obtener como respuesta el siguiente JSON: **{"message":"Hello World!"}**

Ejemplo completo

```go
package main

import "github.com/gin-gonic/gin"

func main() {
    // Crea un router con gin
    router := gin.Default()

    // Captura la solicitud GET “/hello-world”
    router.GET("/hello-world", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "Hello World!",
        })
    })

    // Corremos nuestro servidor sobre el puerto 8080
    router.Run()
}
```

net/http

VS.

gin-gonic

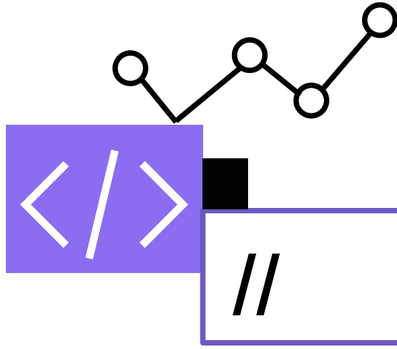
```go
func main() {
    http.HandleFunc("/saludo", func(w
http.ResponseWriter, r *http.Request){
        fmt.Fprintf(w, "Hola Bootcampers!")
    })


    fmt.Printf("Starting server at port
8080\n")
    if err := http.ListenAndServe(":8080",
nil); err != nil {
        log.Fatal(err)
    }
}
```

```go
func main() {
    s := gin.New()
    s.GET("/saludo", func(c *gin.Context)
{
        c.String(http.StatusOK, "Hola
Bootcampers!")
    })
    s.Run()
}
```

Como podemos observar en estos ejemplos, Gin resulta más práctico que el package http. **¡Recordemos que el package http es utilizado en Gin!**

04 Función Marshal



.Marshal( )

La función `func Marshal(v interface{}) ([]byte, error)` toma como parámetro un valor de cualquier tipo, y retorna una slice de bytes que contiene su representación en formato JSON. También retorna un error en caso de encontrar uno.

En caso de querer usar json.Marshal( ) y pasar un struct como parámetro, los campos a transformar a JSON tienen que estar exportados, es decir, en mayúsculas.

json.Marshal( )

En el ejemplo de la derecha, definimos un struct **product** con sus campos **Name**, **Price** y **Published** de distintos tipos. Todos exportados (en mayúsculas) para su posterior transformación a JSON con la función **Marshal()**.

Paso siguiente llamamos a la función que devuelve un error que verificamos, e imprimimos por pantalla el resultado convertido a string, ya que es de tipo **[]byte** el cual es convertible a **string**.

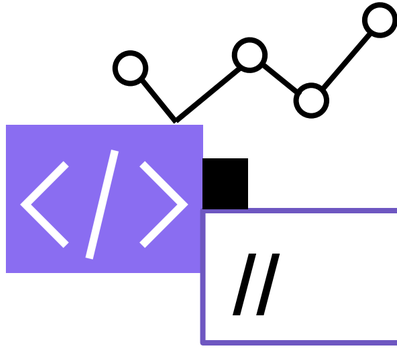
```
type product struct {
    Name      string
    Price     int
    Published bool
}

p := product{
    Name:      "MacBook Pro",
    Price:     1500,
    Published: true,
}

jsonData, err := json.Marshal(p)
if err != nil {
    log.Fatal(err)
}

fmt.Println(string(jsonData))
```

05 Función Unmarshal



json.Unmarshal( )

La función `func Unmarshal(data []byte, v interface{}) error` recibe como primer parámetro un array de bytes y, como segundo parámetro, un puntero a un struct. Si el array de bytes es data en JSON, entonces la función **unmarshal()** va a tratar de decodificarlo y llenar el struct con esos datos. La función devuelve un error, en caso de encontrar uno.

Aquí, definimos un string llamado **jsonData**, el cual contiene un objeto de tipo JSON válido, que corresponde con la forma de nuestro **struct product** previamente definido. Paso siguiente creamos un struct de tipo product vacío llamado **p** y se lo pasamos a la función **unmarshal()** junto con nuestro JSON **string** convertido a slice de bytes. Finalmente, imprimimos por pantalla el resultado.

```
type product struct {
    Name      string
    Price     int
    Published bool
}

jsonData := `{"Name": "MacBook Air", "Price": 900, "Published":
true}`

var p product

if err := json.Unmarshal([]byte(jsonData), &p); err != nil {
    log.Fatal(err)
}
```

DIGITALHOUSE

¡Muchas gracias!