GO

BOOTCAMP

Clase en vivo

//Go Web

IT BOARDING

BOOTCAMP

Objetivos de la clase:

- Conocer e introducirnos en el uso del package “encoding/json”.
- Implementar nuestro primer servidor web con el uso de un multiplexer llamado Chi.

Índice

IT BOARDING
BOOTCAMP

1 | Repaso

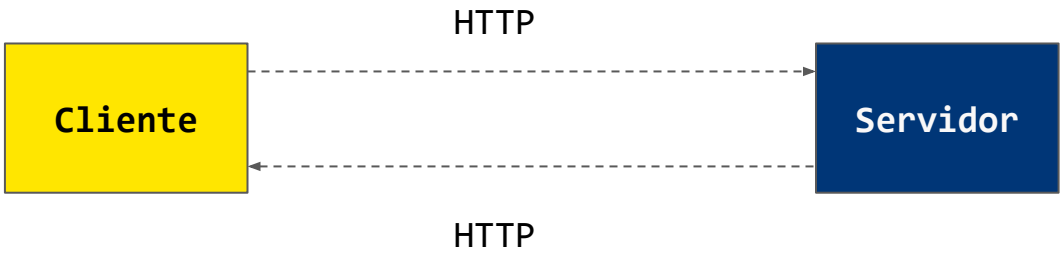
IT BOARDING

BOOTCAMP

APIs

Cliente-servidor

La Arquitectura web es una arquitectura del tipo cliente-servidor porque debe permitir que tanto la aplicación del cliente como la aplicación del servidor se desarrollen o escalen sin interferir una con la otra. Es decir, permite integrar con cualquier otra plataforma y tecnología tanto el cliente como el servidor.



// ¿Qué es REST?

REST (Representational State Transfer) es el nombre que le dio Fielding a la Arquitectura Web compuesta por las claves vistas anteriormente.

JSON

Sintaxis de un JSON

JSON

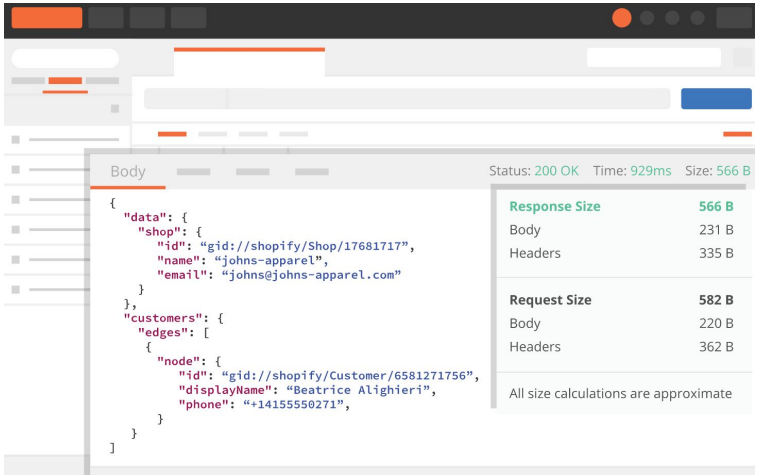
```json
{
  "string": "my string",
  "integer": 1,
  "float": 2.59,
  "bool": true,
  "array": ["prod01", "prod02", "prod03"],
  "object": {
    "number_1": 1,
    "number_2": 2
  }
}
```

POSTMAN

¿Qué es Postman?

Es una aplicación utilizada para pruebas de APIs webs.

Una de sus funciones es simular un cliente REST con el que podremos enviar peticiones a nuestra aplicación y ver como responde.



2 | Package JSON

IT BOARDING

BOOTCAMP

// ¿Qué es el package json?

El package “json” es una librería que nos permite transformar estructuras y tipos de datos en Go a JSON y viceversa.

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.

{}

```go
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))
```

json.Marshal( )

En el ejemplo anterior 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( )**.

A continuación imprimimos por pantalla el resultado convertido a string ya que es de tipo **[]byte** el cual es convertible a **string**.

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 de cualquier tipo en dónde se almacenarán los datos decodificados. Si el array de bytes es data en JSON, entonces la función **Unmarshal()** va a tratar de decodificarlo y llenar con esos datos el elemento al que apuntemos.

La función devuelve un error, en caso de encontrar uno.

{}

```go
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)
}

fmt.Println(p)
```

.Unmarshal( )

En el ejemplo anterior 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. A continuación 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 mostramos por pantalla el resultado.

Decoder/Encoder

Decoder/Encoder

De forma análoga a las funciones anteriores (Marshal/Unmarshal) El package json también dispone de los types Decoder/Encoder, los cuales exponen interfaces que permite transformar data a json y viceversa. A diferencia de las funciones Marshall/Unmarshall, Decoder/Encoder lo hacen sobre un streaming y no sobre una interfaz.

Encoder Ejemplo

{}

```
// It is necessary to create our type Encode
// for this the NewEncoder function is called
// this receives a streaming as a parameter
// we use one of the standard streams offered by the OS Stdout pkg
// stdout generates a stream to a file that is printed to the console.

 myEncoder := json.NewEncoder(os.Stdout)

// prepare the information you want to send in json format to the streaming
type MyData struct {
    ProductID string
    Price     float64
}
data := MyData{
    ProductID: "XASD",
    Price:     25.50,
}


// the Encode method is invoked.
// internally this method makes a kind of marshall and writes it to the stream

myEncoder.Encode(data)
```

Decoder Ejemplo

```
    {"ProductID": "AXW123", "Price": 25.50}
    {"ProductID": "NLBR17", "Price": 357.58}
    {"ProductID": "KNUB82", "Price": 150}
    `
    // It is necessary to create our type Decode, for this the NewDecoder function is called
    // this receives a streaming as a parameter
    // A jsonStream variable is created and the NewReader method of the strings pkg is used
    // NewReader generates a streaming for the text string it receives.

    myStreaming := strings.NewReader(jsonStream)
    myDecoder := json.NewDecoder(myStreaming)
    type MyData struct {
        ProductID string
        Price     float64
    }
    // streaming behaves so that each line in the jsonStrem text is streamed separately
    // we go through all the data transmitted in the streaming until the end of the text is reached
    for {
        // the variable on which the data is going to be written is created
        var data MyData
        // the decode method is invoked
        // Decode is responsible for reading the data transmitted by the streaming and transforming it from json to
our interface
        if err := myDecoder.Decode(&data); err == io.EOF {
            break
        } else if err != nil {
            log.Fatal(err)
        }
        // The received data is printed
        log.Println("Data:", data.ProductID, data.Price)
    }
```

A codear...

3 Creando nuestro servidor

IT BOARDING

BOOTCAMP

¿Que es Chi?

Así como podemos generar un servidor con el package net/http también existen otras herramientas que nos facilitan crear un web server.

Chi es un enrutador (multiplexer) liviano, idiomático y de alto rendimiento que se puede utilizar para crear aplicaciones web y microservicios en Go. Está especializado para REST-API. Mantiene un entorno de trabajo cercano a lo nativo en go.

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

¿Cómo funciona Chi?

Veamos rápidamente cómo Chi procesa una solicitud. El flujo de control para una aplicación web típica, un servidor API o un microservicio.

Cuando llega una solicitud de un cliente, Chi primero analiza la ruta. Si se encuentra una definición de ruta coincidente, Chi invoca los middleware (son opcionales) en un orden definido por la definición de ruta (si es que tienen) y el handler de ruta. Aplica el patrón **decorator** entre los **handlers** y **middlewares** a través de la interfaz **http.Handler,** para su ejecución.

Obtener Gin

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

```
$ go get -u github.com/go-chi/chi/v5
```

Luego lo importamos a nuestro código.

```
{} import "github.com/go-chi/chi/v5"
```

Crear nuestro router con Chi

Ya teniendo instalado Chi, creamos un servidor web simple. `chi.NewRouter()` el cual genera un router de Chi.

```
{}
// Create a router with chi
router := chi.NewRouter()
```

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.

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 nativa de go para los handlers (**http.HandlerFunc**), que toma como parámetro **http.ResponseWriter** y ***http.Request**. En el siguiente ejemplo, la función de Handler sirve una respuesta JSON con un estado de 200.

{}

```
// Create a new endpoint GET “/hello-world”
router.GET("/hello-world", func(w http.ResponseWriter, r *http.Request) {
    // set code
    w.WriteHeader(200)
    // set body
    w.Write(byte[](`{"message":"Hello World!"}`))
})
```

Correr nuestro servidor

Por último, iniciamos el router usando **http.ListenAndServe()** que pide el address (**addr**) compuesto de un **host** y un **port**; además un **http.Handler** que es una interfaz con un método de escucha, que en este caso usaremos el **router** de **chi**

```
{}
    // server
    rt := chi.NewRouter()
    // run
    http.ListenAndServe(":8080", rt)
```

Para correr nuestra aplicación, lo que hacemos es utilizar 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 (
    "fmt"
    "net/http"
    "github.com/go-chi/chi/v5"
)

func main() {
    // server
    rt := chi.NewRouter()
    // -> endpoints
    rt.Get("/hello-world", func(w http.ResponseWriter, r *http.Request) {
        // set code and body
        w.WriteHeader(http.StatusOK); .Write([]byte("Hello World!"))
    })
    // run
    if err := http.ListenAndServe(":8080", rt); err != nil {
        panic(err)
    }
}
```

A codear...

Conclusiones

En esta clase aprendimos cómo usar el package Json para mapear nuestros datos recibidos sobre estructuras y viceversa.

Además aprendimos a implementar Chi para crear aplicaciones con este enrutador web

Actividad

Gracias.

IT BOARDING

BOOTCAMP