INTRODUCCIÓN A LA CLASE

GO WEB

Objetivos de la clase

A lo largo de esta clase nos estaremos adentrando dentro de conceptos bastante interesantes que nos permitirán construir de una manera efectiva nuestras APIs.

Puntualmente lo que estaremos aprendiendo será:

- Qué es el sistema REST y para qué lo usamos dentro de una API.
- Cómo podemos construir nuestra propia API.
- Cómo Postman nos permite testear algunos endpoints de nuestras APIs.
- Conocer y utilizar GIN.
- Crear una API REST Básica.

Sin más, vayamos entonces a aprender un poco más al respecto.

INTRODUCCIÓN A API

GO WEB

API

// ¿Qué es una API? y ¿Para qué sirve?

IT BOARDING

BOOTCAMP

¿Qué es un API?

El término API es una abreviatura de Application Programming Interfaces, que en español significa interfaz de programación de aplicaciones. Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

Podemos hablar de una API como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que vayan a utilizar, y de los permisos que les dé el propietario de la API a los desarrolladores.

¿Para qué sirve una API?

Una de las principales funciones de las API es poder facilitarle el trabajo a los desarrolladores y ahorrarles tiempo y dinero. Por ejemplo, si estás creando una aplicación que es una tienda online, no necesitarás crear desde cero un sistema de pagos. Podrás utilizar la API de un servicio de pago ya existente, por ejemplo PayPal.
Con ello, no será necesario tener que reinventar la rueda con cada servicio que se crea, ya que podrás utilizar piezas o funciones que otros ya han creado.

A veces otros servicios crean API de forma deliberada para ser utilizados por terceros en tareas concretas, y así extender su uso y popularidad creando nuevas funciones. Por ejemplo, Google creó una para Google Docs con las que permite la creación automatizada de facturas o informes de ventas a otros servicios.

Estos son algunos de los tantos usos que le podemos dar a una API.

HTTP

// Protocolo de comunicación

IT BOARDING

BOOTCAMP

// ¿Qué es HTTP?

“HTTP es un protocolo de transferencia que permite de manera estandarizada la comunicación entre el cliente y el servidor.”

ARQUITECTURA WEB

// Seis claves según Roy Fielding

IT BOARDING

BOOTCAMP

Arquitectura Web

En 1993, Roy Fielding, co-fundador del proyecto Apache HTTPD, se preocupó por el problema de escalabilidad de la Web.

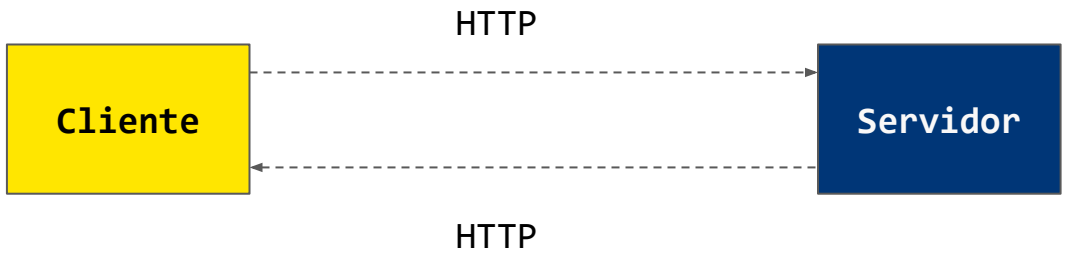
Tras el análisis, Fielding reconoció que la escalabilidad de la Web se regía por un conjunto de claves.

Las claves, que Fielding agrupó en seis categorías y llamó Arquitectura Web, son:

1. Cliente-servidor
2. interfaz uniforme
3. Sistema de capas
4. Cache
5. Sin estado (stateless)
6. Código a demanda

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.



Interfaz uniforme

Desde el lado del servidor, una **arquitectura web** expone a los clientes a una interfaz uniforme.

- Todos los recursos del servidor tienen un nombre en forma de URL o hipervínculo.
- Toda la información se intercambia a través del protocolo HTTP.

Un endpoint está ligado al recurso que solicitamos, dicho recurso debe tener solamente un identificador lógico, y esté proveer acceso a toda la información relacionada.

Sistema de capas

La clave, sistema de capas, permite que los intermediarios en la red, como los proxies y gateways, se implementen de manera transparente entre un cliente y un servidor utilizando la interfaz uniforme de la Web.

En términos generales, un intermediario en la red intercepta la comunicación cliente-servidor para un propósito específico.

Los usos más comunes son para la aplicación de la seguridad, el almacenamiento en caché de respuesta y el equilibrio de carga.

Cache

El cacheo de datos es una de las claves más importantes de la arquitectura web. Ayuda a reducir la latencia percibida por el cliente, aumentar la disponibilidad y la confiabilidad de una aplicación, y a controlar la carga de un servidor web.

En una palabra, caché reduce el costo total de la Web.

Puede existir una caché en cualquier lugar entre cliente y el servidor.

Sin estado (stateless)

La clave Stateless propone que todas las interacciones entre el cliente y el servidor deben ser tratadas como nuevas y de forma absolutamente independiente sin guardar estado.

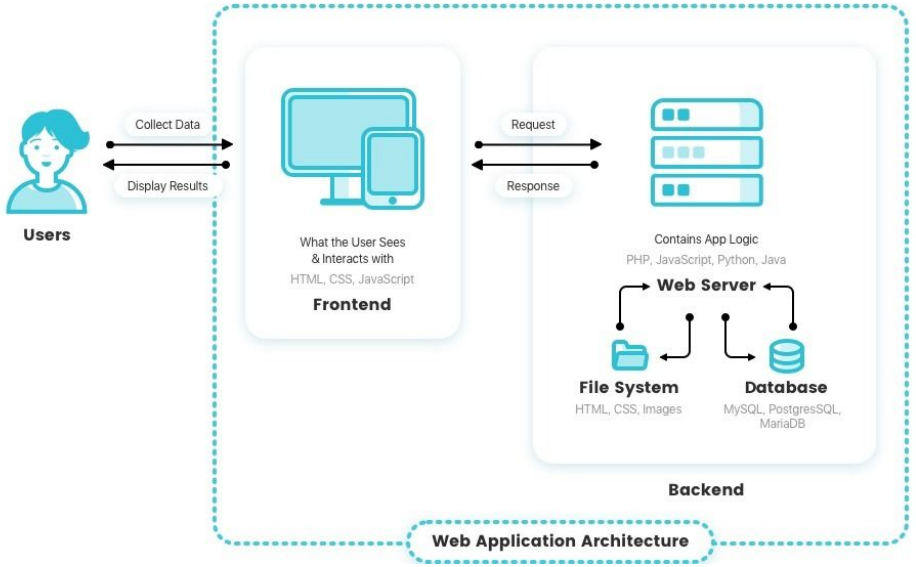
Por lo tanto, si quisiéramos —por ejemplo— que el servidor distinga entre usuarios logueados o invitados, debemos mandar toda la información de autenticación necesaria en cada petición que le hagamos a dicho servidor.

Esto permite desarrollar aplicaciones más **escalables**.

Código a demanda

El código bajo demanda tiende a establecer un acoplamiento tecnológico entre los servidores web y sus clientes, ya que el cliente debe ser capaz de comprender y ejecutar el código que descarga bajo demanda del servidor.

Las tecnologías que soportan los navegadores web son: HTML, Css, JavaScript y entre otras.



// ¿Qué es REST?

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

IT BOARDING
BOOTCAMP

JSON

GO WEB

“JSON se posiciona como una opción eficiente para lograr un intercambio de datos más liviano y rápido”.

¿Qué es JSON?

// Concepto y características

IT BOARDING

BOOTCAMP

// ¿Qué es json?

"JSON es el acrónimo de JavaScript Object Notation, o Notación de Objetos de JavaScript. Es un formato ligero de intercambio de datos".

IT BOARDING
BOOTCAMP

Características principales:

- Aunque su nombre lo diga, no es necesariamente parte de JavaScript, de hecho es un estándar basado en texto plano para el intercambio de información.
- Es solo un formato para la escritura de datos.
- Requiere usar comillas dobles para las cadenas y los nombres de propiedades. Las comillas simples no son válidas.
- Una coma o dos puntos mal ubicados pueden producir que un archivo JSON no funcione.
- Puede tomar la forma de cualquier tipo de datos que sea válido para ser incluido en un JSON, no solo arreglos u objetos.
- Es un formato común para ‘serializar’ y ‘deserializar’ objetos en la mayoría de los idiomas

JSON

// Sintaxis

IT BOARDING

BOOTCAMP

Sintaxis de un JSON #1

JSON se basa en la sintaxis que tiene Javascript para crear objetos. Puede contener varios tipos de datos. Está constituído por dos estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, o listas.

JSON puede representar cuatro **tipos primitivos** (cadenas, números, booleanos y valores nulos) y dos **tipos estructurados** (objetos y arreglos).

Sintaxis de un JSON #2

En **JSON**:

- Una **Cadena** es una secuencia de ceros o más caracteres Unicode.
- Un **Objeto** es una colección desordenada de cero o más pares “*nombre:valor*”, donde un nombre es una cadena y un valor es una cadena, numero, booleano, nulo, objeto o arreglo.
- Un **Arreglo** es una secuencia desordenada de ceros o más valores.

Sintaxis de un JSON #3

Veamos ejemplos.

- un JSON debe tener como mínimo, la sintaxis de un objeto vacío:

```JSON
{
}
```

Sintaxis de un JSON #4

Un JSON puede contener varios tipos de datos:

JSON

```json
{
  "cadena": "mi cadena",
  "int": 1,
  "booleano": true,
  "array": ["prod01", "prod02", "prod03"],
  "objeto": {
    "llave": 1,
    "otra": 2
  }
}
```

UTILIDAD

// Ventajas y desventajas

IT BOARDING

BOOTCAMP

“JSON puede ser leído por cualquier lenguaje de programación. Por lo tanto, puede ser usado para el intercambio de información entre distintas tecnologías o lenguajes”.

Ventajas:

- Es autodescriptivo y fácil de entender.
- Su sencillez le ha permitido posicionarse como alternativa a XML.
- Es más rápido en cualquier navegador.
- Es más fácil de leer que XML.
- Es más ligero (bytes) en las transmisiones.
- Se parsea más rápido.
- Velocidad de procesamiento alta.

Desventajas:

- Algunos desarrolladores encuentran su escueta notación algo confusa.
- No cuenta con una característica que posee XML: extensibilidad.
- No soporta grandes cargas, solo datos comunes.
- Para la seguridad requiere de mecanismos externos como expresiones regulares.

JSON vs XML

// Diferencias

IT BOARDING

BOOTCAMP

Ventajas de JSON sobre XML

XML (eXtensible Markup Language) o lenguaje de marcado extensible, se utiliza para el intercambio de datos estructurados.

Sin embargo, JSON posee claras ventajas sobre XML. Por ejemplo:

- Es significativamente menos detallado que XML.
- Se necesita menos tiempo para estructurarlo y su transmisión y el procesamiento es mucho más rápido que XML.
- Es serializado y deserializa drásticamente más rápido que XML.
- No necesita ser extensible porque es flexible por sí solo. Puede representar cualquier estructura de datos pudiendo añadir nuevos campos con total facilidad.

PACKAGE JSON

GO WEB

// ¿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.”

IT BOARDING

BOOTCAMP

Marshal

// Función

IT BOARDING

BOOTCAMP

.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( )**.

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.

Unmarshal

// Función

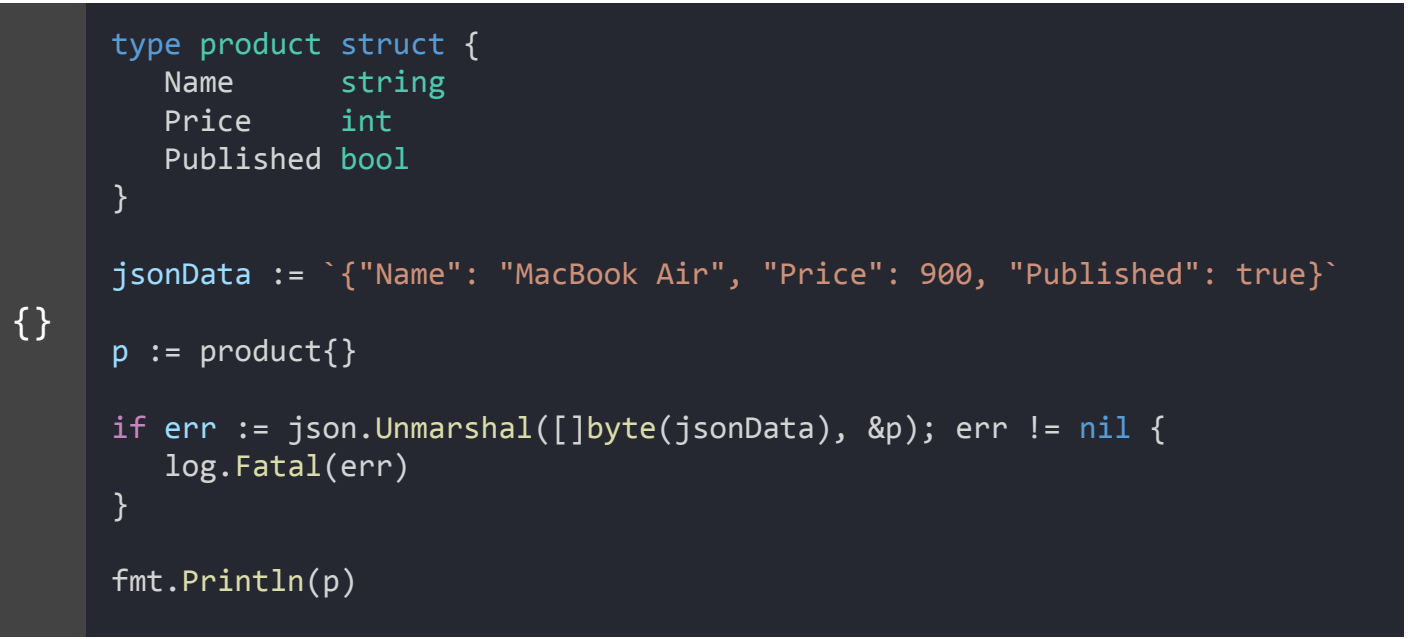
IT BOARDING

BOOTCAMP

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 unmarshall va a tratar de decodificarlo y llenar el struct con esos datos.

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}`

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. 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.

NewEncoder

// Función

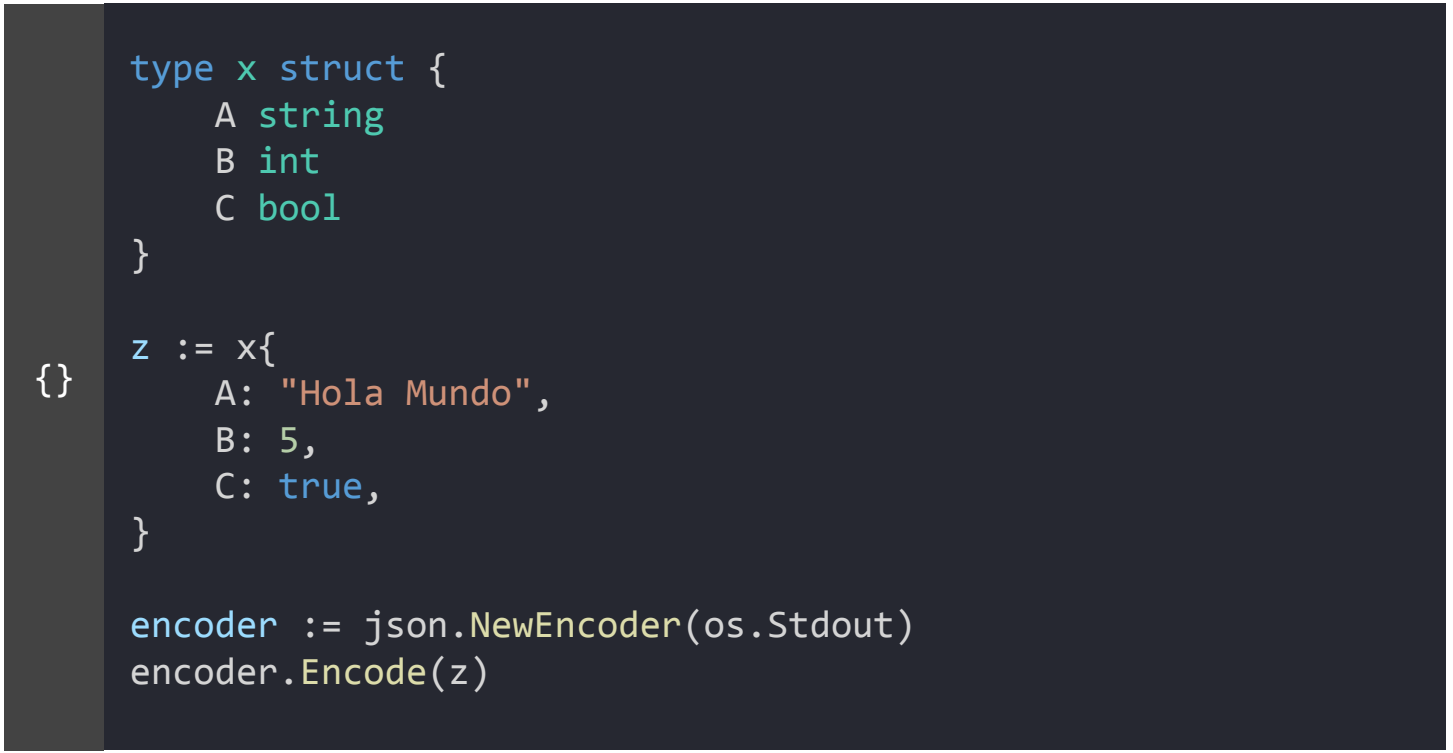
IT BOARDING

BOOTCAMP

.NewEncoder( )

Así como podemos codificar structs a JSON usando la función Marshall( ), también existe la función NewEncoder(w io.Writer) *Encoder que acepta como parámetro un struct que implemente la interfaz io.Writer y devuelve un puntero a un Encoder.

El struct Encoder implementa varios métodos, entre ellos el método Encode() que convierte un struct a notación JSON.



```
{}
```

```go
type x struct {
    A string
    B int
    C bool
}

z := x{
    A: "Hola Mundo",
    B: 5,
    C: true,
}

encoder := json.NewEncoder(os.Stdout)
encoder.Encode(z)
```

.NewEncoder( )

En el ejemplo anterior modificamos el ejemplo de marshall para usar Encode() en su reemplazo.
Primero inicializamos un struct con valores, y luego creamos un encoder con la función NewEncoder( ). Esta función toma como parámetro un io.Writer el cual os.Stdout que es la pantalla satisface.
Llamamos la función Encode( ) del Encoder y le pasamos como parámetro el struct que queremos codificar a JSON, que en este caso es z.

NewDecoder

// Función

IT BOARDING

BOOTCAMP

.NewDecoder( )

Otra forma de decodificar JSON a un struct es usando un decoder. Un decoder se crea a partir de un reader.
La función NewDecoder(r io.Reader) *Decoder toma como parámetro un struct que implemente la interfaz io.Reader y devuelve un puntero a un Decoder.
El struct decoder implementa varios métodos entre ellos el método decode que sirve para decodificar un objeto JSON a un struct.

{}

```go
type x struct {
    A string
    B int
    C bool
}

jsonData := `{
    "A": "Hola Mundo",
    "B": 5,
    "C": true}`

z := x{}

buff := bytes.NewBuffer([]byte(jsonData))
decoder := json.NewDecoder(buff)
decoder.Decode(&z)

fmt.Println(z)
```

.NewDecoder( )

En el ejemplo anterior modificamos el ejemplo de Unmarshall para usar decoder en su reemplazo.
Primero definimos el string de JSON a decodificar, y luego a partir de ese string creamos un buffer usando el paquete bytes, ya que buffer implementa la interfaz io.Reader que Decoder() necesita.
Luego creamos un Decoder a partir de este buffer, y luego llamamos al método Decode del Decoder para así llenar nuestro struct z con los valores del objeto JSON.

PACKAGE NET/HTTP

GO WEB

Package net/http

El package net/http te 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 devolver la respuesta HTTP.

Aquí nuestra respuesta simple es simplemente "hola \ n".

{}

```
func hello(w http.ResponseWriter, req *http.Request) {
   fmt.Fprintf(w, "hello\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 toma una función como argumento.
Finalmente, llamamos a ListenAndServe con el puerto y un controlador. nil le dice que use el router predeterminado que acabamos de configurar.

{}

```
func main() {
   http.HandleFunc("/hello", hello)
   http.ListenAndServe(":8090", 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:8090/hello

Deberíamos obtener como respuesta el siguiente texto: hello.

Ejemplo completo

{}

```
package main

import (
    "fmt"
    "net/http"
)

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

func main() {
    http.HandleFunc("/hello", hello)
    http.ListenAndServe(":8090", nil)
}
```

CREA UN SERVIDOR WEB

GO WEB

Nuevo Proyecto Go

// ¿Cómo crear un repo y un módulo?

IT BOARDING

BOOTCAMP

1. Crear Repositorio

Crear repos en github

https://github.com/new

Luego de haber creado el repositorio, debemos clonarlo.

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

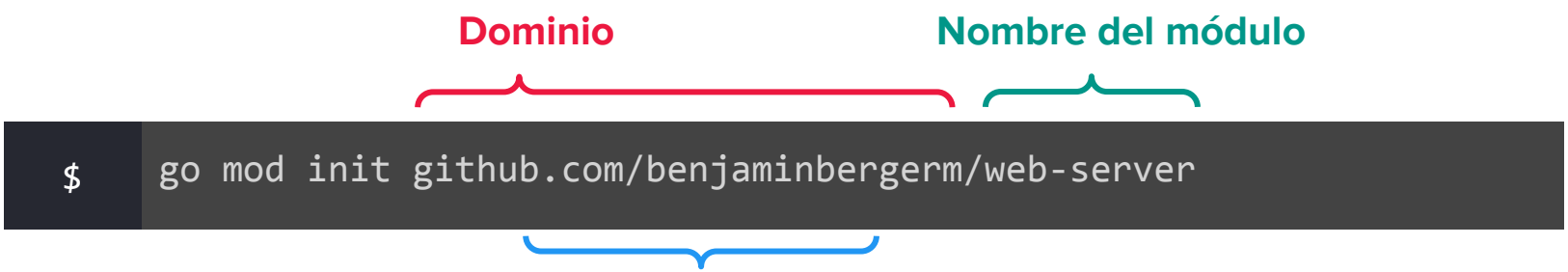
Abrimos nuestra carpeta con el Visual Studio Code.

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

2. Inicializar el módulo

Luego de haber creado y clonado el repositorio debemos inicializar nuestro módulo.

Para iniciar un módulo, utiliza este comando:



Gin Web Framework

// ¿Qué es? y ¿Cómo se implementa en GO?

IT BOARDING

BOOTCAMP

¿Que es Gin?

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 micro-framework de alto rendimiento que se puede utilizar para crear aplicaciones web y microservicios en Go.

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 Gin?

Veamos rápidamente cómo Gin 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, Gin primero analiza la ruta. Si se encuentra una definición de ruta coincidente, Gin invoca los middleware (son opcionales) en un orden definido por la definición de ruta (si es que tienen) y el handler de ruta.

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 2 middlewares por defecto: logger and recovery middleware.

```
{}
// Creates a gin router with default middleware
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.

{}

```
// A handler for GET request on /hello-world
router.GET("/hello-world", func(c *gin.Context) {

    c.JSON(200, gin.H{
        "message": "Hello World!",
    }) // gin.H is a shortcut for map[string]interface{}
})
```

6. Correr nuestro servidor

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

```
{}  router.Run() // listen and serve on port 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() {
   // Creates a gin router with default middleware
   router := gin.Default()

   // A handler for GET request on /example
   router.GET("/hello-world", func(c *gin.Context) {
       c.JSON(200, gin.H{
           "message": "Hello World!",
       }) // gin.H is a shortcut for map[string]interface{}
   })
   router.Run() // listen and serve on port 8080
}
```

Gracias.

IT BOARDING

BOOTCAMP

Autor: Benjamin Berger
Email: benjamin@digitalhouse.com
Última fecha de actualización: 21-06-21

IT BOARDING
BOOTCAMP