



# INTRODUCCIÓN A LA CLASE

GO WEB

# ¿Qué aprenderemos hoy?

En la clase de hoy aprenderemos conceptos que nos permitirán poder desarrollar una API.

Los temas que veremos hoy son:

- Que es una API y para qué sirve
- Protocolos de comunicación (HTTP)
- Como esta compuesta una arquitectura web
- Qué es JSON - Funcionalidad y características
- Nuestro primer proyecto utilizando Gin
- Conceptos y características de Gin





# INTRODUCCIÓN A API

GO WEB

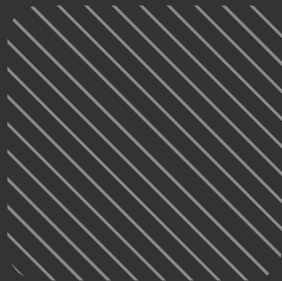


# API

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

IT BOARDING

**BOOTCAMP**



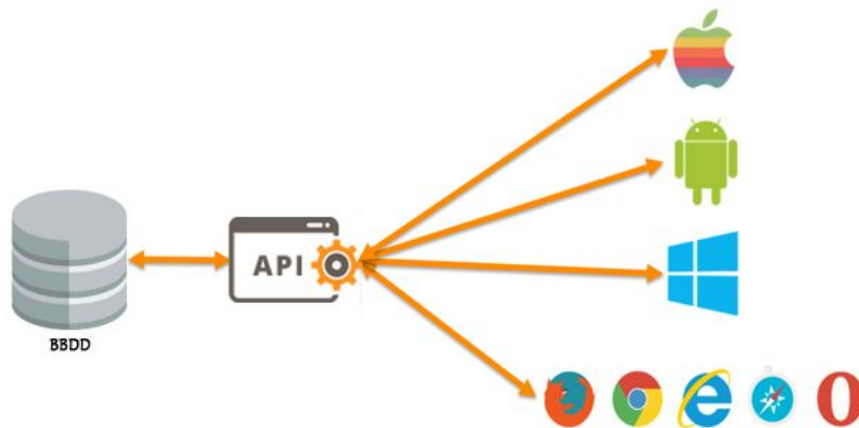


# ¿Qué es un API?

Una API es un conjunto de definiciones y protocolos que se utilizan para desarrollar e integrar el software de las aplicaciones.

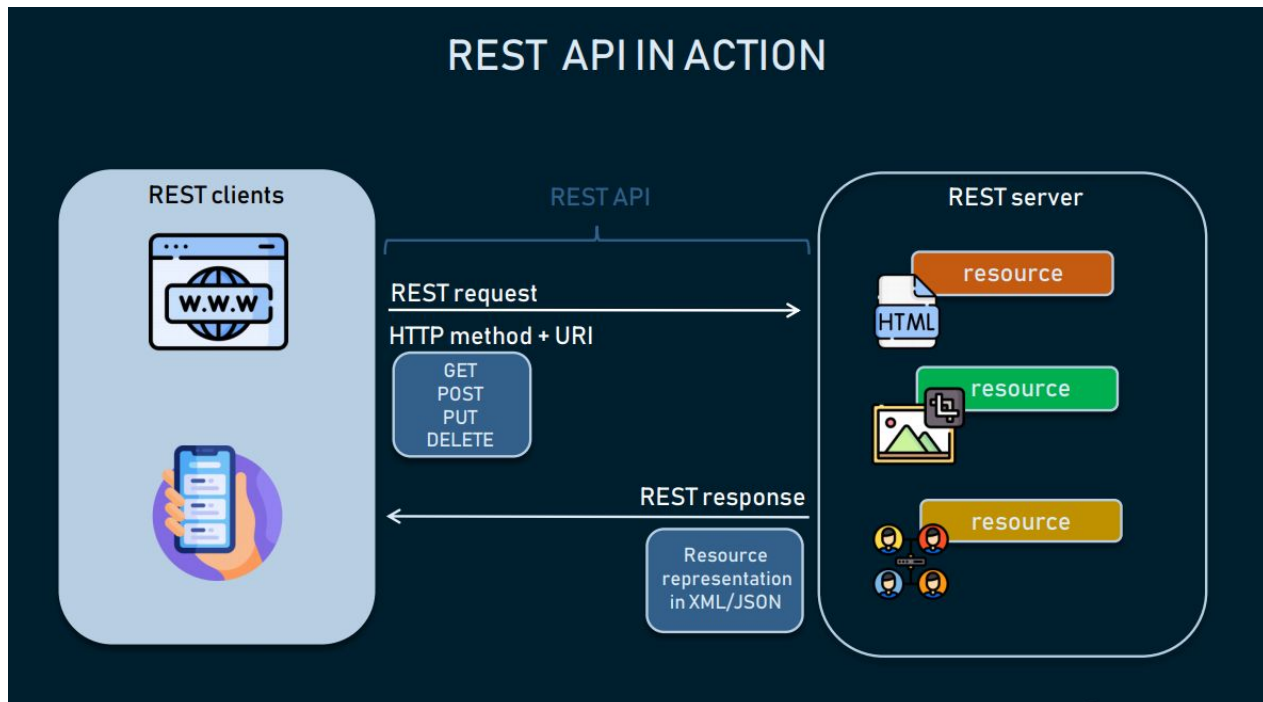
API significa interfaz de programación de aplicaciones.

No son la parte visible, sino los circuitos internos que sólo los desarrolladores ven y conectan para hacer funcionar una herramienta.





# // Un poco más de API





# ¿Para qué sirve una API?

Una de las principales funciones de las API es poder facilitar el trabajo a los desarrolladores y ahorrarles tiempo y dinero.

Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Otorgan flexibilidad; simplifican el diseño, la administración y el uso de las aplicaciones, y proporcionan oportunidades de innovación.

Las API (públicas) le permiten a empresas habilitar el acceso a sus recursos y, al mismo tiempo, mantener la seguridad y el control

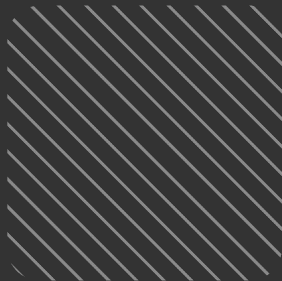


# HTTP

// 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.”**

IT BOARDING

**BOOTCAMP**

## // ¿Qué es un protocolo?

“Un protocolo puede ser un documento o una normativa que establece cómo se debe actuar en ciertos procedimientos. De este modo, recopila conductas, acciones y técnicas que se consideran adecuadas ante ciertas situaciones.”

IT BOARDING

BOOTCAMP

*Para*

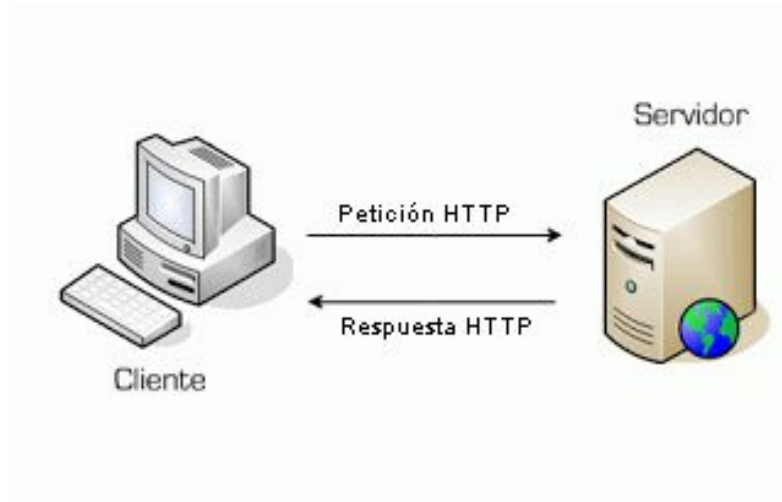
*más*

*información:*

<https://developer.mozilla.org/en-US/docs/Web/HTTP>

# Más sobre HTTP

Las comunicaciones de internet funcionan a través de protocolos. HTTP es el protocolo que utilizan los navegadores para comunicarse con los servidores web, aunque también es utilizado para el intercambio de información entre distintas aplicaciones de todo tipo.



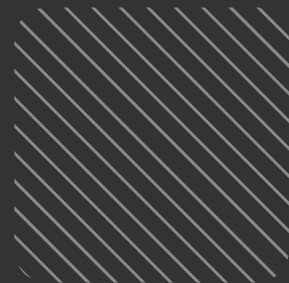


# ARQUITECTURA WEB

// Seis claves según Roy Fielding

IT BOARDING

**BOOTCAMP**





# Arquitectura Web

En 1993, Roy Fielding, cofundador 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.

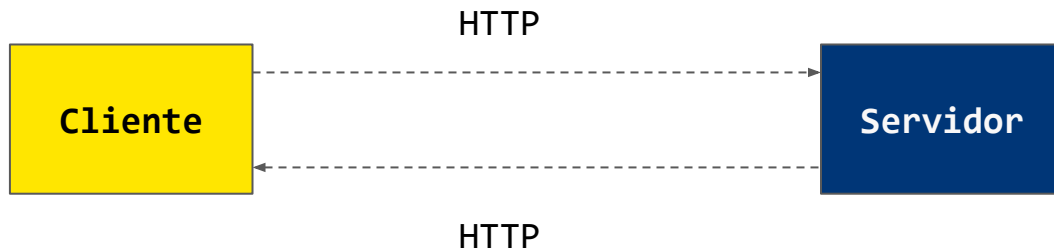
Las claves, que Fielding agrupó en seis categorías y llamó Arquitectura Web, son:

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.
- Lo más común es que los servicios REST estén implementados sobre 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.



La interfaz uniforme simplifica y desacopla la arquitectura, lo que permite que cada parte evolucione de forma independiente



# 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.

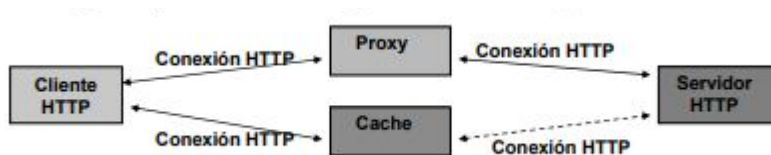




# ¿Qué es el Cache?

El caché web es un recurso utilizado por navegadores y servidores para reducir la cantidad de datos utilizados al cargar una página web.

El término “*caché*” hace referencia a un recurso usado como almacenaje de datos de acceso rápido y limitado, para que no tengan que ser buscados en memorias más lentas.





# ¿Para qué sirve el 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.

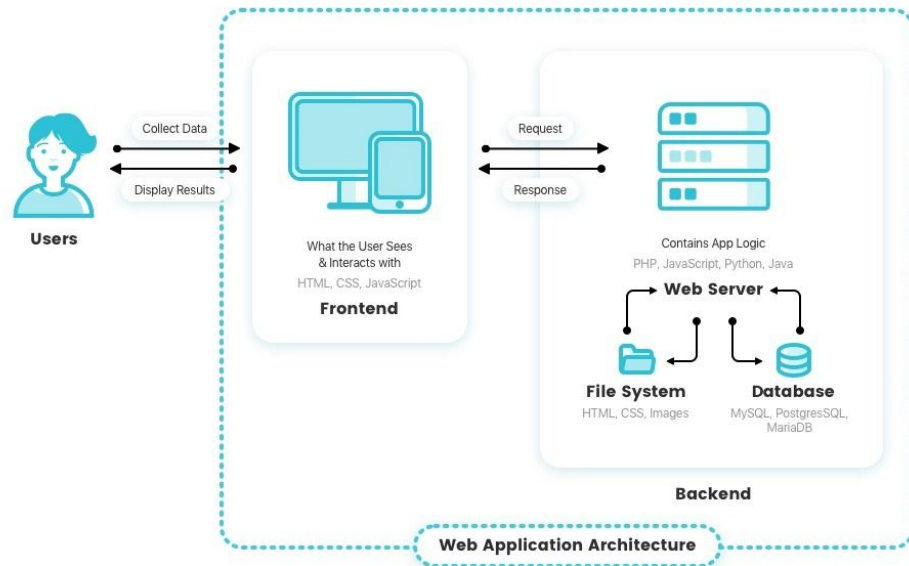
Algunas características son:

- Diferentes servidores entregan diferente información al mismo tiempo
- No se mantienen almacenadas las peticiones previas
- Las peticiones son independientes, no dependen de resultados previos

# 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 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

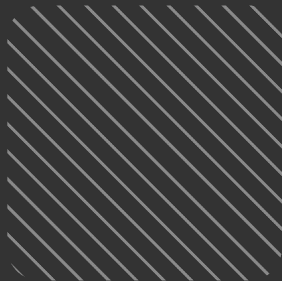
GO WEB

# ¿Qué es JSON?

// Concepto y características

IT BOARDING

**BOOTCAMP**



“**JSON** es un formato de intercambio de datos conveniente por su fácil **interoperabilidad** con otros lenguajes de programación y su **legibilidad** para los seres humanos.”

IT BOARDING

**BOOTCAMP**



## // Definición:

“**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 un formato para el intercambio 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 lenguajes de programación



# 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 constituido 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).



# Tipos de datos en JSON

**Números:** Permite números negativos y números con coma

**Cadenas:** Secuencia de caracteres. Ejemplo: *“Hola Estudiantes”*

**Booleanos:** Valores booleanos *true* y *false*

**null:** Representan el valor nulo.

**Array:** Lista de cero o más valores los cuales pueden ser de cualquier tipo. Los valores se separan por comas y el vector se mete entre corchetes. Ejemplo  
["Backend","Go","Digital House"]



## 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 #2

Veamos ejemplos.

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

JSON

```
{  
}
```



## Sintaxis de un JSON #3

JSON

```
{  
  "cadena": "mi cadena",  
  "entero": 1,  
  "flotante": 2.59  
  "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”.

IT BOARDING

**BOOTCAMP**



## 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 que XML.
- Velocidad de procesamiento alta.



## Desventajas:

- Algunos desarrolladores encuentran su escueta notación algo confusa.
- No soporta grandes cargas, solo datos comunes.
- Para la seguridad se requiere de mecanismos externos como expresiones regulares.

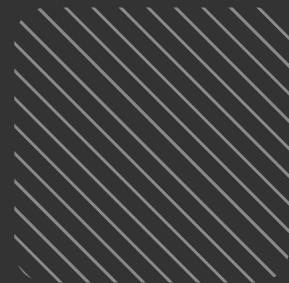


# JSON vs XML

// 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



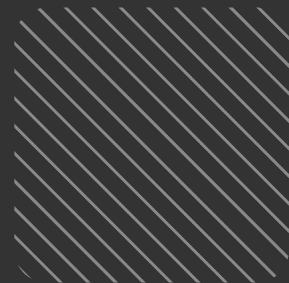


# Marshal

// 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, empezar en mayúsculas.

{}

```
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))
```





{}

```
type product struct {
    Name      string
    Price     float32
    Published bool
}

notebook := product{
    Name:      "MacBook Pro",
    Price:     1399.99,
    Published: true,
}

jsonData, err := json.Marshal(notebook)
if err != nil {
    log.Fatal("error: ", err)
}
fmt.Printf("Producto: \n %s \n", 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**.



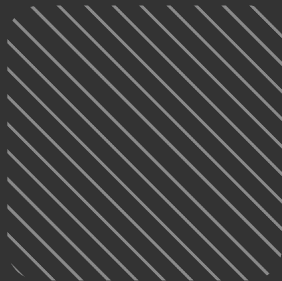


# Unmarshal

// 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 **unmarshal()** va a tratar de decodificarlo y llenar el struct con esos datos.

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



```
jsonData = []byte(`{"Name": "MacBook Air", "Price": 900,  
"Published": true}`)
```

```
var p product
```

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

```
fmt.Println("Product: ", 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.



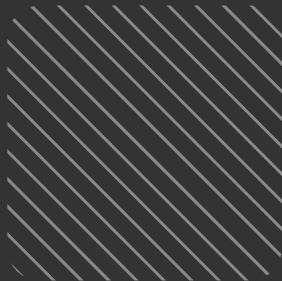


# NewEncoder

// Función

IT BOARDING

**BOOTCAMP**





## .NewEncoder( )

Así como podemos codificar structs a JSON usando la función `Marshal( )`, 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.



## .NewEncoder( )

```
{  
  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)  
}
```

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.

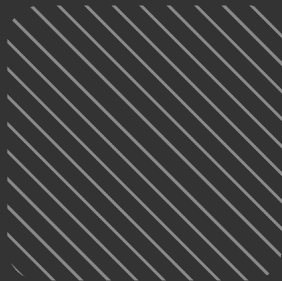


# NewDecoder

// 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.

## .NewDecoder( )

```

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)

```

En el ejemplo anterior modificamos el ejemplo de Unmarshal 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 "holaHandler".

```
{}  
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 toma una función como argumento.

Finalmente, llamamos a `ListenAndServe` con el puerto “:8080” y un controlador. `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

```
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)
}
```





# 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/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 iniciar un módulo, utiliza este comando:

**Dominio**

**Nombre del módulo**

```
$ go mod init github.com/usuario/web-server
```

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





# 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.

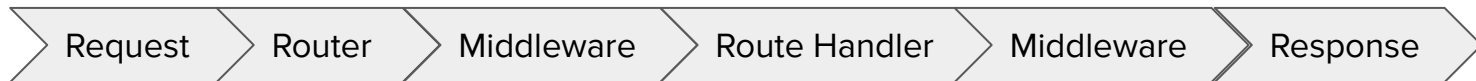
Gin es un micro-framework de alto rendimiento que se puede utilizar para crear aplicaciones web y microservicios en Go.

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.

```
{}  
    // 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 ultimo, 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

```
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

```
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)  
    }  
}
```

vs

## Gin-gonic

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



Luego de ver ejemplos de http y gin, podemos observar la practicidad que nos brinda gin con respecto al package http.

¡Recordemos que el package http es utilizado en Gin!



# Gracias.

IT BOARDING

**BOOTCAMP**





Autor: Benjamin Berger

Email: [benjamin@digitalhouse.com](mailto:benjamin@digitalhouse.com)

Última fecha de actualización: 12-10-21

IT BOARDING

**BOOTCAMP**

