



INTRODUCCIÓN A LA CLASE

GO WEB

Objetivos de esta clase

- Comprender las partes que componen un mensaje HTTP
- Conocer y diferenciar los mensajes que intercambian cliente y servidor.
- Comprender qué son los Routers.
- Obtener parámetros a partir de un Request.
- Conocer cómo filtrar contenido a partir de los parámetros de un request.
- Aplicar en la práctica el concepto de filtros.





MENSAJES HTTP

GO WEB



Fundamentos

// ¿Qué es el contexto? ¿Cómo se compone?

IT BOARDING

BOOTCAMP



// ¿Qué es un Context?

Es entre otras muchas cosas, un método para mover información entre una cadena de llamadas.

// ¿Y qué es un

El contexto, como un tiempo de espera, una fecha límite o un canal, señala un cierre y el retorno de las llamadas.

// ¿Cuándo necesitamos un contexto?

Como sugiere el nombre, usamos el paquete de contexto siempre que queremos pasar "contexto" o datos de ámbito común dentro de nuestra aplicación. Por ejemplo:

- Solicitar ID para llamadas a funciones y goroutines que forman parte de una llamada de solicitud HTTP
- Errores al recuperar datos de una base de datos
- Señales de cancelación al realizar operaciones asíncronas utilizando goroutines

¿Qué partes intervienen en un mensaje HTTP?

La comunicación en nuestro modelo cliente servidor intercambian dos tipos de mensajes:

- **Request.**
- **Response.**

El primero corresponde a aquellas acciones que el cliente quiere que haga el servidor, mientras que el segundo es el resultado de dichas acciones que el servidor devuelve al cliente.



Mensaje Request [1/6]

Un mensaje request posee la siguiente estructura:

REQUEST	
Method	URL
Version	Header
Body	



Mensaje Request [2/6]

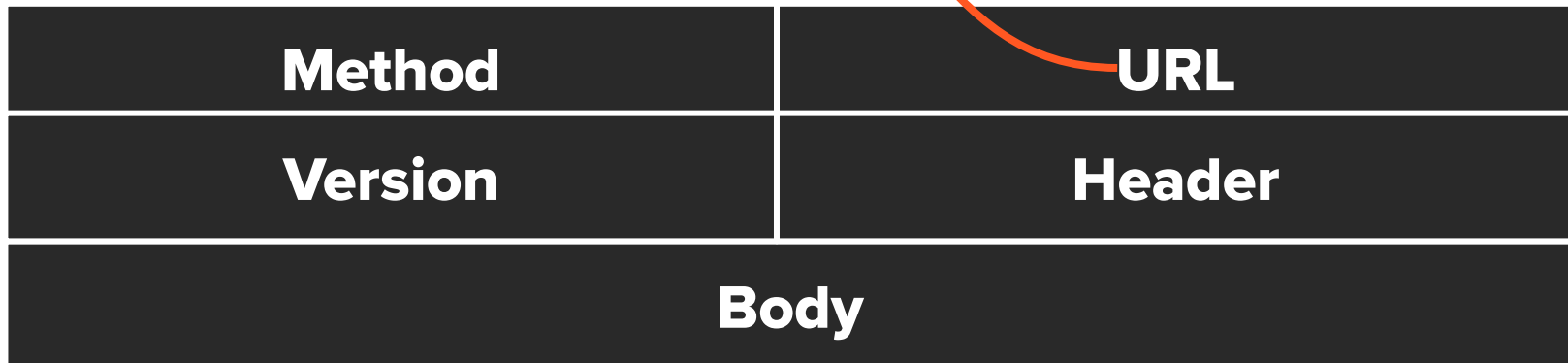
Este campo contiene el método que el cliente envía al servidor (GET/POST/PUT/DELETE).

Method	URL
Version	Header
Body	



Mensaje Request [3/6]

Este campo contiene el nombre de la URL o *path* a la cual el cliente quiere llevar a cabo la petición.



Method	URL
Version	Header
Body	

Mensaje Request [4/6]

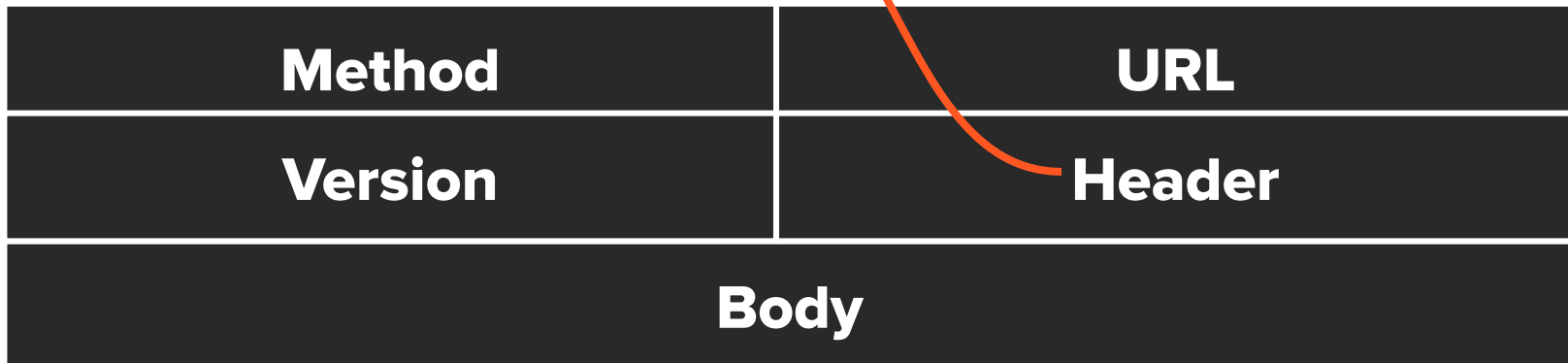
Aquí va la versión del protocolo HTTP que se usa en la comunicación, por ejemplo, HTTP / 1.1

Method	URL
Version	Header
Body	



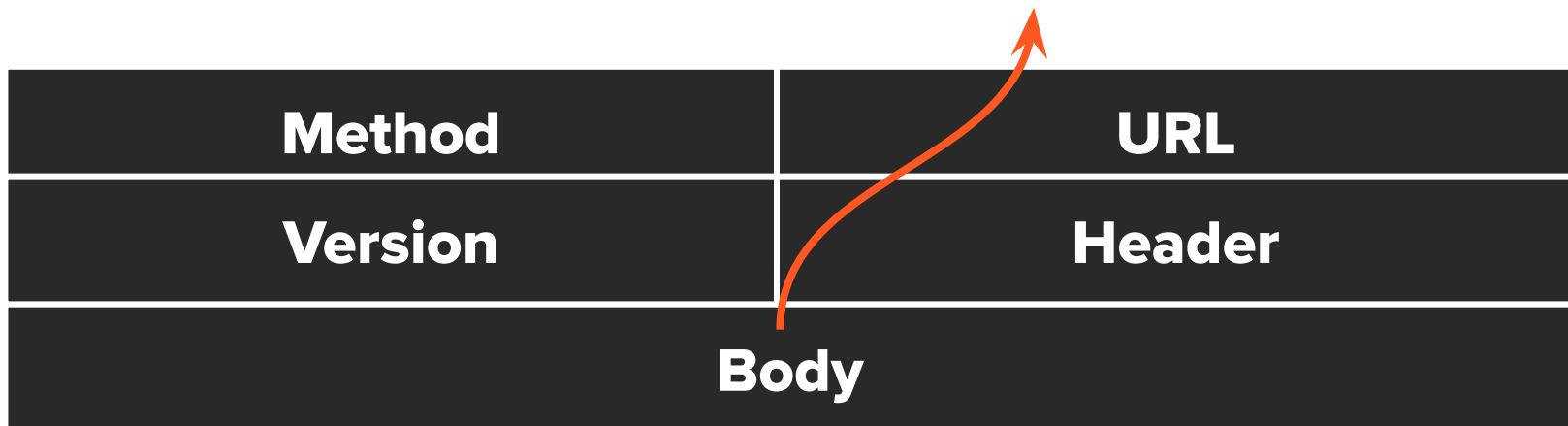
Mensaje Request [5/6]

El header contiene los **metadatos**, es decir, información sobre cookies, sesiones, etc. Puede ser opcional y los valores se separan por :



Mensaje Request [6/6]

Es un bloque de datos arbitrarios,
en formato de texto plano o JSON.
Es totalmente opcional.



Mensaje Response [1/3]

Mientras que, un mensaje response tiene la siguiente estructura:

RESPONSE	
Version	Status
Headers	
Body	

Mensaje Response [2/3]

Vemos que la estructura es similar, cambia el orden y añade un campo que es la frase textual. ¿Qué es?

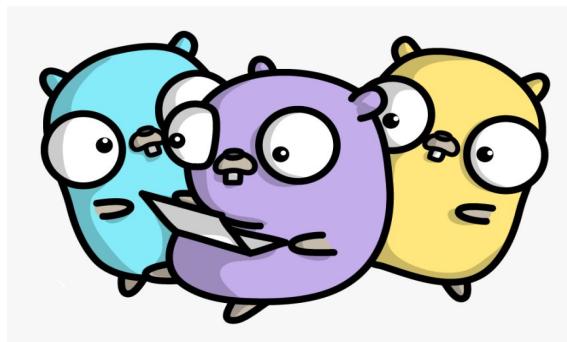
Cuando el servidor da una respuesta al cliente, lo hace por medio de códigos ya establecidos, estos se conocen como **códigos HTTP**.

Veamos algunos ejemplos de códigos y sus frases:

Mensaje Response [3/3]

Código	Descripción
200	OK
401	UNAUTHORIZED
404	NOT FOUND
500	INTERNAL ERROR
502	BAD GATEWAY

Ejemplo ilustrado [1/5]

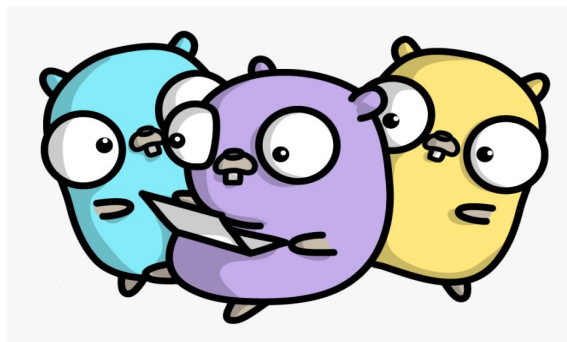


Cliente



www.gophers.com

Ejemplo ilustrado [2/5]



Cliente



GET /hi/greetings.txt HTTP/1.1

Accept: text/*

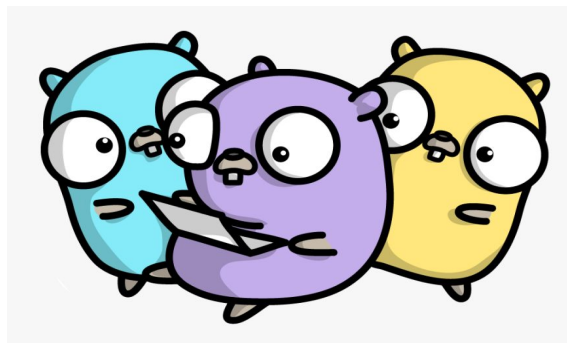
Host: www.gophers.com



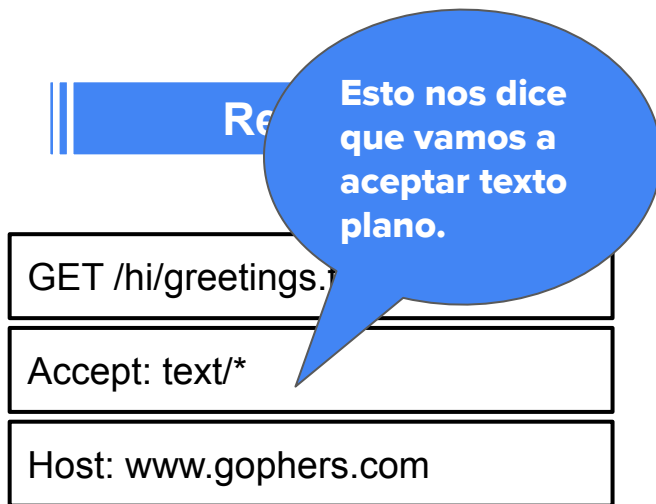
www.gophers.com



Ejemplo ilustrado [3/5]

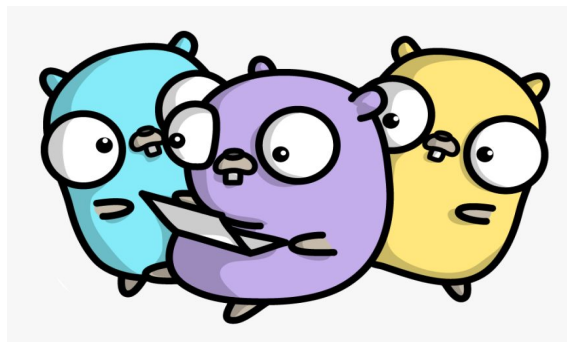


Cliente

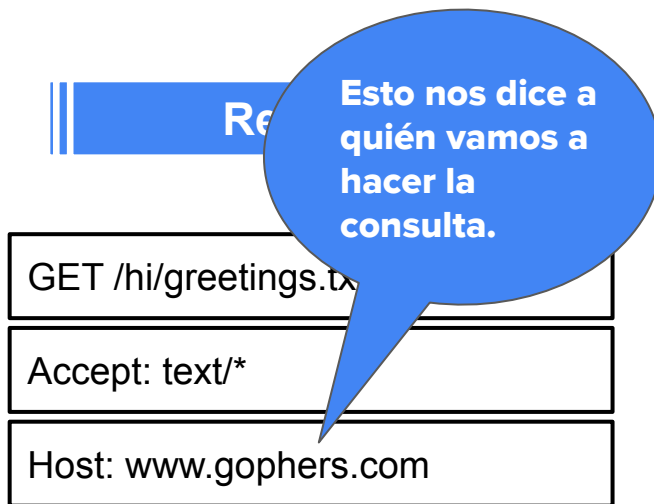


www.gophers.com

Ejemplo ilustrado [4/5]

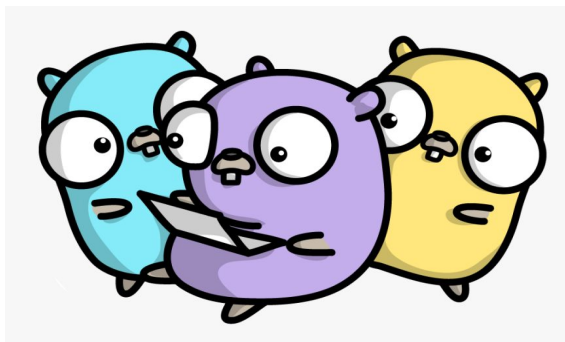


Cliente



www.gophers.com

Ejemplo ilustrado [5/5]



Cliente



HTTP/1.0	200	OK
Content type: /text/plain		
Content length: 19		
Body: Hi gophers !		



www.gophers.com



// Para concluir

Una solicitud HTTP se compone de los mensajes de request y response que intercambian el cliente y servidor. Estos forman el estado de la comunicación.

¡Continuemos aprendiendo!

IT BOARDING

BOOTCAMP

Gin Context

IT BOARDING

BOOTCAMP



¿Qué es gin.context?

Gin context es la parte más importante de este framework. Nos permite pasar variables entre middleware, así como los headers, parámetros, query string parameters, method entre otras variables del request.

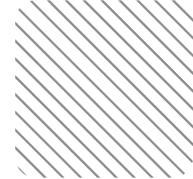
La misma se encarga de validar el JSON de una solicitud y generar una respuesta JSON.

Gin framework usa parte del package nativo de Golang para manejar peticiones, así como `http.Request` y `ResponseWriter` ¡Si, son los mismos del package `net/http`!, pero le agrega funcionalidades.

Es importante comprender que `gin.Context` es una estructura de Gin, si generamos una aplicación donde no utilizemos Gin, debemos utilizar el contexto de Go



Ejemplo



{}

```
//Esta función nos permite ver la anatomía de un mensaje Request de una
func(context *gin.Context) {
    //El body, header y method están contenidos en el contexto de gin.
    body := context.Request.Body
    header := context.Request.Header
    metodo := context.Request.Method

    fmt.Println("¡He recibido algo!")
    fmt.Printf("\tMetodo: %s\n", metodo)
    fmt.Printf("\tContenido del header:\n")

    for key, value := range header {
        fmt.Printf("\t\t%s -> %s\n", key, value)
    }
    fmt.Printf("\tEl body es un io.ReadCloser:(%s), y para trabajar con el vamos a tener que
        leerlo luego\n", body)
    fmt.Println("¡Yay!")
    context.String(200, "¡Lo recibí!") //Respondemos al cliente con 200 OK y un mensaje.
}
```





ROUTER

GO WEB

¿Que es un Router?

Un Router es una de las características principales que proporcionan todos los frameworks web. Nos permite definir varias rutas y luego decidir dónde es procesada cada una.

Todas las páginas web o endpoints de nuestras APIS se acceden mediante una URL.

Los Frameworks usan rutas para manejar solicitudes a estas URL.

Si una URL es `http://www.example.com/some/random/route`, la ruta será `/some/random/route`.

Trabajar con rutas nos permite tener más de un endpoint en una sola aplicación.



Default Router

Creando un default router, `gin.Default()` se crea un router de Gin con 2 middlewares por defecto: logger and recovery middleware.

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.

```
{}
```

```
func main() {  
    router := gin.Default()  
    //Cada vez que llamamos a GET y le pasamos una ruta, definimos un nuevo endpoint.  
    router.GET("/", HandlerRaiz)  
    router.GET("/gophers", HandlerGophers)  
    router.GET("/gophers/get", HandlerGetGopher)  
    router.GET("/gophers/info", HandlerGetInfo)  
    router.GET("/about", HandlerAbout)  
    router.Run(":8080")  
}
```



Agrupamiento de endpoint

En el ejemplo anterior vimos que usamos un mismo endpoint, que podemos decir general, y luego endpoints particulares... ¿Existe una forma de poder agruparlos?

Afortunadamente ¡Sí, la hay!

Gin provee una función que nos permite agrupar endpoints.

```
func main() {  
    server := gin.Default()  
    server.GET("/", HandlerRaiz)  
    //Ahora podemos atender peticiones a /gophers/, /gophers/get o /gophers/info de una  
    //forma más elegante.  
    gopher := server.Group("/gophers")  
    {  
        gopher.GET("/", HandlerGophers)  
        gopher.GET("/get", HandlerGetGopher)  
        gopher.GET("/info", HandlerGetInfo)  
    }  
    server.GET("/about", HandlerAbout)  
    server.Run(":8080")  
}
```

{ }





PARÁMETROS, FILTROS Y QUERIES

GO WEB

// ¿Qué es un parámetro?

“Algunos recursos, como por ejemplo, servicios de base de datos, pueden ser consultados de un determinado recurso a partir de una petición. Qué información se solicita, se logra mediante parámetros.”

Fundamentos

Una empresa posee una lista de empleados, y queremos consultar la información de empleados de acuerdo a su ID. Si dirigimos la consulta al path “/empleado/:ID” sería algo como el siguiente ejemplo:

<http://www.empresa.com/empleados/11>

<http://www.empresa.com/empleados/24>

Estamos recibiendo del cliente un dato como parámetro sin nombre, y le estamos dando un nombre “ID” al recibirlo en nuestra ruta.

En el ejemplo de arriba, al recibir el valor el “11” nuestro router lo convierte en el parámetro “ID” cuyo valor será “11”



Fundamentos

Ahora bien, los parámetros de ruta (*path*) nos resultan útiles hasta cierto punto. ¿Cómo podemos aclarar qué queremos una información que cumpla determinados requerimientos?

Supongamos que tenemos un servidor de una tienda virtual, y queremos obtener de nuestra colección de artículos aquel ítem cuyo código sea 12742

`http://www.mitienda.com/articulos?codigo=12742`

Vemos que se parece a una web como las que vimos en ejemplos anteriores, pero, aquí hay un detalle adicional: El signo de pregunta (?)



Fundamentos

Lo que denota el signo de pregunta (question mark) es la componente de **query** de la URL que se pasa al servidor o recurso de gateway.

Esta componente se acompaña del path de la URL identificando al recurso.

Toda consulta query se compone siempre de una **llave** y un **valor** (¡Sí, como un mapa!) y si deseamos enviar más de una llave/valor debemos separar los pares por &.

Ejemplo:

`http://www.mitienda.com/check-inventario.com?item=28&marca=Amaizing`



Ejemplo de path params

Ahora bien, vimos qué son las query, y qué son los paths. Veamos cómo podemos hacerlo funcionar con Gin. ¡Let's go!

```
{}
```

```
//Definimos una pseudobase de datos donde consultaremos la información.
var empleados = map[string]string{
    "644"    : "Empleado A",
    "755"    : "Empleado B",
    "777"    : "Empleado C",
}

func main() {
    server := gin.Default()
    server.GET("/", PaginaPrincipal)
    server.GET("/empleados/:id", BuscarEmpleado)
    server.Run(":8085")
}
```



(continuación) Ejemplo de path params

```
//Este handler se encargará de responder a /.
func PaginaPrincipal(ctx *gin.Context) {
    ctx.String(200, "¡Bienvenido a la Empresa Gophers!")
}

//Este handler verificará si la id que pasa el cliente existe en nuestra base de datos.
func BuscarEmpleado(ctx *gin.Context) {
    empleado, ok := empleados[ctx.Param("id")]
    if ok{
        ctx.String(200, "Información del empleado %s, nombre: %s", ctx.Param("id"), empleado)
    } else{
        ctx.String(404, "Información del empleado ¡No existe!")
    }
}
```

{}



¿Y si mi param no existe en la URL?

¿Cómo gin sabe cuáles son los parámetros que pasamos en una request? Aquí vemos cómo gin maneja los parámetros de una request. El método `Param` es un atajo a `Params.ByName(key)` y devuelve el valor del primer parámetro cuya clave coincide con el nombre dado. Si no se encuentra ningún parámetro coincidente, se devuelve una cadena vacía.

{ }

```
//Retorna el valor del "param" de la URL.  
//Es un atajo para c.Params.ByName(key)  
func (c *Context) Param(key string) string {  
    return c.Params.ByName(key)  
}  
  
//Definición de "Params"  
type Params []Param  
  
// Es un slice de parámetros dados por la URL.  
//Estos "Param" ocupan el mismo orden que en la URL.  
  
type Param struct{  
    Key string  
    Value string  
}
```



Ahora trabajemos con los query params

La forma más sencilla de acceder a aquellos parámetros que hayan sido provistos como query params (aquellos con la forma ?clave=valor), es con el método Query de nuestro context

```
{}
```

```
valorDeNuestraKey := ctx.Query("key")
```



Ejemplo [1/2]

Supongamos que tenemos una lista de empleados, y queremos saber aquellos que están o no activos, mediante un campo “Activo: “true”/”false””

¿Cómo podemos hacer esto?

Para hacerlo, hacemos uso de los **Query** de gin, que dada una key, nos dice su valor. Así, iteramos sobre la lista de empleados y obtenemos los activos e inactivos. ¡Veamos!





BODY

GO WEB

Leyendo el contenido del body

Veamos cómo leer el contenido del body enviado en la request HTTP (en aquellos métodos donde sea correcto enviarlo. Ej: POST, PUT, PATCH)

```
// Definimos nuestra estructura de información
// y agregamos las etiquetas correspondientes para poder realizar el unmarshalling
type Empleado struct {
    // Una etiqueta de struct se cierra con caracteres de acento grave `
    Nombre    string `form:"name" json:"name"`
    Password  string `form:"password" json:"password"`
    Id        string `form:"id" json:"id"`
    Activo    string `form:"active" json:"activa" binding:"required"`
}
```

{}



(continuación) Leyendo el contenido del body

{}

```
func AutorizarEmpleado(ctx *gin.Context) {  
  
    var empleado Empleado  
    // el metodo ShouldBindJSON de nuestro context, asociará el contenido del body  
    // a los campos de la estructura que le proveamos  
    if err := ctx.ShouldBindJSON(&empleado); err != nil {  
        ctx.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  
        return  
    }  
  
    if empleado.Nombre != "user1" || empleado.Password != "123" {  
        ctx.JSON(http.StatusUnauthorized, gin.H{"status": "no esta autorizado"})  
        return  
    }  
  
    ctx.JSON(http.StatusOK, gin.H{"status": "autorizado"})  
}
```



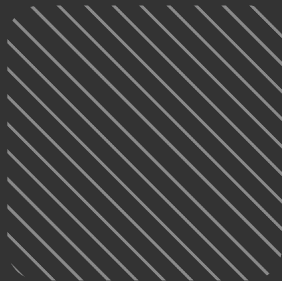


Filtros

// A partir de las Query, podemos filtrar información en nuestro servidor

IT BOARDING

BOOTCAMP



Ejemplo [2/2]

```
//Esta función solo mostrará aquellos empleados activos o  
inactivos, dependiente del parámetro active.  
func FiltrarEmpleados(ctxt *gin.Context) {  
    empleados := GenerarListaEmpleados()  
    var filtrados []*Empleado  
    for i, e := range empleados {  
        if ctxt.Query("active") == e.Activo {  
            filtrados = append(filtrados, &e)  
        }  
    }  
}
```





Gracias.

IT BOARDING

BOOTCAMP



Historial de Cambios

Fecha	Versión	Autor	Comentarios
18/06/2021	1.0.0	Gabriel Valenzuela	Creación de documento
21/06/2021	1.1.0	Gabriel Valenzuela	Corrección de errores y adición de temas*
27/06/2021	1.2.0	Gabriel Valenzuela	[2]

IT BOARDING

BOOTCAMP

