



INTRODUCCIÓN A LA CLASE

GO WEB

Objetivos de esta clase

- Comprender el concepto de web context.
- Conocer y diferenciar los mensajes que intercambian cliente y servidor.
- Comprender qué son los Routers.
- Obtener parámetros a partir de un Requests.
- Conocer cómo filtrar contenido a partir de los parámetros de un request.
- Aplicar en la práctica el concepto de filtros.





WEB CONTEXT

GO WEB



Fundamentos

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

IT BOARDING

BOOTCAMP



// ¿Qué es un Web Context?

“El contexto (HTTP) es la parte más importante de gin, y del request hacia nuestra API. Nos permite pasar variables entre middlewares, administrar el flujo, validar JSON, etc.”

IT BOARDING

BOOTCAMP

¿Cómo está formado el contexto web?[1/1]

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 [1/2]

Un mensaje request posee la siguiente estructura:

REQUEST	
Method	URL
Version	Header
Body	



Mensaje Request [2/2]

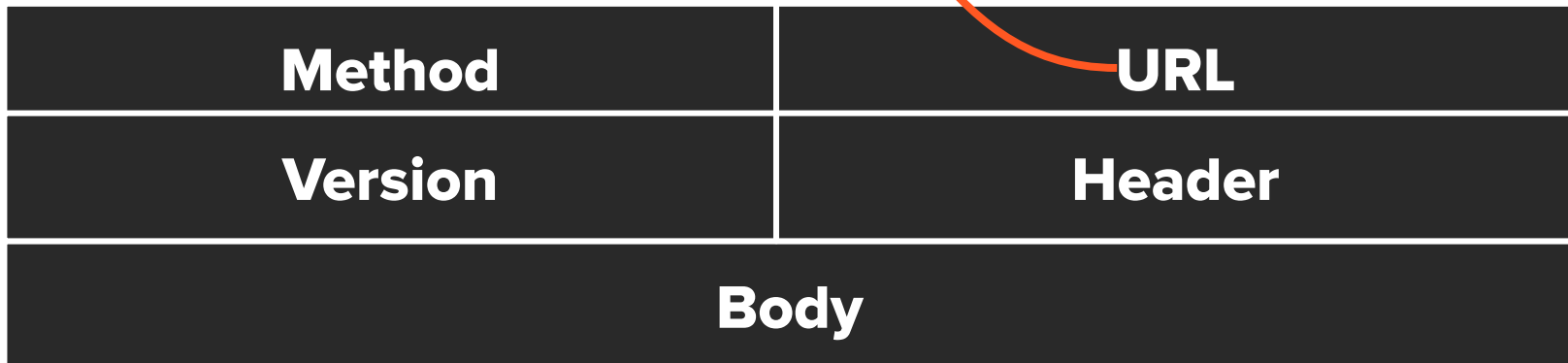
Este campo contiene el método que el cliente envía al servidor (GET/POST/PUT/DELETE).

Method	URL
Version	Header
Body	



Mensaje Request [2/2]

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 [2/2]

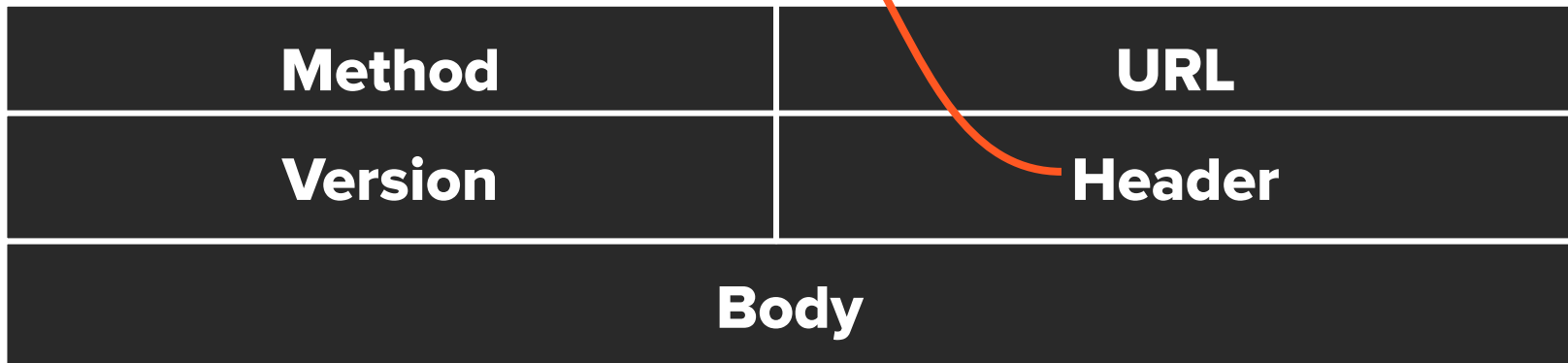
Aquí va la versión del protocolo HTTP que se usa en la comunicación, por ejemplo, HTTP / 1.1

Method	URL
Version	Header
Body	



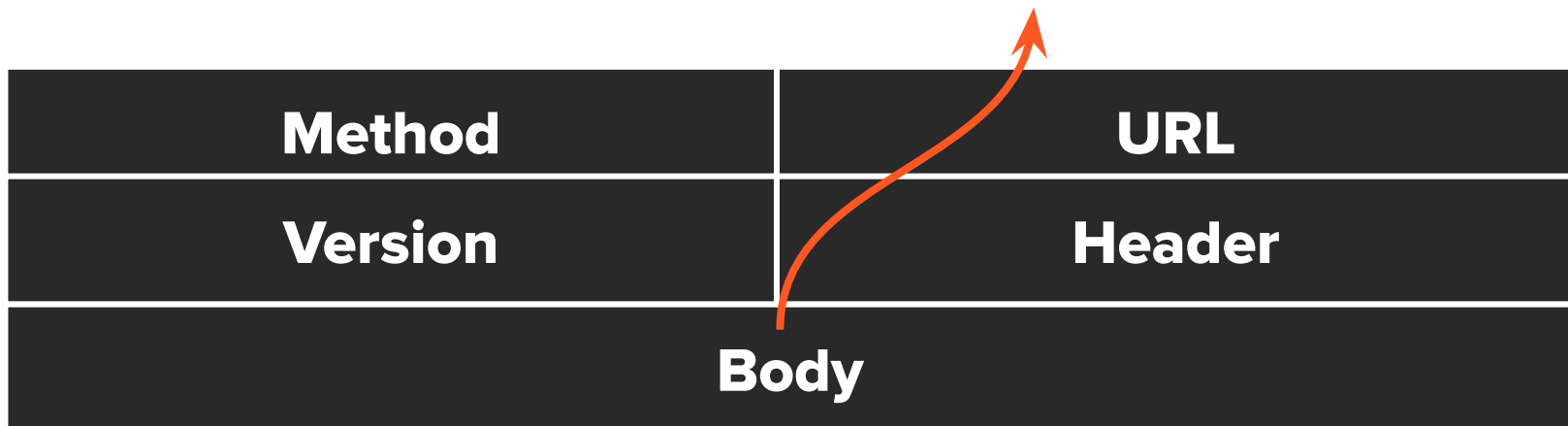
Mensaje Request [2/2]

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



Mensaje Request [2/2]

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



Mensaje Response [1/2]

Mientras, un mensaje response tiene la siguiente estructura:

RESPONSE	
Version	Status
Frase textual	Header
Body	

Mensaje Response [2/2]

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**.

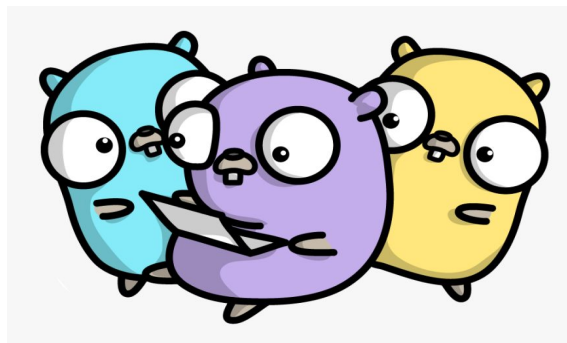
La frase textual es la descripción del código que acompaña al response.

Veamos algunos ejemplos de códigos y sus frases:

Mensaje Response [2/2]

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

Ejemplo ilustrado [1/2]



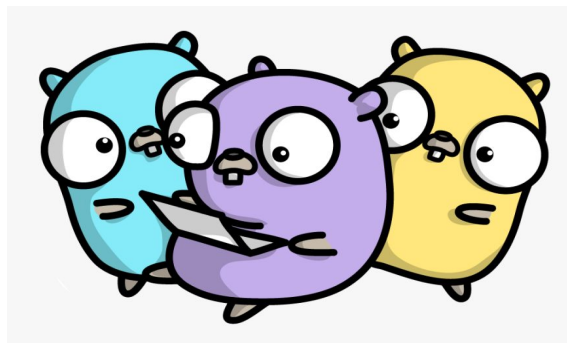
Cliente



www.gophers.com



Ejemplo ilustrado [2/2]



Cliente



GET /hi/greetings.txt HTTP/1.1

Accept: text/*

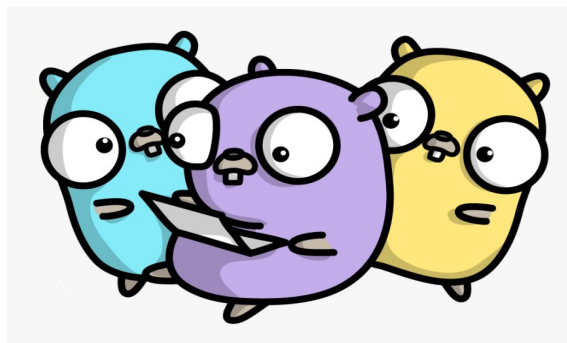
Host: www.gophers.com



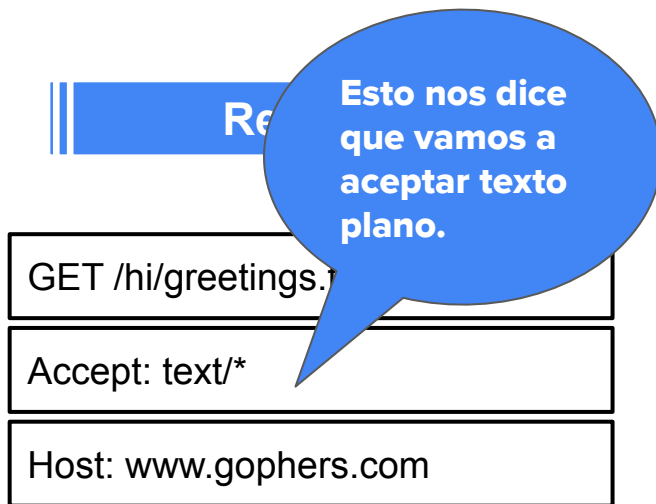
www.gophers.com



Ejemplo ilustrado [2/2]

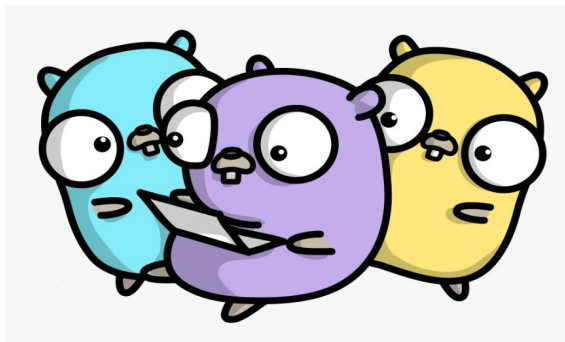


Cliente

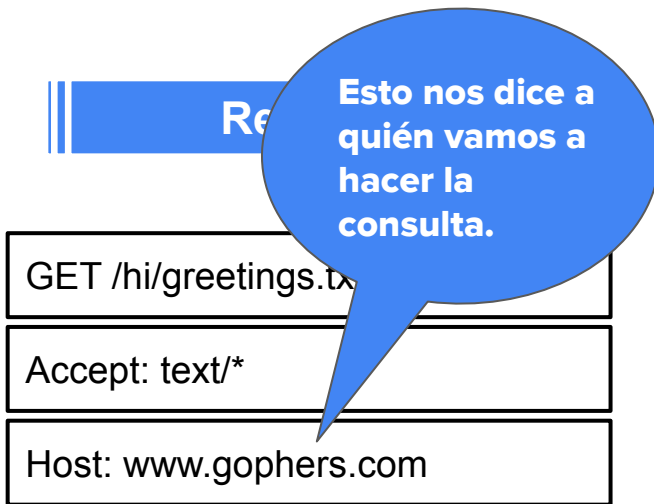


www.gophers.com

Ejemplo ilustrado [2/2]

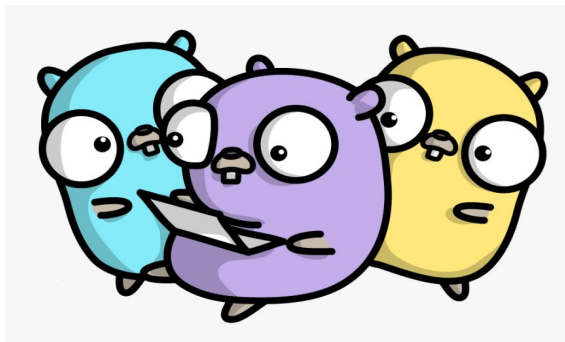


Cliente



www.gophers.com

Ejemplo ilustrado [2/2]



Cliente



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



www.gophers.com



// Para concluir

El contexto web 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

// ¿Qué es 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 y 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` ¡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 Ejemplo(context *gin.Context) {
    //El body, header y method están contenidos en el contexto de gin.
    contenido := 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:\n")

    for key, value := range header {
        fmt.Printf("\t\t%s -> %s\n", key, value)
    }
    fmt.Printf("\tCotenido:%s\n", contenido)
    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(":8081")  
}
```



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(":8081")  
}
```

{ }





PARÁMETROS & FILTROS

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/empleado/11>

<http://www.empresa.com/empleado/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, por lo que la siguiente URL puede ser usada para hacer una consulta (query) a un gateway de una base de datos para obtener información del producto cuya ID es 12742

`http://www.mitienda.com/check-inventario.com?item=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

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

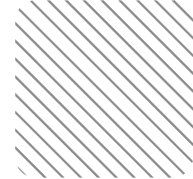
```
{}
```

```
//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("/", handlers.PaginaPrincipal)
    server.GET("/empleados/:id", handlers.BuscarEmpleado)
    server.Run(":8085")
}
```



Ejemplo



{ }

```
//Este handler se encargará de responder a /.
func PaginaPrincipal(ctxt *gin.Context) {
    ctxt.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(ctxt *gin.Context){
    empleado, existe := empleados[ctxt.Param("id")]
    if existe {
        ctxt.String(200,"Información del empleado %s, nombre: %s",ctxt.Param("id"),empleado)
    } else{
        ctxt.String(404,"Información del empleado ¡No existe!")
    }
}
```



Ejemplo [2/3]

¿Cómo gin sabe cuáles son los parámetros que pasamos en un request y no forman parte de la URL? En la documentación oficial, podemos ver cómo gin maneja los parámetros de un request. Nuestra intención no es aprender esto de memoria, sino entender cómo funciona para obtener el máximo beneficio.

{ }

```
//Retorna el valor del "param" de la URL.  
//Es un atajo para c.ParamsByName(key)  
func (c *Context) Param(key string) string {  
    return c.ParamsByName(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  
}
```



Ejemplo [3/3]

Veamos ahora cómo obtener un query y las funciones para obtener a partir de un JSON la información del query.

```
//Definimos nuestra estructura de información
type Empleado struct {
    // Una etiqueta de struct se cierra con caracteres de acento grave `
    Nombre string `form:"name" json:"name"`
    Id      string `form:"id" json:"id"`
    Activo  string `form:"active" json:"activa" bidding:"requiere"`
}
```

{ }



Ejemplo [3/3]

```
func BuscarEmpleado(ctxt *gin.Context) {  
    // ShouldBind verifica el Content-Type para seleccionar un  
    // mecanismo de biding (vinculación) de forma automática.  
  
    var empleado Empleado  
    //Nuestro objetivo aquí es asignar los campos de nuestra estructura con los datos que recibimos del request.  
    if ctxt.Bind(&empleado) == nil {  
        log.Println("==== Bind Por Query String =====")  
        log.Println(empleado.Id)  
        log.Println(empleado.Nombre)  
        ctxt.String(200, "(Query String) - Empleado: %s, Id: %s\n", empleado.Nombre, empleado.Id)  
    }  
    //Por query string como arriba (es decir, form) o por JSON.  
    if ctxt.BindJSON(&empleado) == nil {  
        log.Println("==== Bind Por JSON =====")  
        log.Println(empleado.Id)  
        log.Println(empleado.Nombre)  
        ctxt.String(200, "(Query JSON) - Empleado: %s, Id: %s\n", empleado.Nombre, empleado.Id)  
    }  
}
```

{ }



Filtros

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

IT BOARDING

BOOTCAMP



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!



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 []*Gopher
    cantidadFiltrados := 0
    for _, e := range empleados {
        if ctxt.Query("active") == e.Activo {
            filtrados = append(filtrados, e)
            cantidadFiltrados++
        }
    }
    if cantidadFiltrados > 0 {
        ctxt.String(200, "¡Filtrado exitoso!\n")
        for _, e := range filtrados {
            log.Println("===== Filter Por Query String =====")
            log.Println(e.Id)
            log.Println(e.Nombre)
            log.Println(e.Activo)
        }
    } else {
        ctxt.String(404, "¡Oh no! No se pudo hacer el filtrado\n")
    }
}
```

{ }





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

