GO

BOOTCAMP

Clase en vivo

//Go Web

IT BOARDING

BOOTCAMP

Objetivos de la clase:

- Conocer cómo manejar un método POST en Go.
- Implementar POST en grupos.
- Comprender que son los headers.
- Implementar tokens para validación en el header.

Índice

IT BOARDING

BOOTCAMP

1 | Repaso

IT BOARDING

BOOTCAMP

Postman

Para esta clase es requisito tener instalado Postman

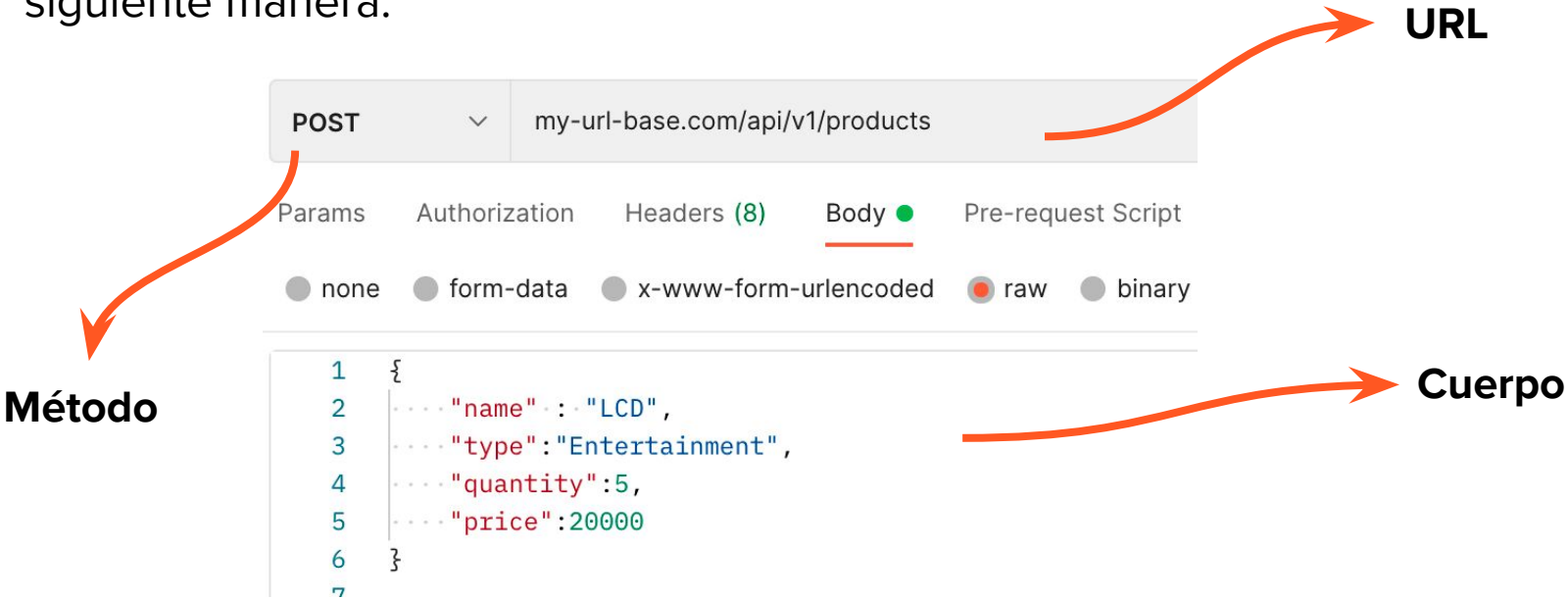
Método Post

// ¿Para qué se utiliza POST?

El método POST es utilizado para crear un nuevo registro, es decir, algo que no existía previamente.

Ejemplo

Una petición POST con un ejemplo para crear un producto podría ser de la siguiente manera:

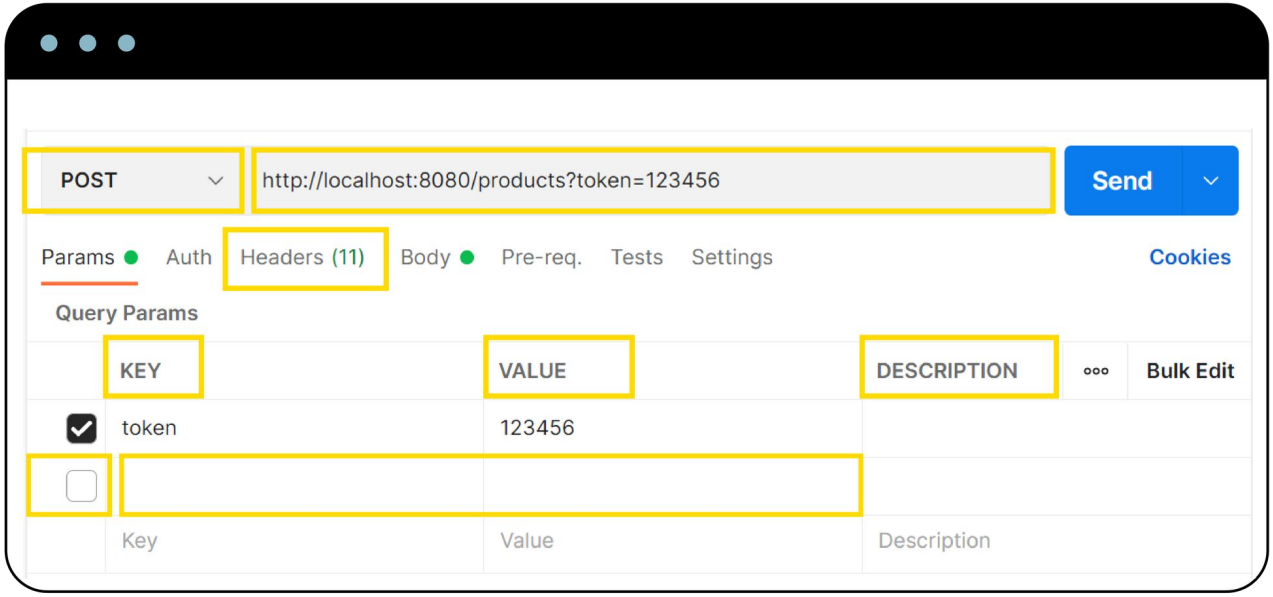


Headers

// ¿Qué es un Header?

Mediante las cabeceras (headers) podemos enviar información adicional junto con la petición, como por ejemplo, el tipo de contenido o un token de autenticación.

// ¿Qué es un header?



Body

// ¿Qué es un Body?

Mediante el body podemos enviar información adicional junto con la petición, por ejemplo, recursos para que nuestra API cumpla la lógica de negocio. Suele estar en formato JSON o texto.

// ¿Qué es un Body?

2 | POST en GO

IT BOARDING

BOOTCAMP

¿Cómo se manipula un POST?

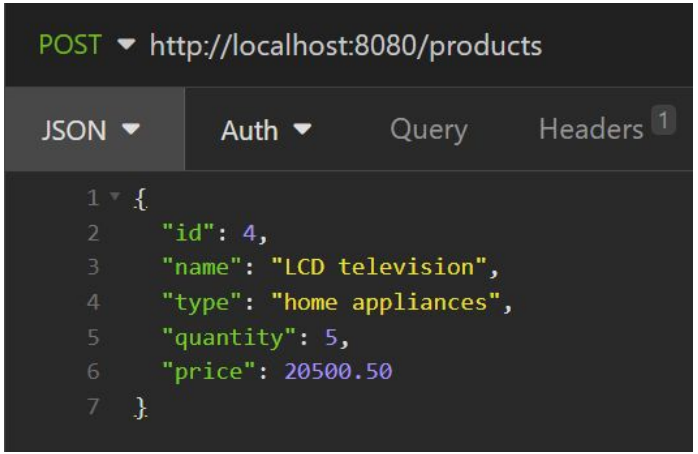
// ¿Cómo podemos recibir una petición POST?

Veremos un ejemplo de cómo levantar un servidor web en Go que reciba peticiones POST y las visualice.

Petición/Respuesta

Dentro de nuestra aplicación vamos a recibir en la petición un producto y devolveremos una respuesta con el producto, agregándole un id (clave). El campo ID se lo agregaremos en el código de nuestra aplicación.

Petición



Respuesta

Controller

Generamos una estructura con acceso a un **storage**. En este caso representado como un **map** de **Products**.

{}

```
// Product is a struct that contains the information of a product
type Product struct {
    Id       int
    Name     string
    Type     string
    Quantity int
    Price    float64
}
// ControllerProducts is a struct that contains the storage of products
type ControllerProducts struct {
    storage map[int]*Product
}
```

Handler #01 - Request Body

Generamos una estructura con los campos de la petición que recibiremos, para poder procesar el **Body Request**

{}

```
type RequestBodyProduct struct {
    Name     string

    Type     string

    Quantity int

    Price    float64
}
```

Handler #01 - Request Body

Utilizaremos la etiqueta **json**, para especificarle cuáles serán los campos que recibiremos de la petición.

{}

```
type RequestBodyProduct struct {
    Name     string  `json:"name"`

    Type     string  `json:"type"`

    Quantity int     `json:"quantity"`

    Price    float64 `json:"price"`
}
```

Handler #02 - Response Body

Creamos una representación del **Body Response**.

{}

```
type ResponseBodyProduct struct {
    Message string `json:"message"`
    Data    *struct {
        Id       int         `json:"id"`
        Name     string      `json:"name"`
        Type     string      `json:"type"`
        Quantity int         `json:"quantity"`
        Price    float64     `json:"price"`
    } `json:"data"`
    Error   bool    `json:"error"`
}
```

Handler #03 - http.HandlerFunc

Creamos el **HandlerFunc** encargado de manejar la petición y guardar el **Product**.
Aplicamos **decoding** sobre el **body request** hacia nuestro **struct**.

{}

```
func (c *ControllerProducts) Create() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // request
        var reqBody RequestBodyProduct
        if err := json.NewDecoder(r.Body).Decode(&reqBody); err != nil {
            code := http.StatusBadRequest
            body := &ResponseBodyProduct{
                Message: "Bad Request",
                Data: nil,
                Error: true,
            }

            w.WriteHeader(code)
            w.Header().Set("Content-Type", "application/json")
            json.NewEncoder(w).Encode(body)
        }
```

Handler #03 - http.HandlerFunc

Aplicamos **deserialization** desde el **RequestBody** hacia el **struct Product** y **guardamos** el producto.

{}

```
func (c *ControllerProducts) Create() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // process
        // -> deserialization
        pr := &Product{
            Id: len(c.storage) + 1,
            Name: reqBody.Name,
            Type: reqBody.Type,
            Quantity: reqBody.Quantity,
            Price: reqBody.Price,
        }
        // -> save product
        c.storage[pr.Id] = pr
```

Handler #03 - http.HandlerFunc

Retornamos el **Response**

{}

```
func (c *ControllerProducts) Create() http.HandlerFunc {
        // response
        code := http.StatusCreated
        body := &ResponseBodyProduct{
            Message: "Product created",
            Data: &struct {
                Id       int        `json:"id"`
                Name     string     `json:"name"`
                Type     string     `json:"type"`
                Quantity int        `json:"quantity"`
                Price    float64    `json:"price"`
            }{Id: pr.Id, Name: pr.Name, Type: pr.Type, Quantity: pr.Quantity, Price: pr.Price},
            Error: false,
        }

        w.WriteHeader(code); w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(body)
    }
}
```

Paquete Chi #01 - Dependencies

Inicializamos las dependencias del **Controller** con una **base de datos**.

{}

```
func main() {
    // dependencies
    db := make(map[int]*handlers.Product)
    ct := handlers.NewControllerProducts(db)

    // ...
```

Paquete Chi #02 - Router

Inicializamos el **router** de **Chi**, al cual registramos nuestro **endpoint** con método **POST**, el **handler** encargado de manejar la **creación** de un **Product**. Luego iniciamos el **server**.

{}

```
func main() {
    rt := chi.NewRouter()
    // -> routes
    rt.Post("/products", ct.Create())

    // -> run
    if err := http.ListenAndServe(":8080", rt); err != nil {
        panic(err)
    }
}
```

POST con Grupos

Definir Grupo Productos

También podemos definir **route groups** para tener el **registro** de routes **más organizado**.

{}

```
func main() {
    // dependencies
    db := make(map[int]*handlers.Product)
    ct := handlers.NewControllerProducts(db)

    // server
    rt := chi.NewRouter()
    // -> routes
    // -> -> products group
    rt.Route("/products", func(rt chi.Router) {
        // create
        rt.Post("/", ct.Create())
    })
```

3 | HEADERS en GO

IT BOARDING

BOOTCAMP

Headers en Go

Para recibir y procesar los headers se utiliza el package nativo de go, **http**, el **http.ResponseWriter** que nos provee del método **Header()** para obtener una copia y buscar información

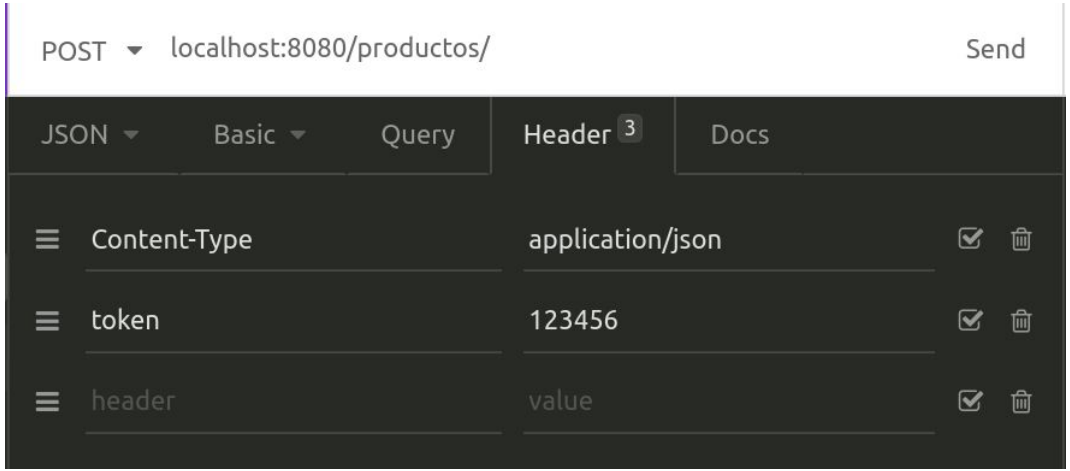
{}

```
func (c *ControllerProducts) Create() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // request
        header := w.Header().Get("My-Header")
```

De esta forma podemos recibir el valor que se haya enviado en la cabecera **My-Header**, siguiendo este formato de texto particular.

Enviar token

Vamos a enviar un token al momento de enviar el producto. Si el token es correcto nos responderá correctamente, en caso contrario nos devolverá un error de autentificación.



Recibir token

En la funcionalidad Guardar, lo primero que se hará es recibir el **token** que haya sido enviado en la petición:

{}

```
func (c *ControllerProducts) Create() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // request
        token := w.Header().Get("Token")

        // ...
```

Validar token

Validamos el token y en caso de no ser el que esperamos, retornamos un error de autentificación:

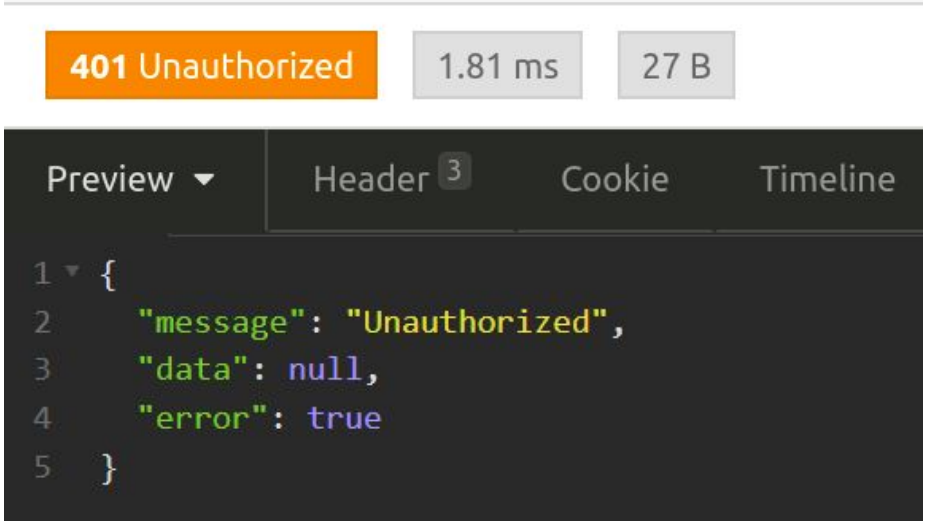
{}

```
func (c *ControllerProducts) Create() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // request
        token := w.Header().Get("Token")
        if token != "123456" {
            code := http.StatusUnauthorized // 401
            body := &ResponseBodyProduct{Message: "Unauthorized", Data: nil, Error: true}

            w.WriteHeader(code); w.Header().Set("Content-Type", "application/json")
            json.NewEncoder(w).Encode(body)
        }
        // ...
```

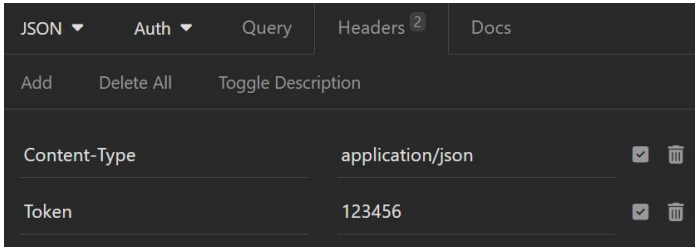
Error de autentificación

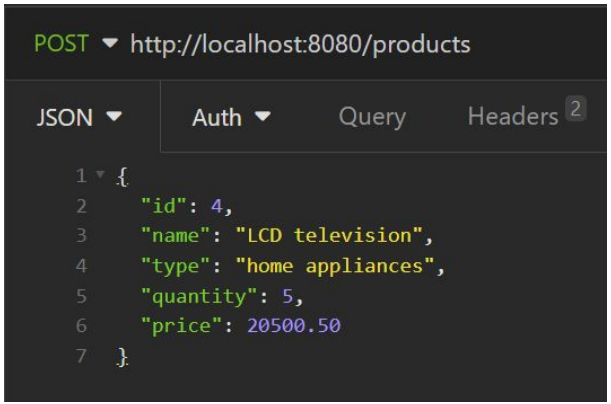
De esta manera, al no enviar el token o enviar un token inválido, la aplicación nos devolverá un error y no se realizará el proceso dándole más seguridad al servicio.



Enviar token válido

Al enviar el token válido, nos devuelve la respuesta correctamente.





A codear...

Conclusiones

Hoy aprendimos cómo implementar el método POST en una aplicación web.

Enviamos información por los headers de la petición y los manipulamos en nuestra aplicación.

Además vimos cómo almacenar datos en memoria para crear pequeñas aplicaciones que almacenan nuestros datos.

Actividad

Gracias.

IT BOARDING

BOOTCAMP