

# POST en Go

# Índice

- 01** [¿Cómo se manipula un POST?](#)
- 02** [POST en grupos](#)
- 03** [Guardar productos en memoria](#)
- 04** [Headers en Go](#)



01

¿Cómo se manipula  
un POST?

# Petición/Respuesta

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

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

POST ▼ localhost:8080/productos

```
JSON ▼ Auth ▼ Query Header 1
1 ▼ {
2   "nombre": "Televisor LCD",
3   "tipo": "electrodomesticos",
4   "cantidad": 5,
5   "precio": 20000
6 }
```

## Respuesta

```
1 ▼ {
2   "id": 4,
3   "nombre": "Televisor LCD",
4   "tipo": "electrodomesticos",
5   "cantidad": 5,
6   "precio": 20000
7 }
```

# Estructura request

Generamos una estructura con los campos de la petición que recibiremos, para poder procesarla.

```
{  
  type request struct {  
    ID      int  
    Nombre  string  
    Tipo    string  
    Cantidad int  
    Precio  float64  
  }  
}
```

# Etiqueta JSON

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

```
{  
  type request struct {  
    ID      int    `json:"id"`  
    Nombre  string `json:"nombre"`  
    Tipo    string `json:"tipo"`  
    Cantidad int    `json:"cantidad"`  
    Precio  float64 `json:"precio"`  
  }  
}
```

# Paquete Gin

Definiremos con **Gin** un servicio web mediante el método **POST**, el cual tendrá como path “**productos**”.

```
{  
  r := gin.Default()  
  r.POST("/productos", func(c *gin.Context) {  
  })  
  r.Run()  
}
```

# Recibir petición

Recibimos la petición y hacemos el traspaso de los datos a nuestra estructura con el método **ShouldBindJSON** (en caso de no poder hacer el traspaso, nos devolverá error).

```
{  
  r.POST("/productos", func(c *gin.Context) {  
    var req request  
    c.ShouldBindJSON(&req)  
  })  
}
```



# Aclaración: ShouldBind vs. Bind

Anteriormente, vimos la utilización de **Bind** en vez de **ShouldBind**, ¿en qué difieren?

## Bind

- Si hay un error, Gin aborta el request seteando el status en 400 y el Content-Type a text/plain; charset=utf-8.

## ShouldBind

- Si hay un error, Gin no hace nada y deja en el desarrollador la responsabilidad de manipularlo.

**¡ShouldBind nos permite un mayor control!**

# Validar error

Tomamos el error que nos devuelve el bind y realizamos una validación.

```
{  
  r.POST("/productos", func(c *gin.Context) {  
    var req request  
    if err := c.ShouldBindJSON(&req); err != nil {}  
  })  
}
```

# Retornar error

En caso de haber un error, lo retornamos. Utilizamos el método JSON para definir el código y el cuerpo del mensaje a retornar.

```
{}  
r.POST("/productos", func(c *gin.Context) {  
    var req request  
    if err := c.ShouldBindJSON(&req); err != nil {  
        c.JSON(400, gin.H{  
            "error": err.Error(),  
        })  
        return  
    }  
})
```

# Agregar ID

En caso de que la petición recibida sea correcta, agregamos un ID a nuestro producto.

```
{  
  {}  
  r.POST("/productos", func(c *gin.Context) {  
    var req request  
    if err := c.ShouldBindJSON(&req); err != nil {  
      c.JSON(400, gin.H{  
        "error": err.Error(),  
      })  
      return  
    }  
    req.ID = 4  
  })  
}
```

# Retornar producto

Enviamos el producto con el ID asignado como respuesta.

```
{  
  {}  
  r.POST("/productos", func(c *gin.Context) {  
    var req request  
    if err := c.ShouldBindJSON(&req); err != nil {  
      c.JSON(400, gin.H{  
        "error": err.Error(),  
      })  
      return  
    }  
    req.ID = 4  
    c.JSON(200, req)  
  })  
}
```

02

# POST en grupos

# Definir estructura

Definiremos nuestra estructura request fuera de la función main.

```
type request struct {  
    ID      int      `json:"id"`  
    Nombre  string    `json:"nombre"`  
    Tipo    string    `json:"tipo"`  
    Cantidad int      `json:"cantidad"`  
    Precio  float64   `json:"precio"`  
}  
  
func main() {  
  
}
```

# Definir grupo productos

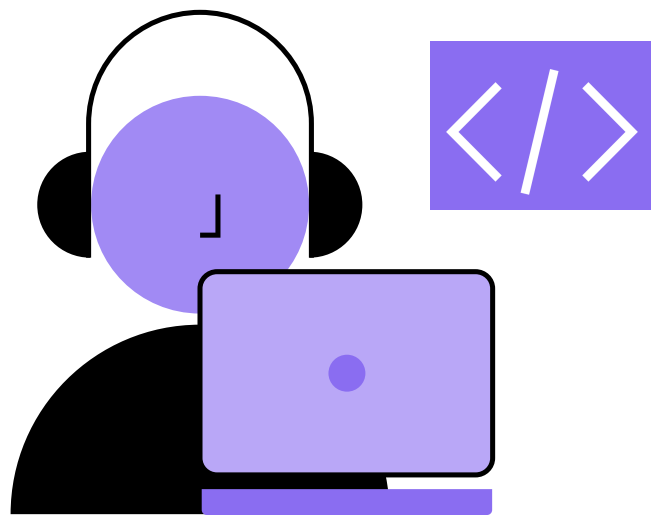
Creamos una agrupación para productos en el cual definiremos los diferentes **endpoints**. En nuestro caso, solo tendremos el endpoint “**Guardar**”.

```
{  
func main() {  
    r := gin.Default()  
    pr := r.Group("/productos")  
    pr.POST("/", Guardar())  
    r.Run()  
}
```



# Función Guardar

Por último, para implementar la funcionalidad de **Guardar**, necesitamos generar una función que nos devuelva otra función con el contexto de Gin por parámetro.



```
func Guardar() gin.HandlerFunc {  
    return func(c *gin.Context) {  
        var req request  
        if err := c.ShouldBindJSON(&req); err != nil {  
            c.JSON(404, gin.H{  
                "error": err.Error(),  
            })  
            return  
        }  
        req.ID = 4  
        c.JSON(200, req)  
    }  
}
```

03

# Guardar productos en memoria

# Guardar productos en memoria

Procederemos a guardar todos los productos enviados en memoria, siempre y cuando la petición sea correcta.  
Lo primero que haremos es declarar a nivel global:

- Una variable de productos donde se guardarán los productos que enviemos.
- Una variable que guarde y vaya incrementando el ID, para siempre tomar el máximo.

```
{  
    var products []request  
    var lastID int  
  
    func main() {
```

# Incrementar y asignar ID

En lugar de asignar un ID fijo, incrementaremos el ID en 1 y se lo asignaremos a nuestro producto.

```
{}  
  lastID++  
  req.ID = lastID
```

# Guardar producto

Por último, guardamos el producto con el ID asignado en memoria. De esta forma se irán guardando a medida que vayamos haciendo peticiones.

```
{  
    lastID++  
    req.ID = lastID  
  
    products = append(products, req)  
}
```

04

# Headers en Go

# Headers en Go

Para recibir y procesar los headers, se utiliza el contexto de Gin (**\*gin.Context**).

Se obtienen de la siguiente manera, al momento de recibir la petición:

```
{ } c.GetHeader("mi_header")
```

De esta forma podemos recibir el valor que se haya enviado en la cabecera **mi\_header**.

# 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 autenticación.

POST ▾ localhost:8080/productos/

Send

JSON ▾Basic ▾Query

Header <sup>3</sup>

Docs

≡	Content-Type	application/json	<input checked="" type="checkbox"/>	
≡	token	123456	<input checked="" type="checkbox"/>	
≡	header	value	<input checked="" type="checkbox"/>	



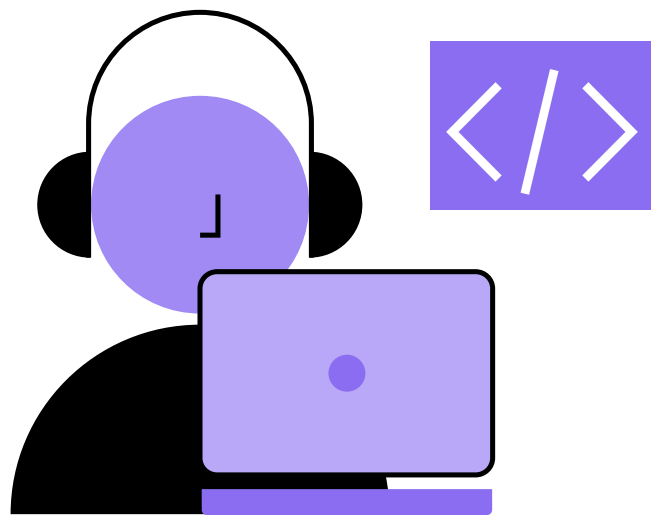
# Recibir token

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

```
{  
func Guardar() gin.HandlerFunc {  
    return func(c *gin.Context) {  
        token := c.GetHeader("token")  
        ...  
    }  
}
```

# Validar token

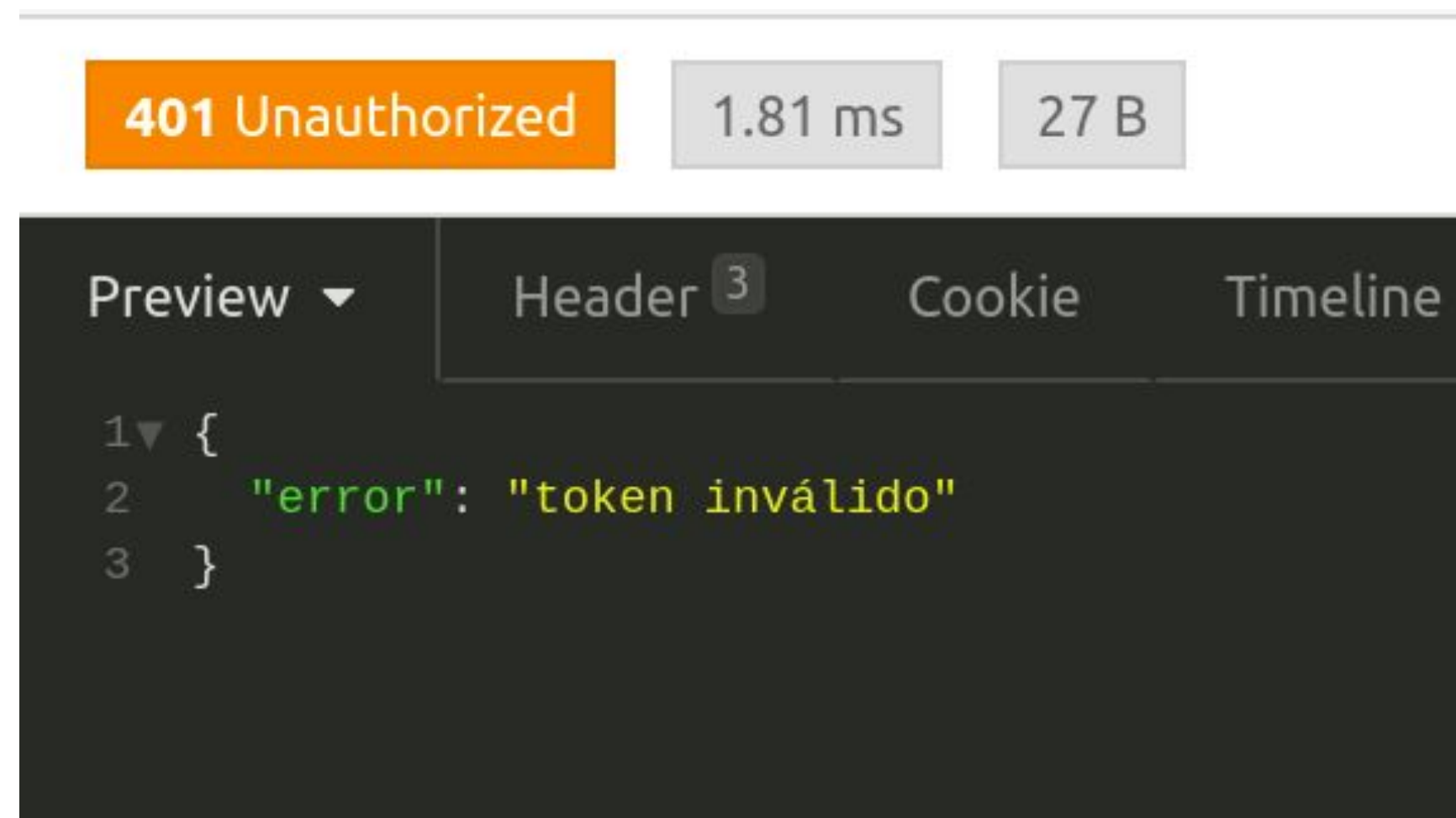
Validamos el token y, en caso de no ser el que esperamos, retornamos un error de autenticación:



```
func Guardar() gin.HandlerFunc {  
    return func(c *gin.Context) {  
        token := c.GetHeader("token")  
        if token != "123456" {  
            c.JSON(401, gin.H{  
                "error": "token inválido",  
            })  
            return  
        }  
        ...  
    }  
}
```

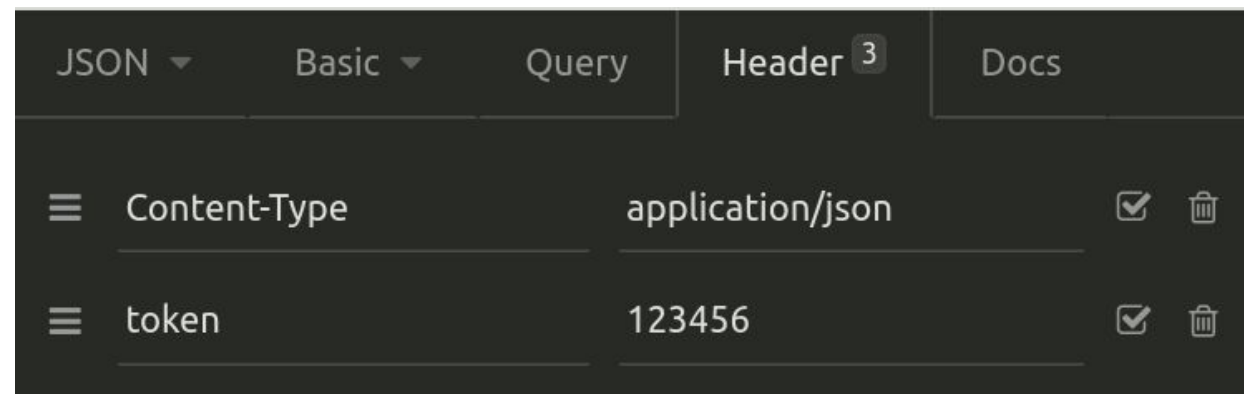
# Error de autenticación

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.

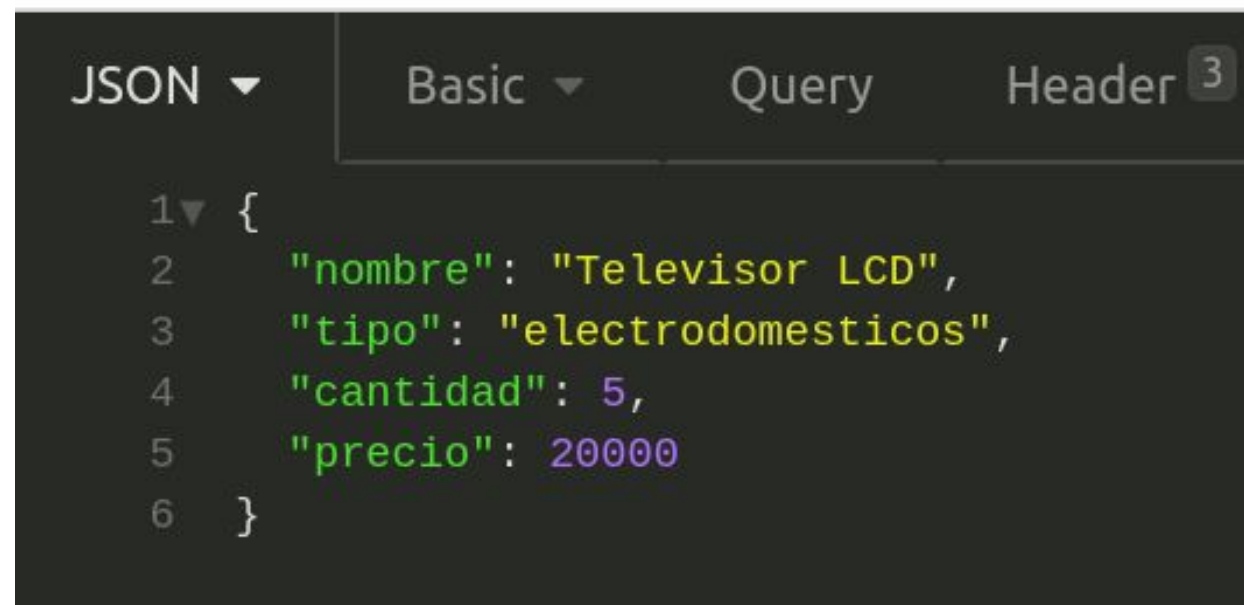


# Enviar token válido

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



POST localhost:8080/productos/

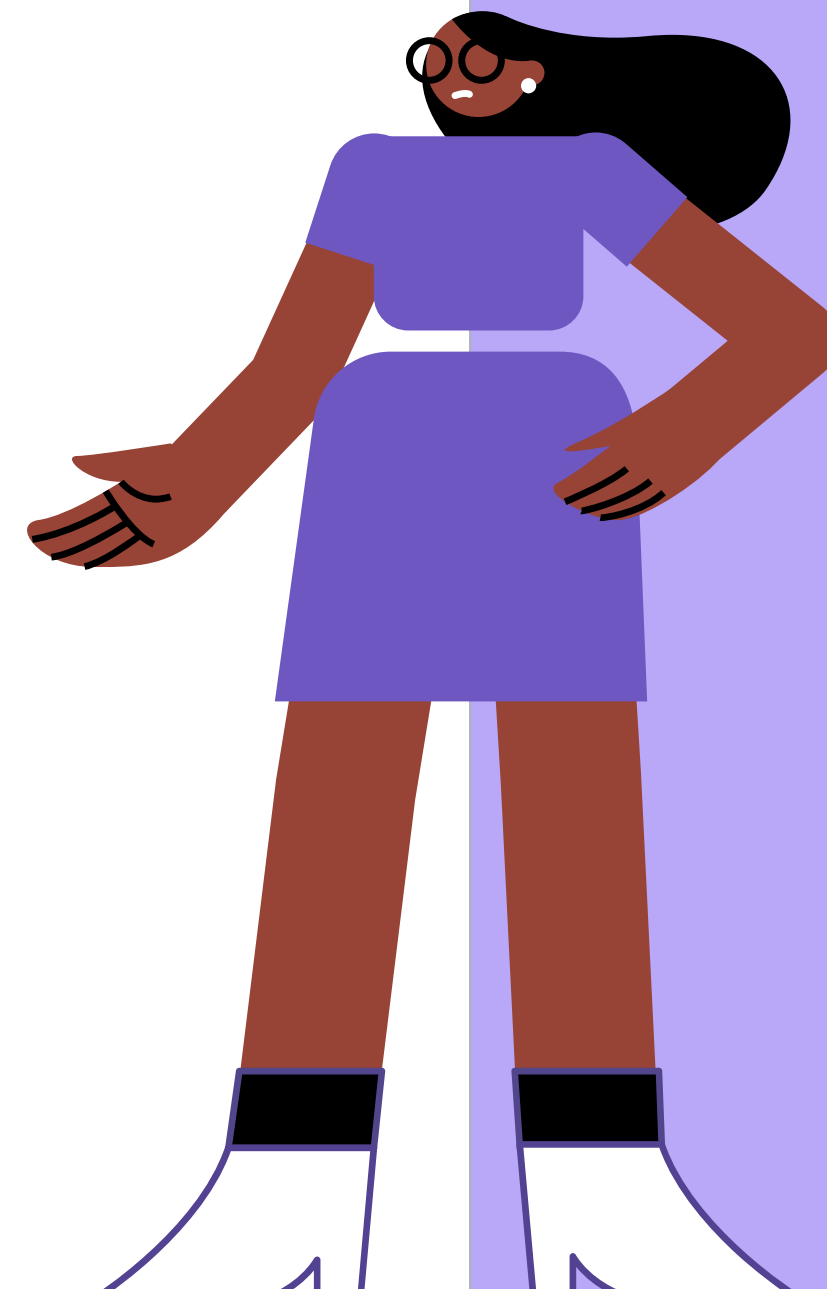


# Conclusiones

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.



¡Muchas gracias!