



INTRODUCCIÓN A LA CLASE

GO WEB

Objetivos de esta clase

- Conocer y comprender los métodos *PUT* y *PATCH*.
- Comprender las diferencias de utilidad entre *PUT* y *PATCH* para saber cuál utilizar en cada caso en particular.
- Manipular peticiones *PUT* y *PATCH* en *GO*.
- Conocer y comprender el método *DELETE*.
- Comprender la utilidad del método *DELETE*.
- Manipular una petición *DELETE* en *GO*.





MÉTODOS “PUT” & “PATCH”

GO WEB

// ¿Qué son y para qué sirven?

“Ambos **son métodos HTTP** para realizar peticiones al servidor.

Sirven para **actualizar un recurso** en una ubicación.”



- Ambos métodos se utilizan para actualizar un recurso.
- Cada uno trabaja de diferente forma.



PUT

// Método de HTTP

IT BOARDING

BOOTCAMP





PUT [Utilidad y características]

Una solicitud “**PUT**” se utiliza para actualizar un recurso, reemplazándolo -en su totalidad- por otro recurso nuevo.

Una solicitud “**PUT**” siempre contiene un recurso completo. Esto es necesario, porque una cualidad de las solicitudes “**PUT**” es la *idempotencia*.



La *idempotencia* es la calidad de producir el mismo resultado, incluso, si la misma solicitud se realiza varias veces.

No produce efectos secundarios.

PUT [Diferencias con el método POST]

| POST | PUT |
|--|---|
| Envía información al servidor para CREAR un nuevo recurso -aún no existente-, con los datos enviados en el cuerpo de la solicitud. | Envía información al servidor para ACTUALIZAR o REEMPLAZAR un recurso ya existente, con los datos enviados en el cuerpo de la solicitud. Reemplaza los datos del recurso completamente. |
| No es idempotente. | Es idempotente. |



Utiliza una petición “PUT” cuando quieras reemplazar una representación del elemento de destino con los datos de la petición.



Ejemplo simbólico haciendo un “PUT”

En el caso que Internet es una calle, las casas en esta calle están prefabricadas y las direcciones de las casas son URL.

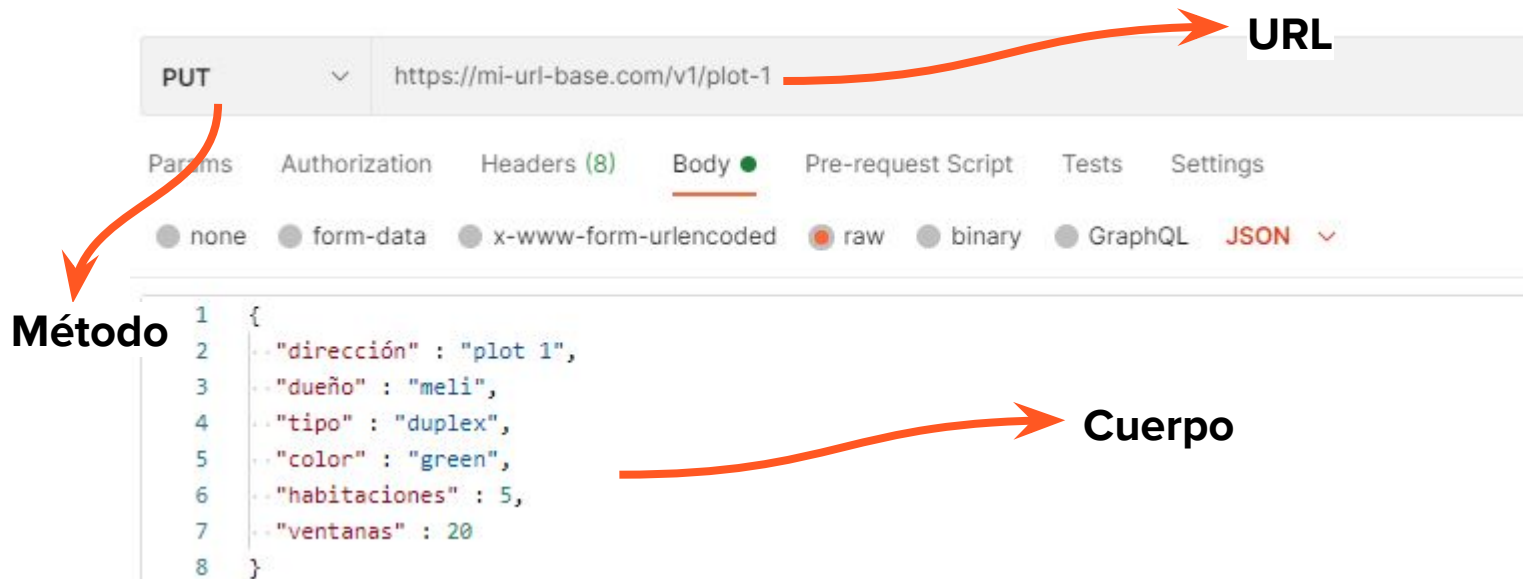
La calle se dividió en parcelas numeradas con una casa por parcela de modo que la casa-1 estaba en la parcela-1 y así sucesivamente. Las casas están prefabricadas, por lo que simplemente se dejan caer en su ubicación. No se requiere construcción en el sitio.

Si se realiza una solicitud “PUT” a '<https://mi-url-base.com/v1/plot-1>' con una casa prefabricada, se indica *“Por favor, PONGA esta casa en la ubicación marcada como plot-1”*. Esa instrucción busca en nuestra calle la ubicación especificada y reemplaza el contenido en esa ubicación. Si no se encuentra nada en la ubicación, simplemente COLOCARÁ el recurso en la ubicación. En este caso, una casa prefabricada completa.



Ejemplo simbólico haciendo un “*PUT*”

Así se visualiza una solicitud “*PUT*” para “poner” una casa prefabricada en lugar de otra ya existente:

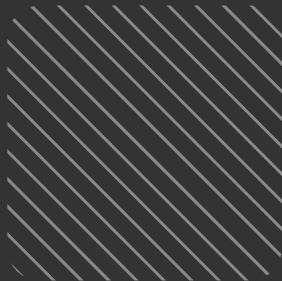


PUT EN GO

//Método de HTTP

IT BOARDING

BOOTCAMP



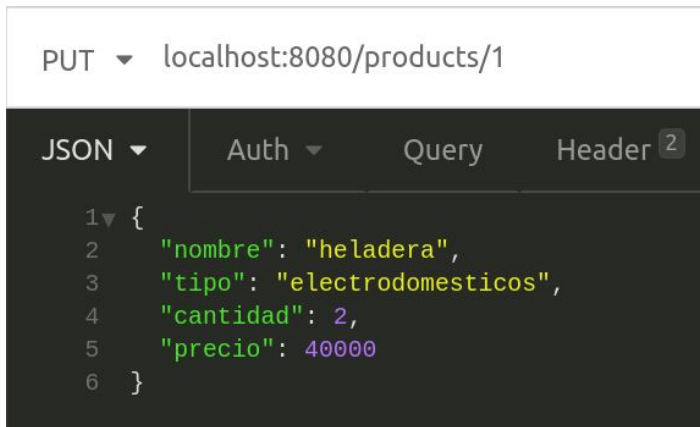
// ¿Cómo podemos recibir una petición *PUT*?

```
{  
  pr.POST("/", p.Store())  
  pr.GET("/", p.GetAll())  
  pr.PUT("/:id", p.Update())  
}
```

Petición/Respuesta

Al recibir una solicitud *PUT* con un *path parameter* se nos indica el *Id* del producto almacenado que se está intentando reemplazar.

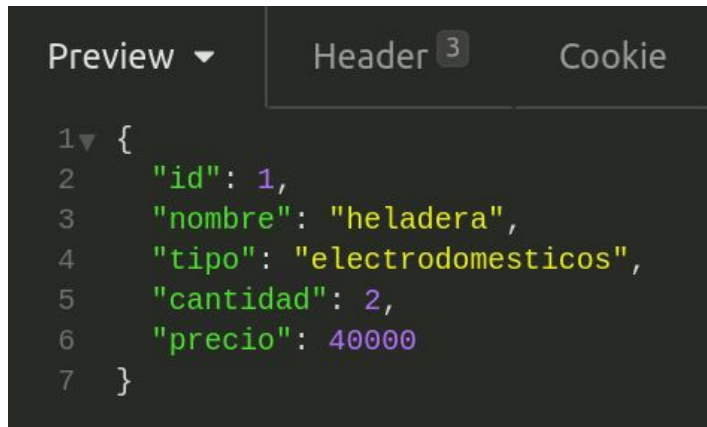
Petición:



The screenshot shows a REST client interface. The top bar displays the method 'PUT' and the URL 'localhost:8080/products/1'. Below this, there are tabs for 'JSON', 'Auth', 'Query', and 'Header'. The 'JSON' tab is selected, showing a JSON body with the following structure:

```
1 {  
2   "nombre": "heladera",  
3   "tipo": "electrodomesticos",  
4   "cantidad": 2,  
5   "precio": 40000  
6 }
```

Respuesta:



The screenshot shows the response of the PUT request. The top bar has tabs for 'Preview', 'Header', and 'Cookie'. The 'Preview' tab is selected, showing the response body as a JSON object:

```
1 {  
2   "id": 1,  
3   "nombre": "heladera",  
4   "tipo": "electrodomesticos",  
5   "cantidad": 2,  
6   "precio": 40000  
7 }
```



Paso 1 - Agregar método en Interface

- Definir un servicio web mediante el método *PUT*, el cual tendrá como path "**products/:id**".
- Agregar los métodos a utilizar en las interfaces de **Repository** y **Service**.

{}

```
type Repository interface {  
    GetAll() ([]Product, error)  
    Store(id int, name, productType string, count int, price float64) (Product, error)  
    LastID() (int, error)  
    Update(id int, name, productType string, count int, price float64) (Product, error)  
}
```

{}

```
type Service interface {  
    GetAll() ([]Product, error)  
    Store(name, productType string, count int, price float64) (Product, error)  
    Update(id int, name, productType string, count int, price float64) (Product, error)  
}
```



Paso 2 - Repositorio

Se implementa la funcionalidad para actualizar el producto en memoria, en caso que coincida con el ID enviado, caso contrario retorna un error.

{}

```
func (r *repository) Update(id int, name, productType string, count int, price float64) (Product, error) {
    p := Product{Name: name, Type: productType, Count: count, Price: price}
    updated := false
    for i := range ps {
        if ps[i].ID == id {
            p.ID = id
            ps[i] = p
            updated = true
        }
    }
    if !updated {
        return Product{}, fmt.Errorf("Producto %d no encontrado", id)
    }
    return p, nil
}
```



Paso 3 - Servicio

Dentro del servicio se llama al repositorio para que proceda a actualizar el producto.

```
{  
    func (s *service) Update(id int, name, productType string, count int, price float64)  
    (Product, error) {  
        return s.repository.Update(id, name, productType, count, price)  
    }  
}
```



Paso 4 - Se agrega el controlador Update en el Handler de productos.

{}

```
func (c *Product) Update() gin.HandlerFunc {
    return func(ctx *gin.Context) {
        token := ctx.GetHeader("token")
        if token != "123456" {
            ctx.JSON(401, gin.H{ "error": "token inválido" })
            return
        }
        id, err := strconv.ParseInt(ctx.Param("id"), 10, 64)
        if err != nil {
            ctx.JSON(400, gin.H{ "error": "invalid ID" })
            return
        }
        var req request
        if err := ctx.ShouldBindJSON(&req); err != nil {
            ctx.JSON(400, gin.H{ "error": err.Error() })
            return
        }
        if req.Name == "" {
            ctx.JSON(400, gin.H{ "error": "El nombre del producto es requerido" })
            return
        }
        if req.Type == "" {
            ctx.JSON(400, gin.H{ "error": "El tipo del producto es requerido" })
            return
        }
        if req.Count == 0 {
            ctx.JSON(400, gin.H{ "error": "La cantidad es requerida" })
            return
        }
        if req.Price == 0 {
            ctx.JSON(400, gin.H{ "error": "El precio es requerido" })
            return
        }
        p, err := c.service.Update(int(id), req.Name, req.Type, req.Count, req.Price)
        if err != nil {
            ctx.JSON(404, gin.H{ "error": err.Error() })
            return
        }
        ctx.JSON(200, p)
    }
}
```



Paso 5 - Main del programa

Dentro del main del programa se agrega el router correspondiente al método PUT.

```
func main() {  
    repo := products.NewRepository()  
    service := products.NewService(repo)  
    p := handler.NewProduct(service)  
    r := gin.Default()  
    pr := r.Group("/products")  
    pr.POST("/", p.Store())  
    pr.GET("/", p.GetAll())  
    pr.PUT("/:id", p.Update())  
    r.Run()  
}
```



Retornar Producto

En el ejemplo, retorna un *json* con un *status code* 200 y la *data* actualizada del recurso reemplazado.

Es posible retornar también un listado actualizado de los productos almacenados, para constatar que se ha reemplazado el indicado en la *request*.

Si el recurso a reemplazar no fue encontrado, el *json* de la *response* contendrá *status code* 400, una *data* vacía, y un mensaje de error indicando lo sucedido.

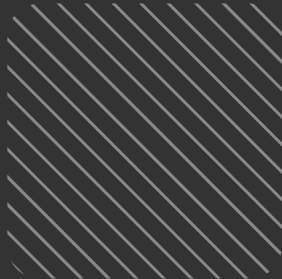


PATCH

//Método de HTTP

IT BOARDING

BOOTCAMP





PATCH [Utilidad y características]

- Una solicitud *PATCH* se utiliza para realizar cambios en parte del recurso en una ubicación.
- “parchea” el recurso, cambiando sus propiedades.
- Se utiliza para realizar actualizaciones menores a los recursos y no es necesario que sea idempotente.
- Una solicitud *PATCH* no contiene un recurso completo, sino únicamente las propiedades a modificar del recurso de que se trate.

PATCH [Diferencias con el método *PUT*]

| PATCH | PUT |
|--|--|
| Envía información al servidor para ACTUALIZAR o MODIFICAR alguna/s propiedad/es de un recurso ya existente, con los datos enviados en el cuerpo de la solicitud. | Envía información al servidor para ACTUALIZAR o REEMPLAZAR un recurso ya existente, con los datos enviados en el cuerpo de la solicitud. Coloca (pone) un recurso en una ubicación indicada. |
| No es idempotente. | Es idempotente. |



Utiliza una petición "*PATCH*" para modificar alguna propiedad de un recurso existente en el servidor con los datos de la petición.



Ejemplo simbólico haciendo un “*PATCH*”

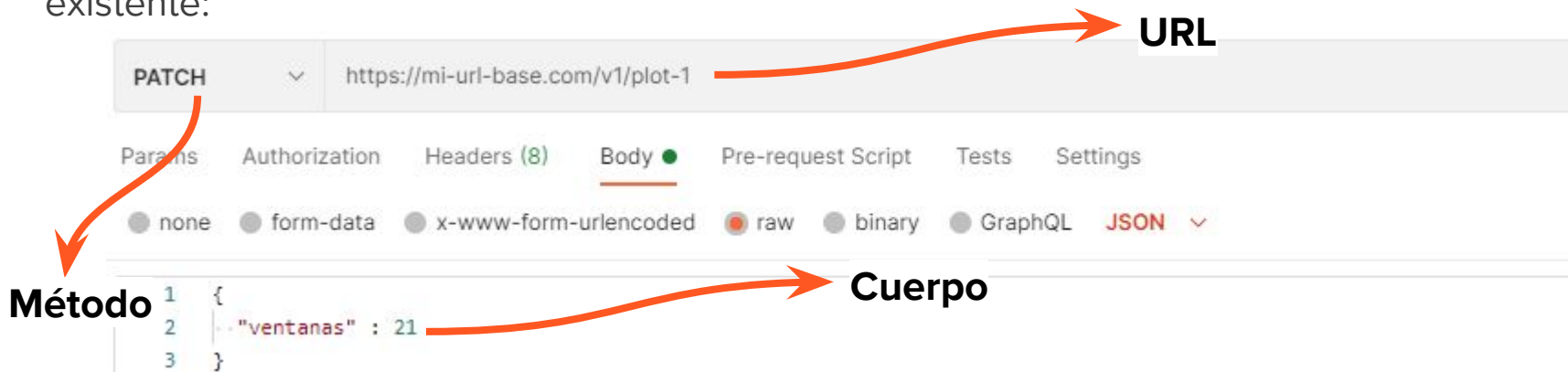
Continuando con el ejemplo anterior, se puede agregar fácilmente una nueva ventana a la casa en la parcela 1 sin tener que enviar una casa entera completamente nueva.

Todo lo que hay que hacer es enviar sólo la nueva ventana y “PARCHEAR” la casa vieja con una nueva ventana.



Ejemplo simbólico haciendo un “*PATCH*”

Así se visualiza una solicitud “*PATCH*” para agregar una ventana en una casa ya existente:



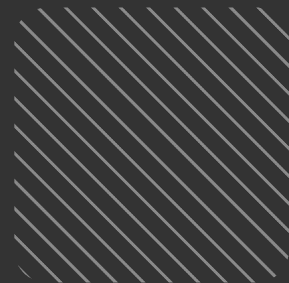


PATCH EN GO

//Método de HTTP

IT BOARDING

BOOTCAMP



// ¿Cómo podemos recibir una petición *PATCH*?

{}

```
pr.POST("/", p.Store())  
pr.GET("/", p.GetAll())  
pr.PUT("/:id", p.Update())  
pr.PATCH("/:id", p.UpdateName())
```

IT BOARDING

BOOTCAMP

Petición/Respuesta

Se recibe una solicitud *PATCH* con un *path parameter* que indique el *Id* del producto almacenado que se está intentando modificar, y se modifica solo el campo nombre.

Petición:

PATCH ▾ localhost:8080/products/2

JSON ▾

Auth ▾

Query

He

```
1 ▾ {  
2   "nombre": "microondas"  
3 }
```

Respuesta:

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



Paso 1 - Agregar método en Interface

Definir un servicio web mediante el método *PATCH*, el cual tendrá como path "**products/:id**".
Como primer paso se agregarán los métodos a utilizar en las interfaces de **Repository** y **Service**.

{}

```
type Repository interface {
    GetAll() ([]Product, error)
    Store(id int, name, productType string, count int, price float64) (Product, error)
    LastID() (int, error)
    UpdateName(id int, name string) (Product, error)
    Update(id int, name, productType string, count int, price float64) (Product, error)
}
```

{}

```
type Service interface {
    GetAll() ([]Product, error)
    Store(name, productType string, count int, price float64) (Product, error)
    Update(id int, name, productType string, count int, price float64) (Product, error)
    UpdateName(id int, name string) (Product, error)
}
```



Paso 2 - Repositorio

Se implementa la funcionalidad para actualizar el nombre del producto en memoria, en caso que coincida con el ID enviado, caso contrario retorna un error.

```
{}  
  
func (r *repository) UpdateName(id int, name string) (Product, error) {  
    var p Product  
    updated := false  
    for i := range ps {  
        if ps[i].ID == id {  
            ps[i].Name = name  
            updated = true  
            p = ps[i]  
        }  
    }  
    if !updated {  
        return Product{}, fmt.Errorf("Producto %d no encontrado", id)  
    }  
    return p, nil  
}
```



Paso 3 - Servicio

Dentro del servicio se llama al repositorio para que proceda a actualizar el nombre del producto.

```
{  
    func (s *service) UpdateName(id int, name string) (Product, error) {  
        return s.repository.UpdateName(id, name)  
    }  
}
```



Paso 4 - Se agrega el controlador UpdateName en el Handler de productos.

{}

```
func (c *Product) UpdateName() gin.HandlerFunc {
    return func(ctx *gin.Context) {
        token := ctx.GetHeader("token")
        if token != "123456" {
            ctx.JSON(401, gin.H{ "error": "token inválido" })
            return
        }
        id, err := strconv.ParseInt(ctx.Param("id"), 10, 64)
        if err != nil {
            ctx.JSON(400, gin.H{ "error": "invalid ID" })
            return
        }
        var req request
        if err := ctx.ShouldBindJSON(&req); err != nil {
            ctx.JSON(400, gin.H{ "error": err.Error() })
            return
        }
        if req.Name == "" {
            ctx.JSON(400, gin.H{ "error": "El nombre del producto es requerido" })
            return
        }
        p, err := c.service.UpdateName(int(id), req.Name)
        if err != nil {
            ctx.JSON(404, gin.H{ "error": err.Error() })
            return
        }
        ctx.JSON(200, p)
    }
}
```



Paso 5 - Main del programa

Dentro del main del programa se agrega el router correspondiente al método PATCH.

{}

```
func main() {  
    repo := products.NewRepository()  
    service := products.NewService(repo)  
    p := handler.NewProduct(service)  
    r := gin.Default()  
    pr := r.Group("/products")  
    pr.POST("/", p.Store())  
    pr.GET("/", p.GetAll())  
    pr.PUT("/:id", p.Update())  
    pr.PATCH("/:id", p.UpdateName())  
    r.Run()  
}
```



Retornar Producto

En nuestro ejemplo, retornaremos un *json* con un *status code* 200 y la *data* actualizada del recurso modificado.

Tenemos la posibilidad de retornar también un listado actualizado de los productos almacenados, para constatar que se ha modificado el indicado en la *request*.

Si el recurso a modificar no fue encontrado, el *json* de la *response* contendrá *status code* 400, una *data* vacía, y un mensaje de error indicando lo sucedido.



// Para concluir

Hemos aprendido cómo implementar el método PUT y PATCH en un aplicación web.

¡Continuemos aprendiendo!

IT BOARDING

BOOTCAMP



MÉTODO “*DELETE*”

GO WEB

// ¿Qué es y para qué sirve?

“Una solicitud DELETE es un método HTTP que se utiliza para solicitar al servidor que elimine un recurso existente en la ubicación que indica en la URL.

Contiene la ubicación del recurso a eliminar (el ID del recurso a eliminar).

Es un método HTTP idempotente.”

Ejemplo “*DELETE*”

¿Cómo eliminar un producto específico que se encuentra almacenado?

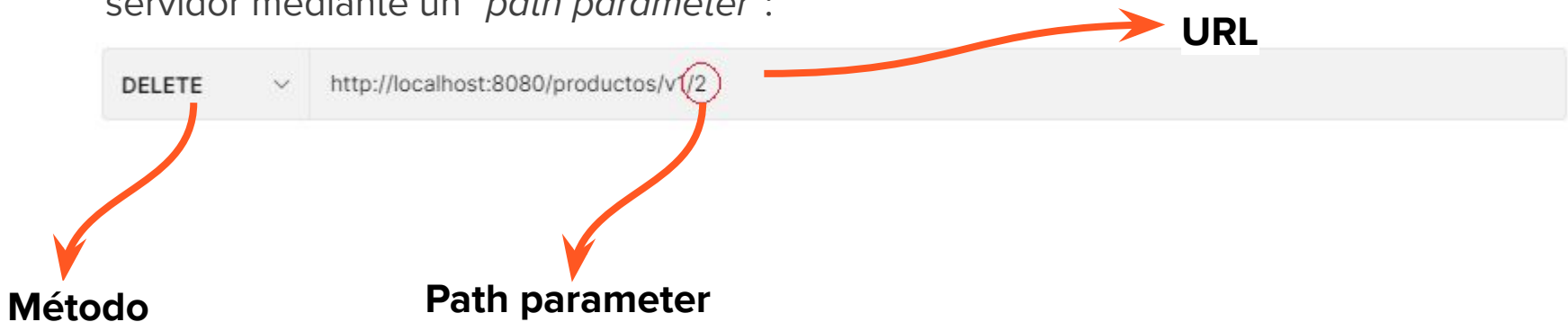
Es necesario:

1. Conocer cuál es el “*ID*” del producto que queremos eliminar.
2. Mediante una solicitud (*request*) *DELETE*, le pedimos al servidor que elimine el producto que le indicamos mediante el “*id*” del mismo, pasado como parámetro en la URL (*path parameter*) a la que enviamos la petición.
3. El servidor buscará si, entre los productos almacenados, existe un producto con el “*id*” pasado como parámetro.
4. En caso de encontrarlo, devolverá un estado 2xx, que indicará que el producto fue eliminado.



Ejemplo “*DELETE*”

Así se visualiza una solicitud “*DELETE*” para eliminar un recurso existente en el servidor mediante un “*path parameter*”:



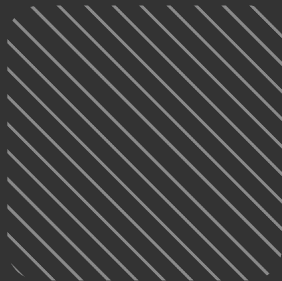
En las solicitudes de tipo “*DELETE*”, no necesitamos enviar datos en el cuerpo (*body*) de las mismas. Sólo enviamos un “*path parameter*” que le indica al servidor el recurso a eliminar.

DELETE EN GO

//Método de HTTP

IT BOARDING

BOOTCAMP



// ¿Cómo podemos recibir una petición *DELETE*?

{}

```
pr.POST("/", p.Store())  
pr.GET("/", p.GetAll())  
pr.PUT("/:id", p.Update())  
pr.PATCH("/:id", p.UpdateName())  
pr.DELETE("/:id", p.Delete())
```

IT BOARDING

BOOTCAMP

Petición/Respuesta

Se recibe una solicitud *DELETE* con un *path parameter* que indica el *Id* del producto almacenado que se está intentando eliminar.

Petición:

- No necesita tener datos en su *body*.
- Solo precisa indicar el “*Id*” del producto que se desea eliminar para identificarlo correctamente.

Se pasará como parámetro de ruta.

Respuesta:

Si al procesar la petición, el servidor encuentra el “*Id*” especificado, elimina el producto y devuelve un status 2xx.



La respuesta, en su *body*, puede contener un detalle actualizado de los productos para constatar que el indicado fue eliminado.



Paso 1 - Agregar método en Interface

Se define un servicio web mediante el método *DELETE*, el cual tiene como path "*products/:id*".
Se agregarán los métodos a utilizar en las interfaces de **Repository** y **Service**.

{}

```
type Repository interface {  
    GetAll() ([]Product, error)  
    Store(id int, name, productType string, count int, price float64) (Product, error)  
    LastID() (int, error)  
    UpdateName(id int, name string) (Product, error)  
    Update(id int, name, productType string, count int, price float64) (Product, error)  
    Delete(id int) error  
}
```

{}

```
type Service interface {  
    GetAll() ([]Product, error)  
    Store(name, productType string, count int, price float64) (Product, error)  
    Update(id int, name, productType string, count int, price float64) (Product, error)  
    UpdateName(id int, name string) (Product, error)  
    Delete(id int) error  
}
```



Paso 2 - Repositorio

Se implementa la funcionalidad para eliminar el producto en memoria, en caso que coincida con el ID enviado, caso contrario retorna un error.

```
func (r *repository) Delete(id int) error {
    deleted := false
    var index int
    for i := range ps {
        if ps[i].ID == id {
            index = i
            deleted = true
        }
    }
    if !deleted {
        return fmt.Errorf("Producto %d no encontrado", id)
    }
    ps = append(ps[:index], ps[index+1:]...)
    return nil
}
```



Paso 3 - Servicio

Dentro del servicio se llama al repositorio para que proceda a eliminar el producto.

```
{}  
func (s *service) Delete(id int) error {  
    return s.repository.Delete(id)  
}
```



Paso 4 - Se agrega el controlador Delete en el Handler de productos.

```
{}
```

```
func (c *Product) Delete() gin.HandlerFunc {
    return func(ctx *gin.Context) {
        token := ctx.GetHeader("token")
        if token != "123456" {
            ctx.JSON(401, gin.H{ "error": "token inválido" })
            return
        }
        id, err := strconv.ParseInt(ctx.Param("id"), 10, 64)
        if err != nil {
            ctx.JSON(400, gin.H{ "error": "invalid ID" })
            return
        }
        err = c.service.Delete(int(id))
        if err != nil {
            ctx.JSON(404, gin.H{ "error": err.Error() })
            return
        }
        ctx.JSON(200, gin.H{ "data": fmt.Sprintf("El producto %d ha sido eliminado",
id) })
    }
}
```



Paso 5 - Main del programa

Dentro del main del programa se agrega el router correspondiente al método DELETE.

```
{  
func main() {  
    repo := products.NewRepository()  
    service := products.NewService(repo)  
    p := handler.NewProduct(service)  
    r := gin.Default()  
    pr := r.Group("/products")  
    pr.POST("/", p.Store())  
    pr.GET("/", p.GetAll())  
    pr.PUT("/:id", p.Update())  
    pr.PATCH("/:id", p.UpdateName())  
    pr.DELETE("/:id", p.Delete())  
    r.Run()  
}
```



// Para concluir

Hemos aprendido cómo implementar el método **DELETE** en un aplicación web.

¡Continuemos aprendiendo!

IT BOARDING

BOOTCAMP

Retornar Producto

En nuestro ejemplo, retornaremos un *json* con un *status code* 200 y un listado actualizado de los productos para constatar que se ha eliminado el indicado en la *request*.

Por lo normal, también se podría retornar el detalle del recurso eliminado.

Si el recurso a eliminar no fue encontrado, el *json* de la *response* contendrá *status code* 400, una *data* vacía, y un mensaje de error indicando lo sucedido.



Si bien la *response* de una request DELETE no requiere un contenido en su *body*, por buenas prácticas debemos siempre retornar un *status code*, *data* del recurso intervenido, y un mensaje de error.





Gracias.

IT BOARDING

BOOTCAMP

