DIGITALHOUSE

# Middlewares

Certified Tech Developer

The Ultimate Degree

# Índice

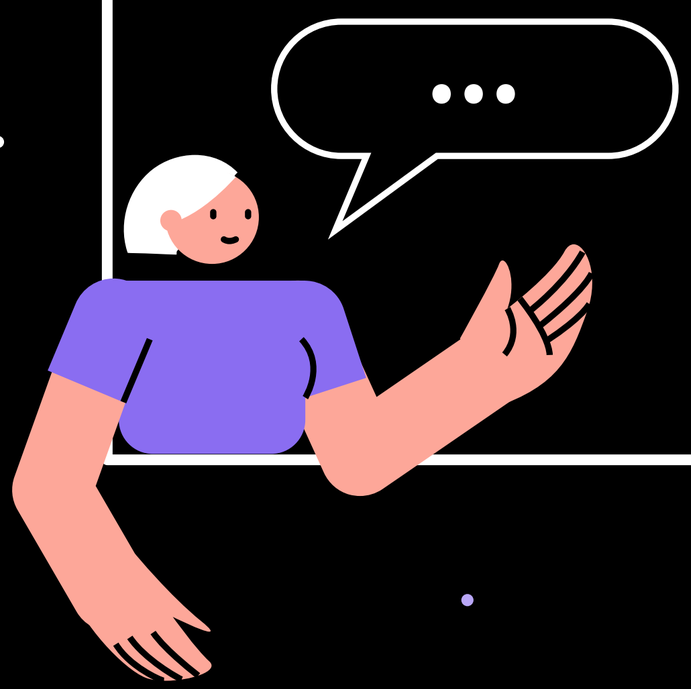


01

# ¿Qué es un middleware?

El middleware es una entidad que intercepta el ciclo de vida de una solicitud o respuesta del servidor. En pocas palabras, es un fragmento de código que se ejecuta antes o después de que el servidor atienda una solicitud con una respuesta.

En términos de arquitectura del software, un middleware es un componente en nuestras aplicaciones que nos va a permitir centrarnos en la lógica de negocio, resolviendo algunas complejidades de bajo nivel.



# Middleware

Para explicar qué son los middlewares, vamos a comenzar con un ejemplo sin sentido.
Tenemos tres funciones que tienen la misma firma: **sumar**, **restar** y **multiplicar**. Ahora supongamos que, por alguna razón, a esas tres funciones haya que agregarles comportamientos.
Primero, necesitamos mostrar la hora cada vez que una de esas funciones es invocada. Además, necesitamos registrar en un archivo el resultado que generó dicha función (a modo de ejemplo solo mostraremos un mensaje).

**¿Qué deberíamos hacer?**
La primera opción que se nos ocurre es modificar todas las funciones:

```
func sumar(a, b int) int {
    //Mostrar la fecha y la hora
    fmt.Println(time.Now())
    resultado := a + b
    // Hacer algo después
    return resultado
}
```

# Middleware

Si seguimos esta lógica, deberíamos escribir en cada función las mismas líneas de código. ¿Qué pasaría si en vez de tener tres funciones, tenemos cientos de funciones?

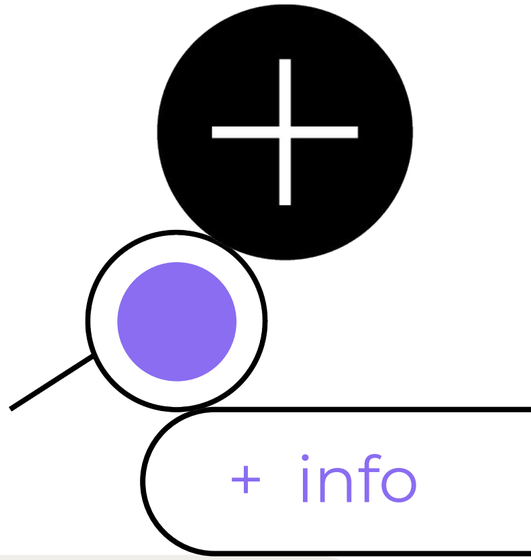
**Sería una locura pretender cambiar la misma lógica en cientos de funciones una y otra vez.**

¿Podemos hacer algo más inteligente? ¿Qué tal si separamos dicho comportamiento, lo abstraemos a otra función? ¿Y si intentamos hacer que las funciones principales (sumar, restar y multiplicar) no se enteren de comportamientos ajenos a ellas?

**Qué interesante sería crear comportamientos y funcionalidades para poder aplicarlos a las funciones que deseemos.**

# Un middleware es...

- Una técnica que se utiliza para resolver este tipo de cuestiones.
- Una función.
- Una función que recibe una función y retorna otra (con la misma firma), pero la nueva función es creada al vuelo y le agrega funcionalidad.




De esta manera, las funciones destino (sumar, restar y multiplicar), jamás se enterarán de los comportamientos que les estamos aplicando; ya que estas son envueltas y ejecutadas por otra función (el middleware).

# Middleware

Cada función hace solo una cosa, es responsable de realizar lo que se tiene que hacer. Además, el comportamiento de cada uno de ellos puede ser utilizado para todas aquellas funciones que posean la misma firma (que sean del mismo tipo).

**Por ejemplo:** si deseamos agregarle a cada una de las funciones (sumar, restar y multiplicar) el comportamiento de mostrar la hora cada vez que una de ellas sea invocada, podemos hacerlo creando el siguiente middleware.

```
func mostrarTiempo(f func(int, int)
int) func(int, int) int {

    return func(a, b int) int {
        fmt.Println(time.Now())
        return f(a, b)
    }
}
```

# Middleware

```
func main() {
    fmt.Println("Resultado: ",
mostrarTiempo(sumar)(10, 5))
}
```

Si lo hacemos paso a paso, sería así:

```
f := mostrarTiempo(sumar)
resultado := f(10, 20)
fmt.Println(resultado)
```

02

# Encadenamiento de middlewares

# Encadenamiento de middlewares

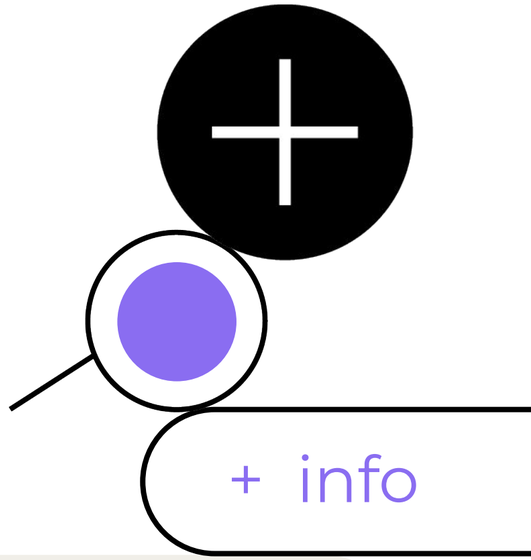
Crearemos otro middleware que mostrará un texto después de haber ejecutado la función recibida:

```
func mostrarPosterior(f func(int, int) int)
func(int, int) int {
    return func(a, b int) int {
        resultado := f(a, b)
        // Hacer algo después
        fmt.Println("Se ejecutó la función")
        return resultado
    }
}
```

# Ejecutamos

Ahora a la función **sumar** le aplicaremos ambos comportamientos sin que se entere:

```go
func main() {
    fmt.Println("Resultado: ",
    mostrarTiempo(mostrarPosterior(sumar))(10, 5))
    // Es lo mismo que hacerlo paso a paso:
    // fm1 := mostrarPosterior(sumar)
    // fm2 := mostrarTiempo(fm1)
    // resultado := fm2(10, 20)
    // fmt.Println(resultado)
}
```




Cada middleware tiene el poder y la responsabilidad de decidir si ejecuta instrucciones antes de la función recibida o después de la función recibida o... ambas a la vez.

# Apilar middlewares

Lo que sucede al apilar middlewares es que cada uno de ellos puede procesar instrucciones antes y después de la función recibida. Cuando finalmente se ejecuta la función recibida, es como si fuera una sola función. De hecho, lo es, con toda la funcionalidad acumulada de las anteriores. Veamos el siguiente código para demostrar el orden en que se ejecutan las instrucciones de dos interceptores:

```go
func main() {
    fmt.Println(inter1(inter2(sumar))(10, 20))
}

func inter1(f func(int, int) int) func(int, int) int {
    return func(a, b int) int {
        fmt.Println("inter1 ANTES")
        // Ejecutar la función recibida con firma func(in,
int) int
        resultado := f(a, b)
        fmt.Println("inter1 DESPUES")
        return resultado
    }
}
```

# Apilar middlewares

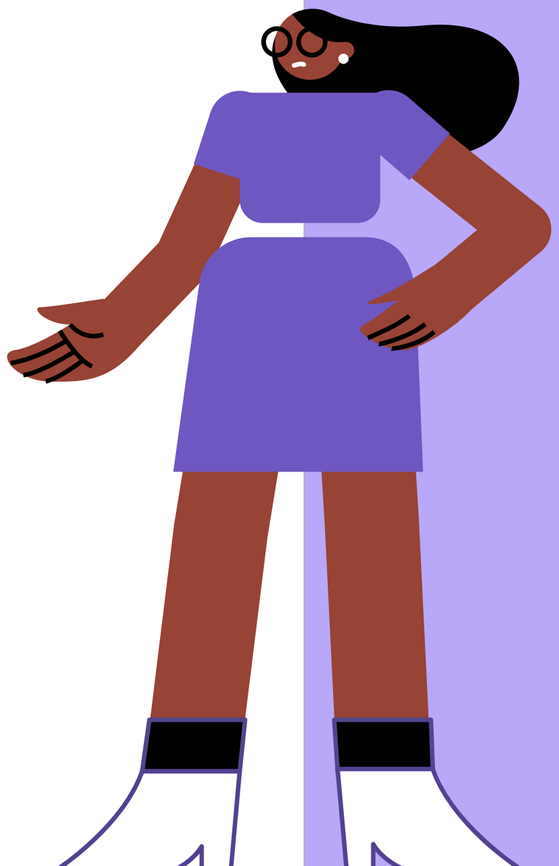
```
func inter2(f func(int, int) int) func(int, int)
int {
    return func(a, b int) int {
        fmt.Println("inter2 ANTES")
        // Ejecutar la función recibida con firma
func(in, int) int
        resultado := f(a, b)
        fmt.Println("inter2 DESPUES")
        return resultado
    }
}
```

Mostrará el siguiente resultado:

```
inter1 antes
inter2 antes
inter2 después
inter1 después
```

# Conclusiones

Un middleware es una técnica de programación que consiste en crear una función que reciba por parámetro otra función con una firma determinada, así podrá retornar una nueva función creada al vuelo con la misma firma que la función recibida. En esta se puede inyectar funcionalidad antes y después de la llamada a la función recibida.



DIGITALHOUSE

# ¡Muchas gracias!