Universidad Nacional de Colombia Sede Bogotá
Asignatura: Ingeniería de Software 1 (2016701)
Docente: Oscar Eduardo Alvarez Rodriguez

PATRÓN DE DISEÑO. CADENA DE RESPONSABILIDAD

Elaborado por: David Ladino Camilo D'Aleman
Team Sobrecupo Jorge Cuadrado Smith Yañez

PROYECTO - CLEAN CODE

QUÉ ES Y PARA QUÉ SIRVE

El patrón *Chain of Responsibility* (Cadena de Responsabilidad) pertenece a la categoría de patrones de comportamiento. Su objetivo principal es permitir que una solicitud pase por una serie de manejadores, uno tras otro, hasta que alguno la procese. Cada manejador decide si actúa sobre la solicitud o la pasa al siguiente en la cadena.

Una implementación muy común de este patrón, especialmente en aplicaciones web, es el uso de **middlewares**. Estos componentes permiten:

- Procesar solicitudes entrantes.
- Modificar el contenido de las solicitudes o respuestas.
- Finalizar el ciclo de solicitud-respuesta.
- O bien, continuar con el siguiente middleware.

El propósito de usar middlewares es dividir el procesamiento de peticiones en pasos pequeños, reutilizables y especializados, lo que hace que el sistema sea más ordenado y flexible.

¿CÓMO SE USÓ EN EL PROYECTO?

Este patrón se aplicó en distintas capas del sistema, aprovechando su naturaleza escalonada:

- **En el servidor backend:**
  Desde el archivo principal, se define una cadena de middlewares que intercepta todas las solicitudes. Primero se configura uno para manejar CORS (para permitir la comunicación con el frontend), luego uno que convierte automáticamente el cuerpo de las solicitudes a JSON, y al final, los middlewares de enrutamiento que redirigen cada solicitud al módulo correspondiente según la URL.

- **En el sistema de autenticación:**
  Se desarrollaron dos middlewares específicos. Uno es de autenticación *opcional*, que verifica si hay un token válido, pero deja continuar incluso si no lo hay. El otro es de autenticación *obligatoria*, y bloquea el paso si el token está ausente o no es válido, devolviendo un error inmediato.

- **En las rutas:**
  Cada conjunto de rutas utiliza los middlewares que necesita. Por ejemplo, las rutas públicas (como la vista general de posts) usan la autenticación opcional, para ofrecer contenido personalizado a quienes están logueados, pero también funcionar sin problemas para visitantes. En cambio, las rutas privadas (como consultar los posts de un usuario) requieren autenticación obligatoria.

- **En el frontend (React):**
  React también aplica el patrón mediante hooks y efectos. Por ejemplo, cuando un usuario cambia un filtro, se activa una cadena de eventos: se detecta el cambio, se construye la URL adecuada, se hace la petición, se procesa la respuesta y se actualiza el estado de la UI. Todo eso ocurre en etapas bien separadas.

- **En el manejo de datos:**
  El componente principal (`Home`) coordina varias cadenas. Una se encarga de la carga inicial de datos, otra responde a los cambios de filtro, y una más maneja acciones como agregar un post a favoritos.

¿POR QUÉ FUE ÚTIL ESTE PATRÓN?

La elección de este patrón no fue casual: se adapta perfectamente a las necesidades del sistema, y estas son algunas de las razones:

- **Separación clara de responsabilidades:** cada middleware se enfoca en una sola tarea. Por ejemplo, uno se dedica a validar tokens, otro maneja los headers de CORS, y los controladores atienden la lógica propia de cada funcionalidad. Esto hace que el código sea más limpio, organizado y fácil de mantener.

- **Reutilización de lógica:** el middleware de autenticación opcional se utiliza en muchas rutas que requieren cierta personalización pero no restricción total. Esto evita repetir código y garantiza que la validación se maneje de forma consistente en todo el sistema.

- **Facilidad para escalar:** si en el futuro se quiere agregar funciones como logging, control de tráfico (rate limiting) o validación de datos, solo hace falta crear un nuevo middleware y agregarlo a la cadena. No es necesario tocar lo que ya funciona.

- **Manejo ordenado de errores:** cada middleware puede detectar y responder a errores en su ámbito sin afectar a los demás. Por ejemplo, si la autenticación falla, se corta el flujo y se devuelve un error adecuado, sin que se ejecute el resto de la lógica.

- **Pruebas más sencillas:** los middlewares se pueden probar de forma individual, y los controladores se prueban asumiendo que las validaciones ya se hicieron correctamente. Esto hace que las pruebas unitarias e integradas sean más fáciles y confiables.

En un proyecto de turismo, donde hay distintos tipos de usuarios y niveles de acceso, este patrón permite adaptar el comportamiento del sistema según el contexto, sin tener que duplicar lógica ni complicar el flujo general.