Análisis del patrón decorador con el framework NestJS

Alejandro Pulido – 202215711

Universidad de los Andes

Información general del proyecto

Para el análisis se ha seleccionado el framework para aplicaciones web NestJS. Este framework está construido sobre TypeScript, un transpilador de JavaScript para poder desarrollar orientado a objetos, y su uso se centra en la creación de aplicaciones server-side rendering que se ejecuten sobre Node.JS. Las aplicaciones server-side rendering, como su nombre lo menciona, renderizan su contenido en el servidor (es decir que se genera el DOM) y le entregan a el cliente (Navegador) el HTML para que lo muestre. Este framework es altamente robusto, tanto es así que hace uso de Express como framework interno para las peticiones HTTPS, sin embargo, se puede configurar para usar Fastify para las peticiones.

El manejo de otro framework dentro de Nest supone un reto para poder manejar las peticiones, ya que el trabajo de Nest como framework, además de proveer a los desarrolladores las herramientas suficientes para construir aplicaciones escalables, es manejar el hilo del desarrollo, impulsando, y hasta cierto punto forzando, al equipo de desarrollo a acoger una metodología de desarrollo y construir software en base a esa filosofía.

Documentación NestJS: https://docs.nestjs.com/custom-decorators

Param Decorators, Adapters y peticiones

Concretamente se analizarán los *param decorators* que provee Nest, en especial los encargados de las peticiones de red. Estos decoradores sirven para manejar las peticiones HTTPS, que como se mencionó anteriormente, se hace uso del framework Express por defecto para dirigirlas y se puede cambiar el uso de Express por Fastify. Adicionalmente, el desarrollador puede crear *custom decorators* que pueden ayudar a que se desarrolle un código más limpio y transparente.

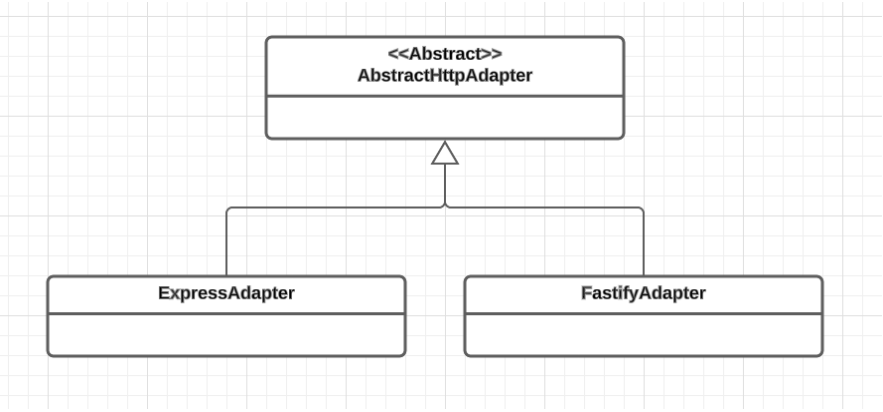
Para integrar el framework deseado para manejar las peticiones HTTPS, Nest posee un Adapter (haciendo uso del patrón adapter) para poder abstraer el

comportamiento del manejo de las peticiones y que el cliente simplemente trabaje con un adaptador.

```ts
import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter()
  );
  await app.listen(3000);
}
bootstrap();
```

El anexo 1, muestra cómo se hace uso de la clase `FastifyAdapter()` para poder cambiar el manejador de peticiones a Fastify. Esto permite que el cliente pueda escoger cómo manejar las peticiones, una decisión crítica para el desarrollo de una aplicación.



En el anexo 2 se puede ver la implementación del patrón adaptador que difiere ligeramente de la implementación original del patrón, sin embargo, mantiene el mismo objetivo y la misma idea de crear una interfaz al objeto envuelto (en este caso el framework a convenir).

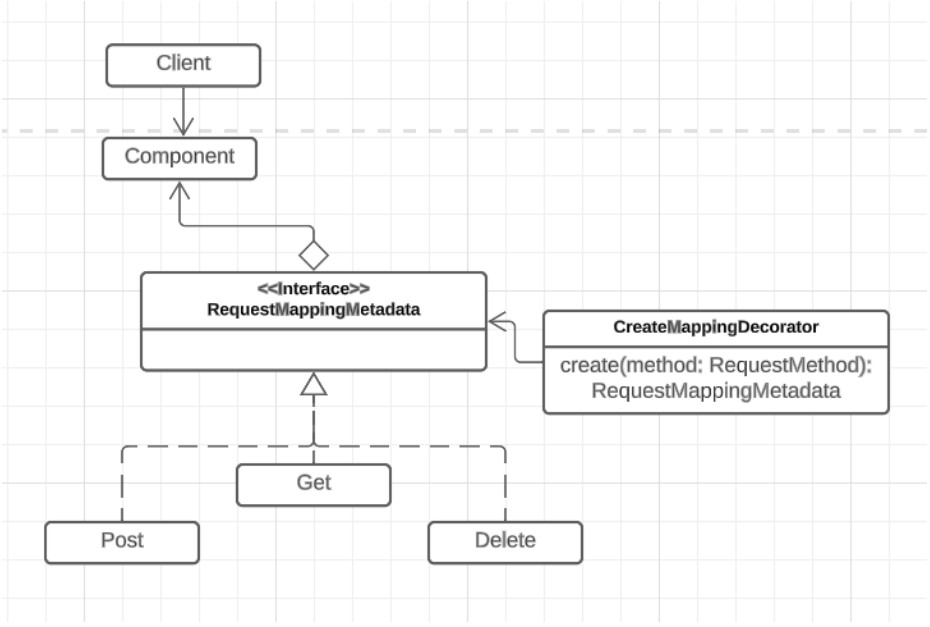
Para complementar el enfoque que tiene Nest, se diseña un conjunto de decoradores que abstraen la petición para que el cliente solo tenga que llamarla sin preocuparse cómo internamente se maneja la petición.

Patrón decorator como mejorador de interfaces

Para el análisis se ha seleccionado el patrón decorador, un patrón estructural que a grandes rasgos se encarga de brindarle una interfaz mejorada al método al cual será aplicado. Este patrón envuelve al método o clase que sea decorado y lo ejecutará, la particularidad de envolver un método o clase con un decorador es que este ejecutará acciones antes o después de la ejecución del objeto decorado.

Los principales usos del patrón se dan cuando a un método se le quieren dar funcionalidades extra sin necesidad de acoplar esta nueva funcionalidad al núcleo del método.

Decorators para manejar peticiones HTTP



El anexo 3 muestra la forma en la que están implementados los decoradores, los decoradores concretos implementan la interfaz `RequestMappingMetadata`, esta interfaz es con la que el componente interactúa, sin embargo, los decoradores concretos en realidad son Post, Get y Delete. El cliente al momento de declarar el componente también declarará con que decorador va a envolver a su método.

Ventajas del uso de HTTP decorators

El uso de los decoradores para manejar las peticiones tiene sentido ya que le provee al cliente una interfaz con la cual podrá realizar peticiones preocupándose únicamente por la lógica de negocio sin tener que lidiar con la forma en la que están implementadas las llamadas HTTP. Adicionalmente permite que el cliente pueda cambiar el framework con el que manejará las peticiones sin hacer cambios drásticos en la aplicación. El uso de decoradores permite que se cumpla el

principio de responsabilidad única, pues al abstraer la forma de hacer las peticiones, los métodos que hacen uso de decoradores solo tienen la tarea de ejecutar la petición que debe realizar sin tener que preocuparse por cómo se hace. Los decoradores también cumplen con el principio de inversión de dependencias, ya que un módulo de alto nivel, como, por ejemplo, la creación de un nuevo elemento no depende de un módulo de bajo nivel, como lo podría ser el manejo del método PUT para la creación del elemento.

Desventajas del uso de HTTP decorators

Una de las características más importantes de los decoradores es que pueden apilarse para brindarle más funcionalidades a un componente base. Sin embargo, este comportamiento no puede ser aprovechado por los decoradores HTTP pues un método solo debería poder realizar un tipo de petición a la vez por lo que de una u otra forma el uso de decoradores pierde sentido y se puede discutir si el uso de decoradores complica innecesariamente la forma en la que están construidas las aplicaciones, en especial teniendo en cuenta que para manejar las peticiones se crea un adaptador que en teoría por sí solo debería poderle permitir al cliente manejar las peticiones sin necesidad de decoradores.

¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

Como se menciona en el anterior punto, los adaptadores de los manejadores de peticiones podrían ser una solución más que suficiente para el manejo de peticiones, pues el objetivo de adapter es brindar comunicación entre la interfaz del framework y el cliente, en teoría el decorador debería brindarle una interfaz mejorada, pero la mejora es que el método se convierte en un método de petición HTTP, algo que el cliente podría hacer simplemente haciendo uso del adapter cuando sea requerido.

Por otro lado, es entendible que Nest haga uso de decoradores para la realización de peticiones, ya que estos hacen parte de la filosofía de Nest como framework, y cuando se toman en cuenta los Custom Decorators y los decoradores por defecto que provee Nest para solucionan otros problemas, se entiende la decisión de abstraer todos estos comportamientos a una interfaz común como es el uso de decoradores.

Referencias

- Documentación oficial de NestJS: https://docs.nestjs.com/
- Refactoring Guru: https://refactoring.guru/es/design-patterns/
- Código fuente NestJS: https://github.com/nestjs/nest
- Documentación patrones de diseño en Java: https://github.com/iluwatar/java-design-patterns