

Incorporando Principios SOLID e Inversión de Dependencias en el Workshop de NestJS

En el desarrollo del ejercicio práctico de la aplicación de e-commerce, se pueden aplicar y explicar los principios SOLID junto con el principio de inversión de dependencias para ayudar a los participantes a escribir código limpio, modular y fácil de mantener. A continuación, te muestro cómo se pueden integrar estos conceptos en el workshop:

S - Single Responsibility Principle (SRP)

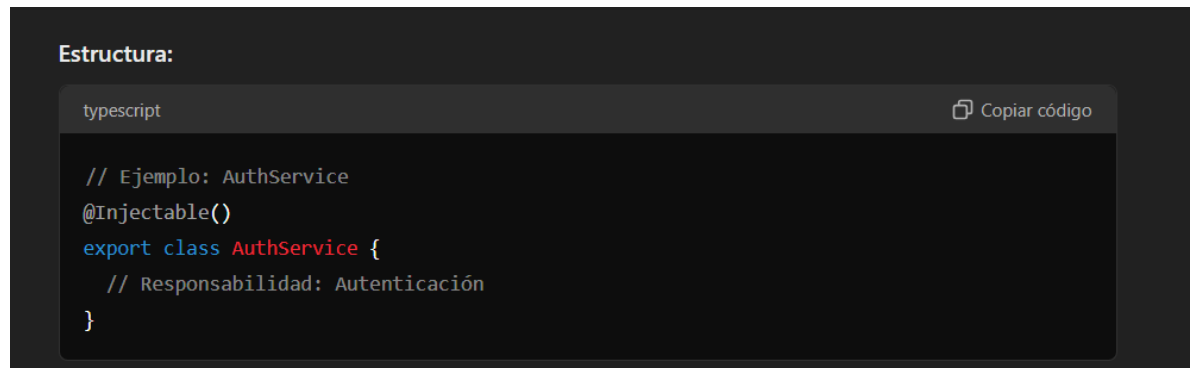
Explicación:

Cada módulo, servicio, o clase debe tener una única responsabilidad o motivo para cambiar. Es decir, un componente debe enfocarse en una tarea específica.

Aplicación en el Ejercicio:

- **Usuarios:** El UsersService se encarga exclusivamente de la lógica relacionada con la gestión de usuarios (creación, actualización, eliminación, etc.).
- **Autenticación:** El AuthService se encarga únicamente de la autenticación y no debería preocuparse por otras responsabilidades, como la gestión de pedidos o productos.

Estructura:

A screenshot of a code editor with a dark theme. The editor shows a TypeScript file named 'typescript'. The code defines an 'AuthService' class decorated with '@Injectable()'. A comment indicates its responsibility is 'Autenticación'. A 'Copiar código' button is visible in the top right corner of the code block.

```
typescript Copiar código

// Ejemplo: AuthService
@Injectable()
export class AuthService {
  // Responsabilidad: Autenticación
}
```

O - Open/Closed Principle (OCP)

Explicación:

El código debe estar abierto para la extensión, pero cerrado para la modificación. Es decir, deberíamos poder añadir nuevas funcionalidades sin alterar el código existente.

Aplicación en el Ejercicio:

- **Pipes y Filters:** Se pueden crear nuevas validaciones o filtros sin modificar los ya existentes, simplemente añadiendo nuevas clases que implementen la lógica necesaria.
- **Guards:** Si se necesita un nuevo tipo de guard (por ejemplo, para permisos adicionales), se puede crear uno nuevo sin cambiar los existentes.

Estructura:

```
typescript Copiar código

// Ejemplo: Añadir un nuevo filtro sin modificar el filtro existente
@Catch()
export class CustomExceptionFilter implements ExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    // Nueva lógica de manejo de excepciones
  }
}
```

L - Liskov Substitution Principle (LSP)

Explicación:

Las clases derivadas deben ser sustituibles por sus clases base sin alterar el comportamiento del programa. Es decir, si un componente utiliza una clase base, debe poder usar cualquier clase derivada de ella sin problemas.

Aplicación en el Ejercicio:

- **Interfaces y DTOs:** Se pueden crear diferentes DTOs que extiendan de un DTO base común, permitiendo que sean intercambiables según la necesidad sin romper la lógica de negocio.

Estructura:

```
typescript Copiar código

// Ejemplo: DTO base para creación y actualización de usuarios
export class CreateUserDto {
  @IsEmail()
  email: string;

  @IsString()
  password: string;
}

export class UpdateUserDto extends CreateUserDto {
  @IsOptional()
  @IsString()
  role?: string;
}
```

I - Interface Segregation Principle (ISP)

Explicación:

Los clientes no deben estar forzados a depender de interfaces que no usan. Es decir, en lugar de tener una interfaz grande, es mejor tener varias interfaces pequeñas y específicas.

Aplicación en el Ejercicio:

- **Servicios y Repositorios:** Crear interfaces específicas para cada tipo de operación (por ejemplo, UserRepository, ProductRepository), de modo que los servicios solo dependan de las interfaces que necesitan.

Estructura:

```
typescript Copiar código

// Ejemplo: Interfaz específica para la gestión de usuarios
export interface IUserRepository {
  findOneByEmail(email: string): Promise<User | undefined>;
  createUser(userDto: CreateUserDto): Promise<User>;
}
```

D - Dependency Inversion Principle (DIP)**Explicación:**

Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones (interfaces). Las abstracciones no deben depender de los detalles; los detalles deben depender de las abstracciones.

Aplicación en el Ejercicio:

- **Inyección de Dependencias:** Utiliza interfaces para abstraer la implementación de los repositorios o servicios, de modo que los controladores o servicios de alto nivel dependan de interfaces, no de implementaciones concretas.

Estructura:

```
typescript Copiar código

// Ejemplo: Inyección de dependencia utilizando una interfaz
@Injectable()
export class UsersService {
  constructor(
    @Inject('IUserRepository') private readonly userRepository: IUserRepository,
  ) {}

  // Lógica de negocio que depende de la abstracción IUserRepository
}
```

Integración en el Ejercicio:

1. **Registro de Usuarios (SRP, DIP):**
 - Explicar cómo el UsersService se encarga exclusivamente de la lógica de usuarios (SRP) y depende de un IUserRepository para acceder a la base de datos (DIP).
2. **Autenticación JWT (SRP, ISP):**
 - El AuthService maneja la autenticación y delega la verificación de roles a un guard que implementa una interfaz específica para ese propósito (ISP).
3. **Gestión de Productos y Pedidos (OCP, LSP):**
 - Se puede extender el sistema para agregar nuevas funcionalidades o nuevos tipos de productos sin modificar el código existente, utilizando DTOs que extiendan de clases base (LSP) y añadiendo nuevas clases que implementen la lógica adicional (OCP).
4. **Inyección de Dependencias (DIP):**
 - Mostrar cómo el UsersService, ProductsService, y OrdersService dependen de interfaces en lugar de implementaciones concretas, lo que permite cambiar la lógica interna sin afectar a los servicios que lo utilizan.