Patrones de diseño

Juan David Cruz Giraldo
Juan Sebastián Gámez Ariza
Juan Camilo López León
Alejandro Ramírez Núñez

Universidad Nacional de Colombia
Facultad de Ingeniería
Ingeniería de software I

Definición de patrón de diseño

Son soluciones establecidas para problemas recurrentes en el desarrollo de software que permiten aumentar la escalabilidad, flexibilidad y facilitar la compresión del código por otros programadores. Los patrones de diseño no son fracciones de código concretas como funciones o librerías que se copian directamente, sino plantillas que deben adaptarse al contexto específico de cada proyecto.

Patrón Singleton

Es uno de los patrones de diseño más populares, hace parte de la categoría de creacionales y se encarga de que una clase tenga una única instancia que sea reutilizada desde múltiples puntos. Este patrón soluciona principalmente dos problemas:

1. Controlar el acceso a algún recurso compartido como por ejemplo una base de datos, de forma que se mantenga consistencia entre los datos y que no se consuman recursos innecesarios.
2. Crear un único punto de acceso global al recurso para que cualquier componente pueda usarlo sin la creación de un nuevo objeto.

Uso del patrón Singleton en el proyecto:

Utilizando el ORM sequelize y el sistema de módulos de NodeJS aplicamos el patrón Singleton de forma implícita cuando creamos la instancia con la configuración adecuada de la base de datos y lo exportamos. Debido al caché de módulos creado por Node, cualquier importación del objeto desde otro archivo utilizará el objeto ya creado, esto se ve aplicado principalmente en la carpeta domain en donde importamos el orm para poder crear las entidades definidas en nuestra base de datos.

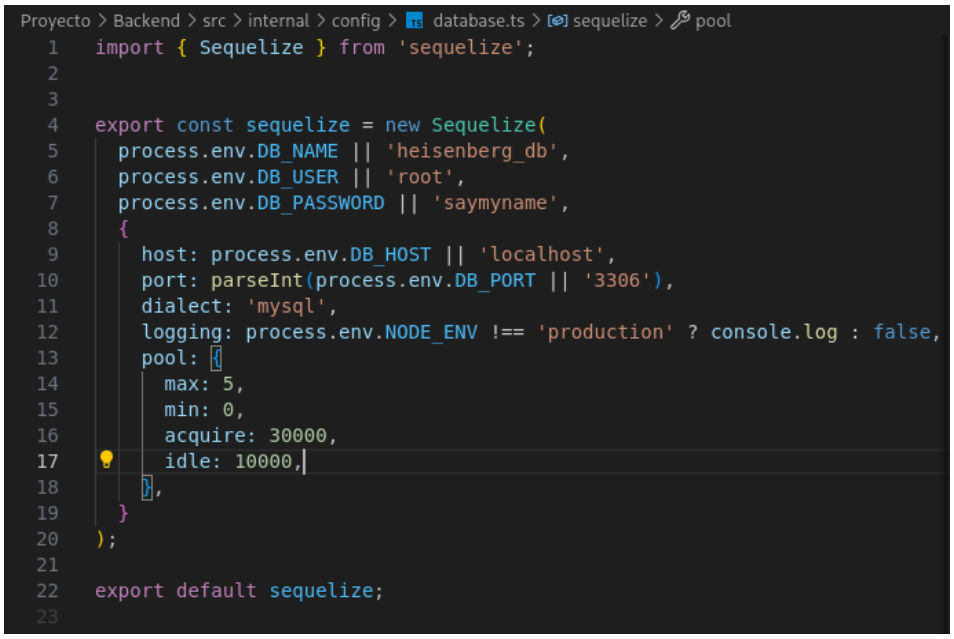

Figura 1: Creación del orm.

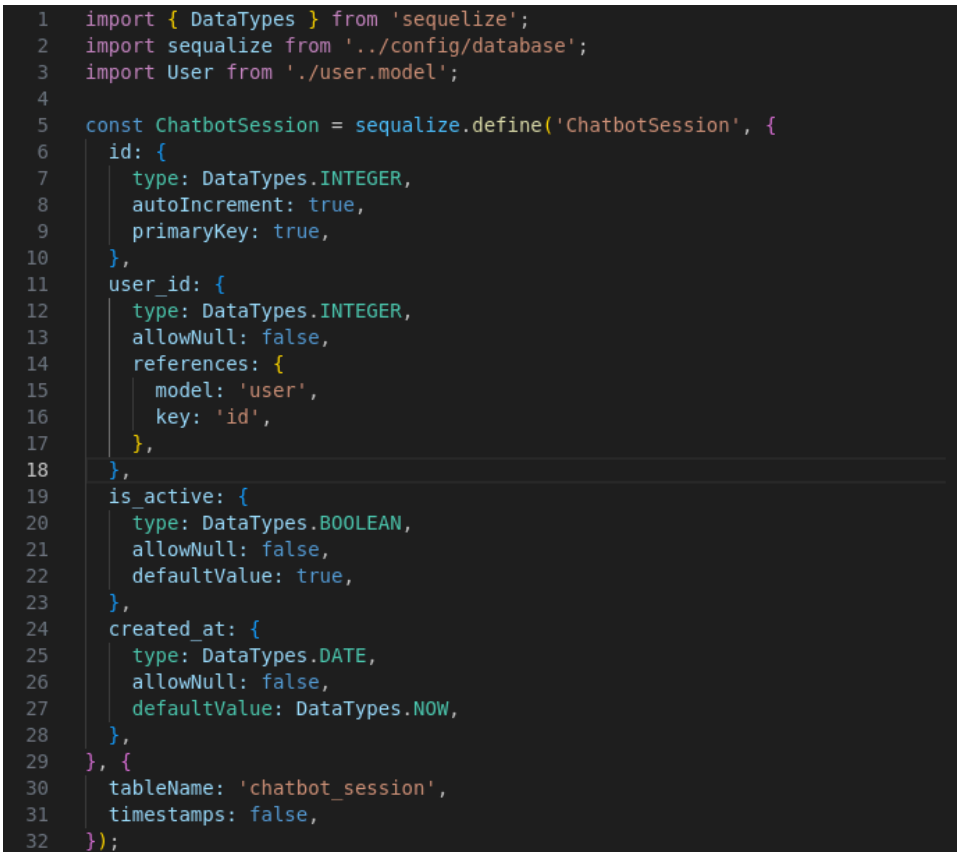

Figura 2: Llamada de sequalize desde un domain.

Patrón Factory

Es otro patrón creacional que delega la función de instanciar nuevos objetos en lugar de usar new dentro del código, con esto eliminamos la lógica de la creación y podemos cambiar fácilmente el tipo de objeto construido. Este patrón es especialmente útil para cumplir con el principio open/closed ya que podemos añadir nuevas tipos de objetos sin modificar los creados anteriormente.

Uso del patrón Factory en el proyecto:
Se utiliza para manejar la lógica de la creación de objetos de tipo repository, handler y services estableciendo una interfaz que contiene todo lo que se espera de cada uno de los objetos. Por ejemplo para un repositorio creamos la interfaz ProductRepositoryInterface, que incluye:

```typescript
export interface ProductRepositoryInterface {
    findById(id: number): Promise<ProductDTO | null>;
    findAll(limit?: number, offset?: number): Promise<ProductDTO[]>;
    create(productData: Omit<ProductDTO, 'id' | 'created_at'>): Promise<ProductDTO>;
    update(id: number, productData: Partial<Omit<ProductDTO, 'id' | 'created_at'>>):
    Promise<ProductDTO | null>;
    delete(id: number): Promise<boolean>;
    existsById(id: number): Promise<boolean>;
}
```

Figura 3: Creación de la interfaz product.

Y luego tenemos una clase fábrica o creadora que es la responsable exclusiva de ejecutar la instancia del Producto Concreto (ProductRepository), ocultando esta dependencia del resto de la aplicación.

```typescript
import { ProductRepositoryInterface, ProductRepository } from "./product.

export interface Repository {
    productRepository: ProductRepositoryInterface;
}

export const createRepository = (): Repository => {
    return {
        productRepository: new ProductRepository(),
    };
};
```

Figura 4: Creación de clase fábrica.

Finalmente, desde nuestro main del proyecto typeScript podemos crear un repositorio sin tener que usar un new:

```typescript
// Setup architecture
const repository = createRepository();
const service = createService(repository);
const handler = createHandler(service);

```