**Principios SOLID -Microservicios Transacciones**

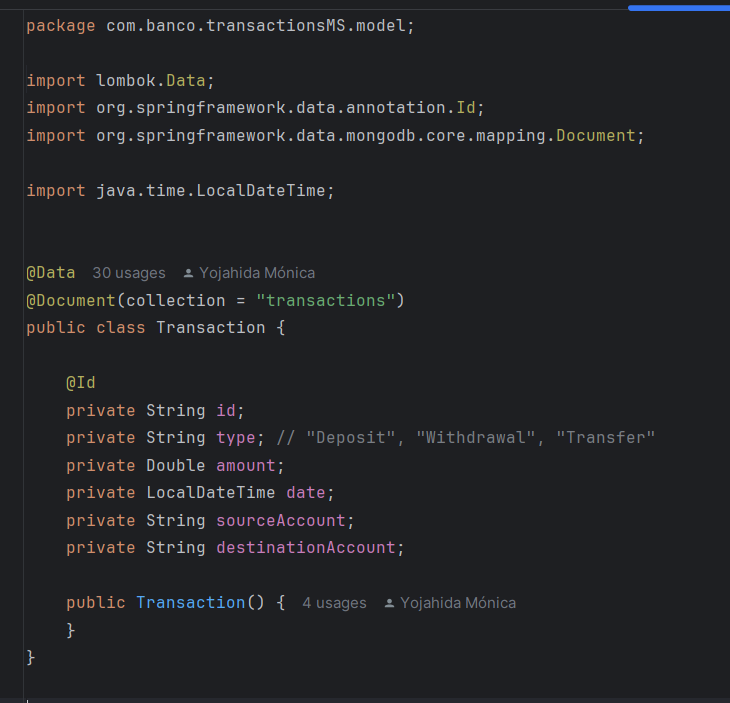
**Principio de Responsabilidad Unica(Single Responsibility Principle - SRP)**

Este principio establece que cada clase debe tener una única responsabilidad o razón para cambiar.

En el microservicio Transactions, las clases Transaction, TransactionRepository, TransactionService, TransactionController, tienen una sola responsabilidad.

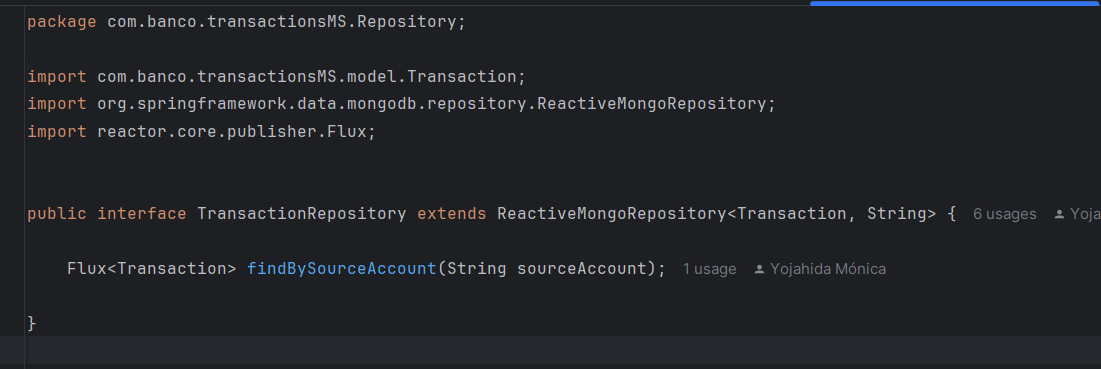
**Clase Transaction**: Solo se encarga de representar una transacción bancaria ,se usaLombok (@Data) para evitar la necesidad de escribir manualmente los métodos comunes (getters, setters, etc.). Se almacena en MongoDB como un documento de la colección "transactions".

*Clase Transaction*



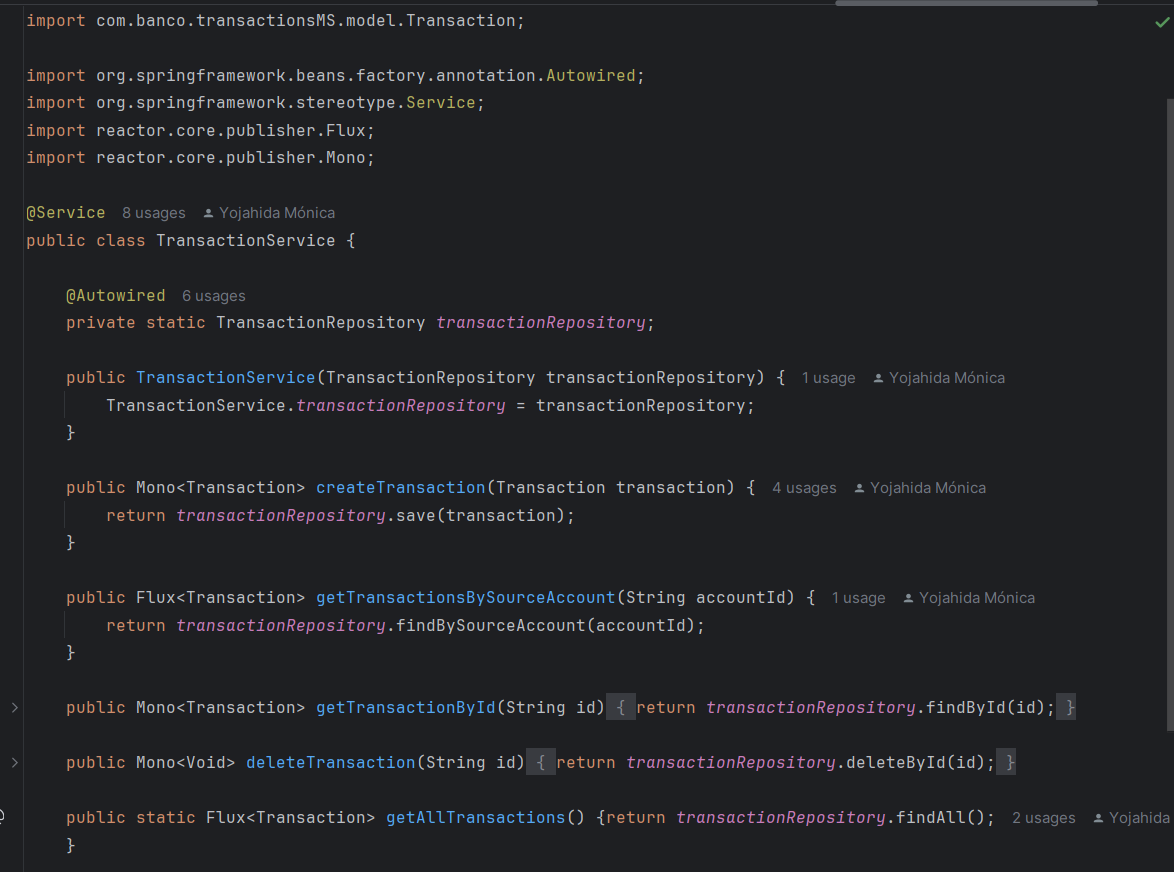
**Clase Transaction Repository.-** Su única responsabilidad es el acceso a los datos de las transacciones en MongoDB.

*Clase TransactionRepository*



**Clase TransactionService.-** El servicio encapsula la lógica de negocio relacionada con las transacciones bancarias.

*Clase TransactionService*

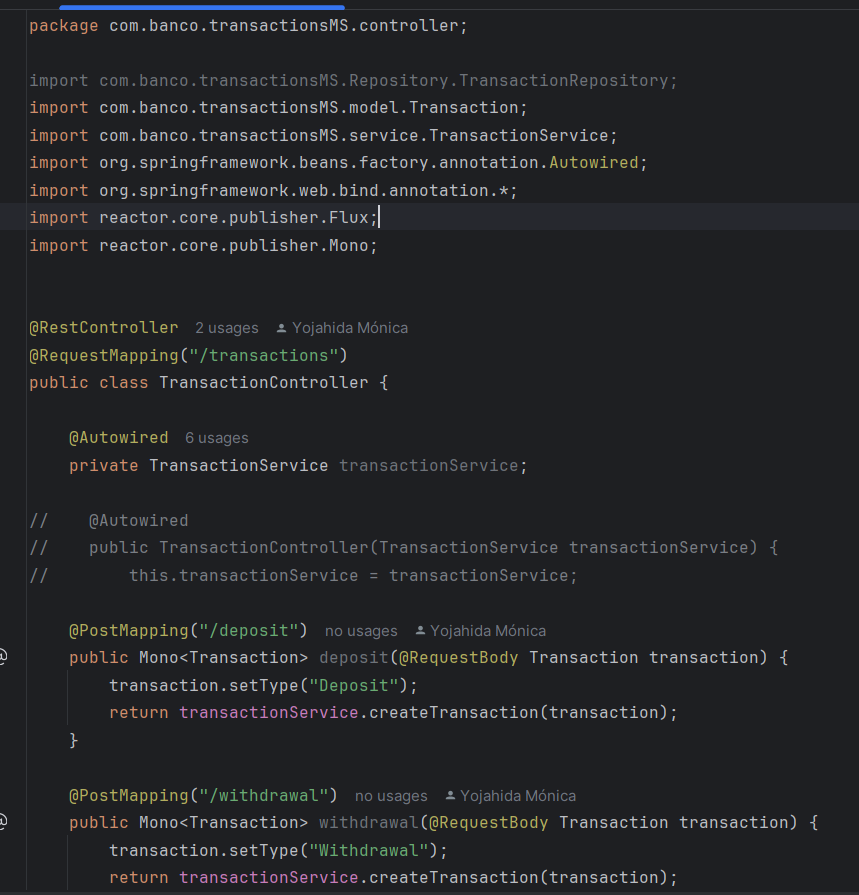
****

**Clase TransactionController**

Este controlador expone varios endpoints para gestionar las transacciones:

* /deposit: Para registrar depósitos.
* /withdrawal: Para retiros.
* /transfer: Para transferencias entre cuentas.

*Clase TransactionController*

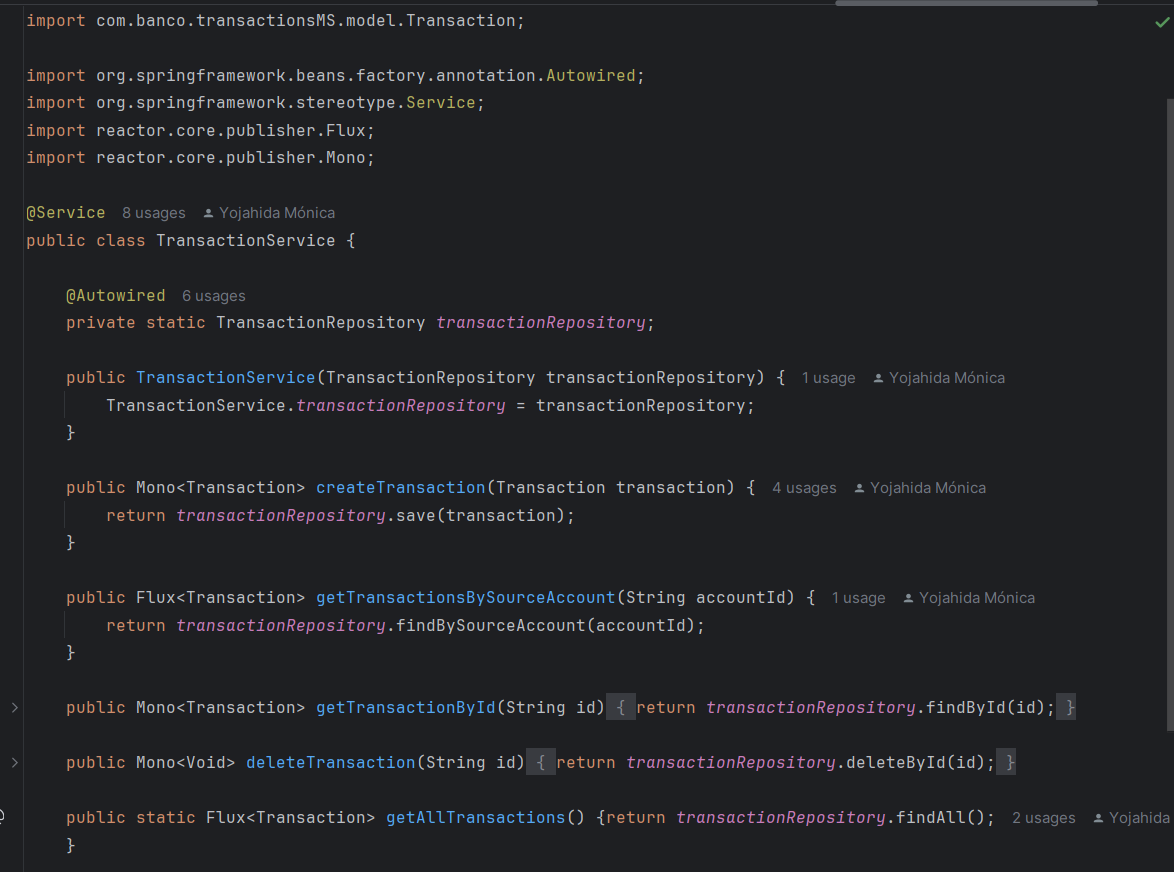
****

**Principio de abierto/cerrado(Open/Closed Principle - OCP)**

Este principio dicta que las clases deben estar abiertas para la extensión, pero cerradas para la modificación.

La clase TransactionService se puede extender para manejar tipos adicionales de transacciones sin necesidad de modificar el código existente. Por ejemplo, si quisiéramos agregar un nuevo tipo de transacción (como un pago de servicios), podríamos extender el servicio sin modificar los métodos actuales para depósitos, retiros y transferencias.

*ClaseTransactionService*

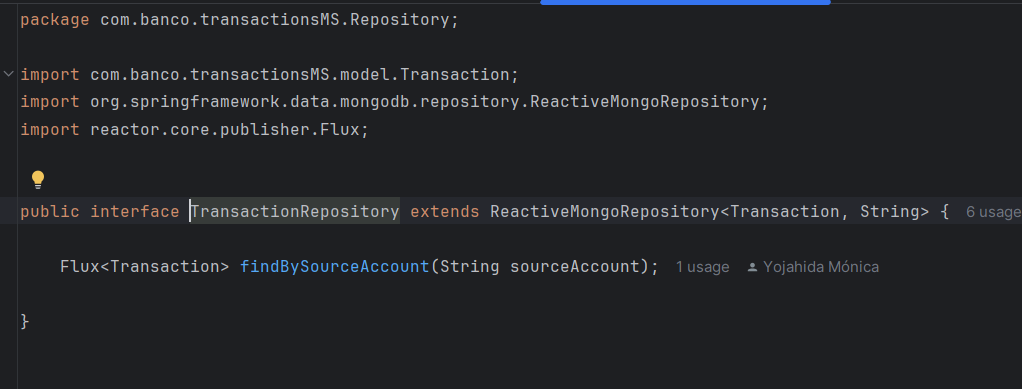
****

**Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP)**:

Este principio establece que los objetos de una clase derivada deben ser sustituibles por objetos de la clase base sin alterar el funcionamiento del sistema.

En el proyecto el uso de interfaz ReactiveMongoRepository permite que cualquier implementación de repositorio que herede esta interfaz pueda ser sustituida por otra (si se cambia la base de datos o el motor de persistencia) sin afectar al resto del sistema.

*Interface TransactionRepository*

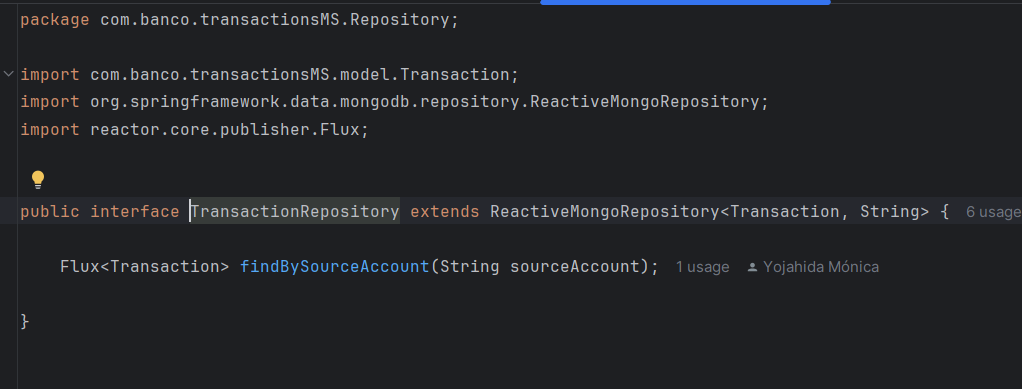
****

**Principio de segregación de interfaces(Interface Segregation Principle - ISP)**

Las interfaces deben ser pequeñas y específicas para evitar que las clases dependan de métodos que no utilizan.

La interfaz ReactiveMongoRepository está especializada en las operaciones reactivas necesarias, y no tiene métodos que no sean pertinentes para las necesidades del microservicio de transacciones.

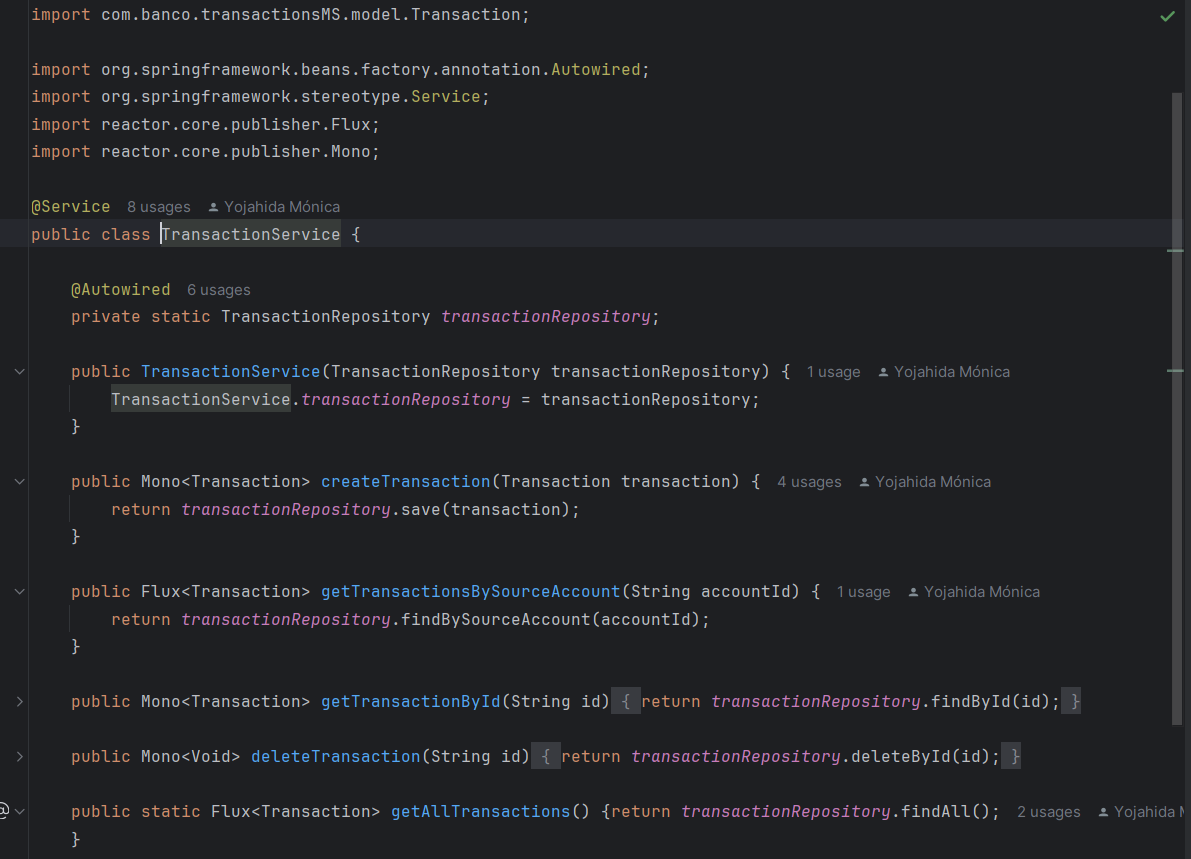
*Interface TransactionRepository*

****

**Principio de inversión de dependencias(Dependency Inversion Principle - DIP)**

Este principio establece que las clases de alto nivel no deben depender de clases de bajo nivel, sino de abstracciones (interfaces). Además, las abstracciones no deben depender de detalles, sino que los detalles deben depender de las abstracciones.

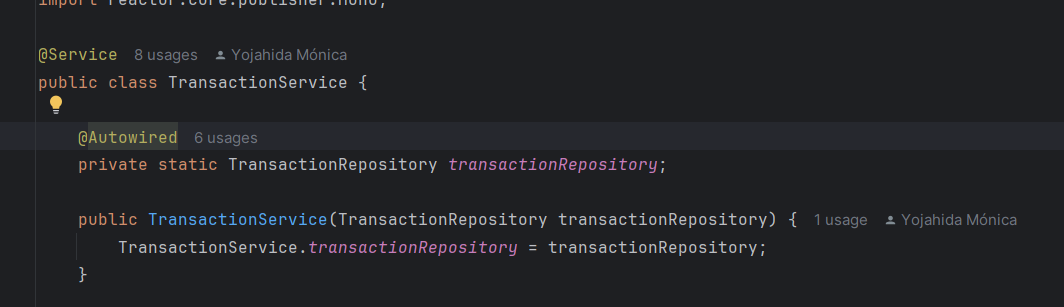
*Clase TransactionService depende de la interfaz TransactionRepository*

**

**Patrones de Diseño Aplicados**

1.- En el proyecto se utilizó el patrón de Inyección de Dependencias (Dependency Injection), este patrón utiliza para desacoplar las clases y facilitar la extensibilidad y las pruebas unitarias.

Spring Boot gestiona la inyección de dependencias automáticamente usando la anotación @Autowired.



2.- Patrón Repositorio (Repository Pattern):

El patrón de repositorio se utiliza para abstraer el acceso a la base de datos. El repositorio actúa como una capa de persistencia que oculta los detalles del almacenamiento de datos.

La clase TransactionRepository implementa este patrón al gestionar las operaciones de persistencia con MongoDB.

3.- Patrón de Pruebas con Mock (Mock Object Pattern):

Se usa el Patrón de Mock para crear versiones simuladas de clases que el código bajo prueba depende de ellas. En el proyecto se utilizó Mockito para simular la interacción con el repositorio y no depender de la base de datos real durante las pruebas.