Spring Boot @Transactional Annotation Example

Author: Ramesh Fadatare

SPRING ANNOTATIONS SPRING BOOT

The `@Transactional` annotation in Spring Boot is used to manage database operations within a Spring application, ensuring that they are done safely and correctly. Applying this annotation to a method or an entire class tells Spring to start a transaction whenever the method starts and commit the transaction when the method finishes. If anything goes wrong (like an error or exception), Spring automatically undoes all the operations performed during the transaction, known as a rollback.

Example Use Case

Imagine you're building a banking application where users can transfer money between accounts. You'd have a method to perform the transfer, which might involve several steps:

- Check if the sender's account has enough balance.
- Deduct the amount from the sender's account.
- Add the amount to the receiver's account.

With `@Transactional`, you can ensure that all these steps are completed successfully or fail. If something goes wrong during the transfer (like a system crash or network issue), the annotation helps ensure that the transaction is rolled back, leaving the accounts as they were before the transaction started. This prevents issues like money being deducted from one account but not added to another.

Key Points about Spring @Transactional Annotation

Transactional Scope:

The `@Transactional` annotation can be applied at both the class and method levels. When applied at the class level, all the public methods in the class will run within a transactional context. When applied at the method level, only the specific method will be transactional. This provides flexibility in managing transactional behavior based on the granularity needed.

Propagation Behavior:

Propagation settings determine how transactions relate to each other.
The `@Transactional` annotation supports various propagation types, such as:

- `REQUIRED (the default)`: Supports a current transaction; creates a new one if none exists.
- `REQUIRES_NEW`: Always creates a new transaction and suspends any existing transaction.
- `SUPPORTS`: Uses a current transaction if it exists; otherwise, executes non-transactionally.
- `NEVER`: Ensures no transaction exists, throwing an exception if a transaction is present.

These options help manage complex transaction scenarios and ensure business processes execute with the correct transactional support.

Rollback Behavior:

By default, transactions are rolled back for runtime exceptions (unchecked exceptions) and errors. However, you can customize this behavior using the `rollbackFor` and `noRollbackFor` attributes of the @Transactional annotation.

`rollbackFor`: Specifies which exception types should cause the transaction to roll back. This can include checked exceptions, which are not typically rolled back by default.

`noRollbackFor`: Specifies exceptions for which the transaction should not be rolled back. This can be useful when certain operations within the transaction can tolerate specific exceptions without failing the entire transaction.

Isolation Levels:

The `@Transactional` annotation allows you to define the transaction's isolation level, which is crucial for preventing issues such as dirty reads, non-repeatable reads, phantom reads, and lost updates. Common isolation levels include:

- `READ_UNCOMMITTED`: Allows dirty reads, where one transaction can see uncommitted changes made by another.
- `READ_COMMITTED`: Prevents dirty reads by ensuring a transaction only reads data that has been committed.
- `REPEATABLE_READ`: Prevents dirty reads and non-repeatable reads by ensuring a transaction reads rows in a consistent state.
- `SERIALIZABLE`: Provides the highest level of isolation by ensuring that transactions are completely isolated.

Each level offers different trade-offs between consistency and performance, and the choice of isolation level can significantly impact application behavior.

Read-Only Setting:

Marking a transaction as `readOnly` suggests that the transaction will only read but not modify data. This is a hint to the transaction manager and the underlying persistence provider, which can optimize performance, for instance, by reducing locking and improving query performance.

In Spring, this is achieved by setting the `readOnly` attribute to true. This setting is particularly useful for transactions that involve complex queries and large amounts of data that do not require modification.

Together, these features make @Transactional a robust tool for managing transactional logic in enterprise applications, providing developers with fine-grained control over transaction behaviors and helping ensure that data integrity is maintained throughout the application.

Example 1

```
// Step 1: Entity class
import jakarta.persistence.*;

@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double price;

    // Constructors, Getters and Setters
}

// Step 2: Repository interface
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product, Long> {
}

// Step 3: Service class with class-level @Transactional
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.beans.factory.annotation.Autowired;

@Transactional
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    // Read-only transactional method
    @Transactional(readOnly = true)
    public Product findProductById(Long id) {
        return productRepository.findById(id).orElseThrow();
    }

    // Method without read-only despite class-level @Transactional
    public void updateProductPrice(Long id, double newPrice) {
        Product product = findProductById(id);
        product.setPrice(newPrice);
        productRepository.save(product);
    }
```

```
    // Class-level transaction applies here
    public void addProduct(Product product) {
        productRepository.save(product);
    }
}
```

Explanation:

1. The `Product` entity represents a simple product with an ID, name, and price.

2. `ProductRepository` is a standard Spring Data repository for CRUD operations on `Product` entities.

3. `ProductService` is annotated with `@Transactional` at the class level, setting a default transactional context for all methods:

- `findProductById` is explicitly marked as `readOnly = true`. Read-only transactions are optimized for performance as they avoid the overhead associated with dirty checking and flush operations, which is suitable for methods that only read data.

- `updateProductPrice` updates the price of a product. It isn't marked as read-only, which allows it to override the class-level read-only setting (if it were set) and supports write operations.

- `addProduct` benefits from the class-level `@Transactional` annotation, ensuring that new products are added within a transactional context even without method-level annotations.

Example 2 - @Transactional with Propagation

```
// Step 1: Entity class
import jakarta.persistence.*;

@Entity
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double balance;
```

```
    // Constructors, Getters and Setters
}

// Step 2: Repository interface
import org.springframework.data.jpa.repository.JpaRepository;

public interface AccountRepository extends JpaRepository<Account, Long> {
}

// Step 3: Service class with @Transactional methods
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.beans.factory.annotation.Autowired;

@Service
public class AccountService {

    @Autowired
    private AccountRepository accountRepository;

    // Basic transactional method
    @Transactional
    public void deposit(Long id, double amount) {
        Account account = accountRepository.findById(id).orElseThrow();
        account.setBalance(account.getBalance() + amount);
        accountRepository.save(account);
    }

    // Transactional with specific propagation
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void withdraw(Long id, double amount) {
        Account account = accountRepository.findById(id).orElseThrow();
        if(account.getBalance() < amount) {
            throw new RuntimeException("Insufficient funds");
        }
        account.setBalance(account.getBalance() - amount);
        accountRepository.save(account);
    }

    // Transactional method with rollback configuration
    @Transactional(rollbackFor = Exception.class)
    public void riskyTransfer(Long fromId, Long toId, double amount) {
        withdraw(fromId, amount);
        if(amount > 10000) {
            throw new RuntimeException("High value transfer alert");
        }
        deposit(toId, amount);
    }
}
```

Explanation:

1. The *`Account`* entity represents a bank account with an ID and balance.

2. `AccountRepository` is a typical Spring Data repository for handling CRUD operations on `Account` entities.

3. `AccountService` contains several methods annotated with `@Transactional`. Each method demonstrates different aspects of transactional behavior:

- `deposit` is a straightforward example of a transactional method ensuring that the deposit operations are committed or rolled back atomically.

- `withdraw` uses a `REQUIRES_NEW` propagation to always start a new transaction, useful when the method needs to be independent of any ongoing transaction.

- `riskyTransfer` is designed to demonstrate rollback behavior; it throws an exception for high value transfers, which triggers a rollback of the entire transaction, including the withdrawal, unless handled properly.