15-Modifying & Transactional

Author: Vincent Lau

Learning Objectives

> Include the correct dependencies to set up MySQL/ PostgreSQL database connection.

> Include the correct application yml configuration to setup MySQL/ PostgreSQL.

> Ways to initialize the database, schema & data.

Modifying

Introduction

The `@Modifying` annotation is typically used in the context of Java Persistence API (JPA) and Spring Data to indicate that a method is intended to perform write or update operations on a database. This annotation is often used in combination with the `@Query` annotation to specify custom SQL or JPQL (Java Persistence Query Language) statements for database manipulation.

Key points and usages

1. **Database Write Operations:** The `@Modifying` annotation is primarily used for methods that modify the state of the database, such as inserting, updating, or deleting records. These are often referred to as write operations. These operations typically change the data in the database and have an effect on its state.
2. **JPA and Spring Data:** `@Modifying` is commonly used in JPA and Spring Data repositories, which provide a convenient way to interact with databases. When used with the `@Query` annotation, you can define custom SQL or JPQL statements to perform the desired database write operations.
3. **Transactional Behavior:** To ensure consistency, database write operations should often be executed within a database transaction. You can use the `@Transactional` annotation in conjunction with `@Modifying` to specify the transactional behavior of the method, ensuring that the changes are either committed or rolled back as a single unit of work.

Example of @Modifying & @Query

In this example, the `updateStock` method is annotated with `@Modifying` and `@Query`. It performs an update operation on the database to change the stock of a product. The `@Modifying` annotation indicates that this method is a database-modifying operation.

```java
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {

    @Modifying
    @Query("UPDATE Product p SET p.stock = :newStock WHERE p.id = :productId")
    void updateStock(@Param("productId") Long productId, @Param("newStock") int newStock);

}
```

When you invoke this method, it will execute an SQL UPDATE statement based on the query provided in the `@Query` annotation, and the changes will be made to the database. This method can be used in a service or business logic class, and you can annotate it with `@Transactional` to ensure that it runs within a transaction.

Summary

In summary, the `@Modifying` annotation is used to mark methods that perform write operations on a database. When used in combination with `@Query`, it allows you to define custom SQL or JPQL statements for database manipulation. This annotation is particularly useful when working with JPA and Spring Data repositories to maintain data integrity during database updates.

Transactional

Introduction

In software development, the `@Transactional` annotation is commonly used in the context of object-relational mapping (ORM) frameworks, such as Java's Hibernate or Spring Framework, to manage database transactions. A database transaction is a series of one or more database operations (such as inserts, updates, or deletes) that are treated as a single, indivisible unit of work. The `@Transactional` annotation is a powerful tool that helps ensure the consistency and integrity of the database by providing a declarative way to define transactional behavior in your application code.

Key Usages

1. **Defining Transaction Boundaries:** When you annotate a method or a class with `@Transactional`, you specify that the annotated method or all methods within the annotated class should be executed within a database transaction. The boundaries of the transaction are established by the annotated method's start and end.
2. **Transactional Attributes:** The `@Transactional` annotation typically allows you to set various attributes to control transactional behavior. These attributes can include isolation level, propagation behavior, rollback rules, and more. These attributes help you customize how transactions should behave for a particular method or class.
3. **Isolation Level:** The isolation level defines the level of visibility of uncommitted changes to other transactions. Common isolation levels include READ_COMMITTED, READ_UNCOMMITTED, SERIALIZABLE, etc.
4. **Propagation:** Propagation defines how a new transaction behaves concerning an existing transaction. For example, whether the new transaction should create a new one or join an existing one.
5. **Rollback Rules:** You can specify rules for when the transaction should be rolled back, typically based on exceptions thrown during the method's execution. This allows you to handle exceptions and control whether the transaction should be committed or rolled back.
6. **Spring Framework:** In the context of the Spring Framework, the `@Transactional` annotation is widely used to enable declarative transaction management. It simplifies the process of handling transactions, making the code cleaner and more maintainable.

Code Examples

An account transfer operation is a great example to illustrate the use of the `@Transactional` annotation. In this scenario, you want to ensure that funds are transferred atomically between two accounts. If any step in the process fails (e.g., deducting funds from the source account or crediting funds to the destination account), the entire transaction should be rolled back to maintain data consistency.

Here's an example of how you can implement an account transfer using `@Transactional` in a Spring service:

```java
@Service
public class AccountService {

    @Autowired
    private AccountRepository accountRepository;

    @Transactional
    public void transferFunds(Long sourceAccountId, Long destinationAccountId, double amount) {
        try {
            Account sourceAccount = accountRepository.findById(sourceAccountId).orElse(null);
            Account destinationAccount = accountRepository.findById(destinationAccountId).orElse(null);

            if (sourceAccount != null && destinationAccount != null) {
                // Check if the source account has sufficient balance
                if (sourceAccount.getBalance() >= amount) {
                    // Deduct funds from the source account
                    sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                    accountRepository.save(sourceAccount);

                    // Credit funds to the destination account
                    destinationAccount.setBalance(destinationAccount.getBalance() + amount);
                    accountRepository.save(destinationAccount);
                } else {
                    throw new InsufficientFundsException("Insufficient funds in the source account.");
                }
            } else {
                throw new AccountNotFoundException("One or both accounts not found.");
            }
        } catch (Exception e) {
            // Handle exceptions and possibly log errors.
        }
    }
}

```

1. The `transferFunds` method in the service layer is annotated with `@Transactional`. This ensures that the entire transfer operation is executed within a single database transaction.
2. Inside the method, it first retrieves the source and destination accounts from the database using their IDs.
3. It checks if the source account has sufficient balance to perform the transfer. If not, an `InsufficientFundsException` is thrown.
4. If both accounts are found and the source account has sufficient balance, it deducts the amount from the source account and credits it to the destination account. Both of these operations involve database updates.
5. If any exceptions occur during the method's execution, the entire transaction will be rolled back, ensuring data consistency. For example, if an exception is thrown when saving the source account or destination account, the changes to both accounts will be undone.

By using the `@Transactional` annotation, you guarantee that the account transfer operation is atomic, and the database remains in a consistent state, even in the presence of exceptions.

Summary

In summary, the `@Transactional` annotation is a valuable tool for managing database transactions in a declarative and flexible manner, reducing the complexity of handling transactions manually and improving the reliability and maintainability of your database-related code.