### Handling Transactions in Spring Boot (Declarative Transaction Management)

In Spring Boot, \*\*transaction management\*\* is essential to ensure data consistency, particularly when dealing with multiple database operations that should either all succeed or all fail. \*\*Declarative transaction management\*\* allows you to handle transactions without having to explicitly manage transaction boundaries in your code.

Spring’s transaction management is supported using the `@Transactional` annotation, which allows you to define transaction boundaries declaratively at the method or class level.

---

### 1. \*\*What is a Transaction?\*\*

A \*\*transaction\*\* is a sequence of one or more SQL operations treated as a single logical unit. If any operation in the sequence fails, the entire transaction should fail (rollback). If all operations succeed, the transaction is committed.

Transactions must adhere to the \*\*ACID\*\* properties:

- \*\*Atomicity\*\*: All operations must succeed or fail as a unit.

- \*\*Consistency\*\*: Data must remain consistent after the transaction.

- \*\*Isolation\*\*: Transactions should not affect each other.

- \*\*Durability\*\*: Once a transaction is committed, the data must be permanent.

---

### 2. \*\*Enable Transaction Management in Spring Boot\*\*

By default, Spring Boot supports transaction management, but you must annotate your methods or classes with `@Transactional` to specify where the transactional boundaries are.

To explicitly enable transaction management, you can add the `@EnableTransactionManagement` annotation in one of your configuration classes.

```java

import org.springframework.context.annotation.Configuration;

import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration

@EnableTransactionManagement

public class TransactionManagementConfig {

// This enables declarative transaction management

}

```

---

### 3. \*\*Using `@Transactional` for Declarative Transaction Management\*\*

The `@Transactional` annotation can be applied to methods or classes to manage the transaction boundaries automatically. When applied to a class, it applies to all methods within that class.

#### \*\*Basic Usage of `@Transactional`:\*\*

```java

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

// A transactional method to create a user and update another user atomically

@Transactional

public void createAndUpdateUser(User newUser, User existingUser) {

// Create new user

userRepository.save(newUser);

// Update an existing user

existingUser.setEmail("updatedEmail@example.com");

userRepository.update(existingUser);

// If any exception occurs, the whole transaction will be rolled back

}

}

```

- If any exception is thrown during the execution of `createAndUpdateUser()`, both the save and update operations will be rolled back.

---

### 4. \*\*Transaction Propagation\*\*

\*\*Propagation\*\* defines how transaction boundaries should behave in nested transactional method calls. Spring provides different propagation options:

- \*\*`REQUIRED`\*\* (default): Joins an existing transaction or creates a new one if none exists.

- \*\*`REQUIRES\_NEW`\*\*: Suspends any existing transaction and creates a new one.

- \*\*`MANDATORY`\*\*: Requires an existing transaction, throws an exception if none exists.

- \*\*`SUPPORTS`\*\*: Joins an existing transaction, but does not create a new one if none exists.

- \*\*`NOT\_SUPPORTED`\*\*: Executes the method without a transaction.

- \*\*`NEVER`\*\*: Throws an exception if a transaction exists.

- \*\*`NESTED`\*\*: Executes within a nested transaction if one exists.

#### \*\*Example of Transaction Propagation:\*\*

```java

@Transactional(propagation = Propagation.REQUIRED)

public void outerTransaction() {

// Outer transaction

innerTransaction(); // Will participate in the same transaction

}

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void innerTransaction() {

// New transaction that will be independent of the outer transaction

}

```

- In the example above, `outerTransaction` uses `REQUIRED` propagation, meaning it will use a single transaction boundary. `innerTransaction`, on the other hand, uses `REQUIRES\_NEW`, meaning it will start a new transaction separate from the outer one.

---

### 5. \*\*Rollback Scenarios in Transactions\*\*

Spring's `@Transactional` annotation rolls back the transaction in the case of \*\*unchecked exceptions\*\* (runtime exceptions). However, it will \*\*not\*\* roll back in case of \*\*checked exceptions\*\* (like `SQLException`) unless you specify it explicitly.

#### \*\*Rollback on Checked Exceptions:\*\*

To force a rollback on checked exceptions, you can configure the `@Transactional` annotation:

```java

@Transactional(rollbackFor = Exception.class)

public void performTask() throws Exception {

// Code that might throw an Exception

}

```

- `rollbackFor = Exception.class` ensures that the transaction is rolled back even if a checked exception occurs.

#### \*\*Rollback on Specific Exceptions:\*\*

You can specify multiple exceptions for rollback as follows:

```java

@Transactional(rollbackFor = {CustomException.class, IOException.class})

public void performTask() throws CustomException, IOException {

// Code that might throw these exceptions

}

```

---

### 6. \*\*Transaction Isolation Levels\*\*

The \*\*isolation level\*\* determines how one transaction is isolated from others. The default isolation level in Spring Boot is \*\*`Isolation.DEFAULT`\*\*, which means it uses the database’s default isolation level.

Isolation levels include:

- \*\*`READ\_UNCOMMITTED`\*\*: Allows dirty reads (least isolated).

- \*\*`READ\_COMMITTED`\*\*: Prevents dirty reads (default in most databases).

- \*\*`REPEATABLE\_READ`\*\*: Prevents dirty reads and non-repeatable reads.

- \*\*`SERIALIZABLE`\*\*: Most restrictive, ensures full isolation (no dirty, non-repeatable, or phantom reads).

#### \*\*Setting Isolation Level:\*\*

```java

@Transactional(isolation = Isolation.SERIALIZABLE)

public void performSerializableTransaction() {

// Transaction will be executed with the SERIALIZABLE isolation level

}

```

---

### 7. \*\*Read-Only Transactions\*\*

For transactions that only read data (e.g., `SELECT` operations), you can optimize performance by marking them as read-only. This signals to the database that no data modifications will be made.

#### \*\*Example:\*\*

```java

@Transactional(readOnly = true)

public List<User> getAllUsers() {

return userRepository.findAll();

}

```

- By marking the transaction as `readOnly`, Spring Boot can optimize the transaction handling, and the database may skip certain locks.

---

### 8. \*\*Using Programmatic Transaction Management (Optional)\*\*

While declarative transaction management is most commonly used, Spring also supports programmatic transaction management using the `PlatformTransactionManager` API. However, it’s more verbose and rarely needed unless you require fine-grained control.

#### Example of Programmatic Transaction Management:

```java

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.TransactionDefinition;

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.support.DefaultTransactionDefinition;

public class UserService {

private final PlatformTransactionManager transactionManager;

public UserService(PlatformTransactionManager transactionManager) {

this.transactionManager = transactionManager;

}

public void executeWithProgrammaticTransaction() {

DefaultTransactionDefinition def = new DefaultTransactionDefinition();

def.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);

TransactionStatus status = transactionManager.getTransaction(def);

try {

// Perform database operations

transactionManager.commit(status);

} catch (Exception ex) {

transactionManager.rollback(status);

throw ex;

}

}

}

```

---

### 9. \*\*Testing Transactions\*\*

When testing transactional methods, you can use `@Transactional` at the test level to ensure that the database remains in a consistent state after each test.

#### Example of Testing with `@Transactional`:

```java

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.annotation.Rollback;

import org.springframework.transaction.annotation.Transactional;

@SpringBootTest

public class UserServiceTest {

@Autowired

private UserService userService;

@Test

@Transactional

@Rollback(true)

public void testCreateUser() {

User user = new User();

user.setName("Test User");

user.setEmail("test@example.com");

userService.createUser(user);

// Assert user was created in the database

// After the test, the transaction will roll back due to @Rollback

}

}

```

---

### Conclusion

Spring Boot’s declarative transaction management using `@Transactional` simplifies handling transactions. By controlling rollback behavior, transaction propagation, and isolation levels, you can ensure that database operations are performed atomically and consistently.