Best Practices for @Transactional

https://medium.com/thefreshwrites/best-practices-for-using-transactional-in-spring-boot-7baafba397ad

https://www.baeldung.com/spring-transactional-propagation-isolation

The instruction to avoid marking tests with `@Transactional` and to clean the database after each test is to ensure test isolation and prevent conflicts between test methods.

Here's why it's important:

1. **Test Isolation**: By default, Spring Boot automatically rolls back transactions at the end of each test method when `@Transactional` is used. This ensures that changes made during a test method do not affect other tests, maintaining test isolation. However, if you're not using `@Transactional`, changes made in one test method may affect the database state for subsequent tests, leading to unpredictable behavior and test failures.

2. **Preventing Side Effects**: Without cleaning the database after each test, changes made during one test method may persist and affect the execution of subsequent test methods. For example, if one test method inserts data into the database but does not clean up afterward, the inserted data may interfere with the execution of other test methods, leading to incorrect test results.

To address these concerns, it's recommended to follow these best practices when writing unit tests:

- **Avoid Using `@Transactional`**: If you're writing unit tests for repository or service layer components, try to avoid using `@Transactional` unless it's necessary to test transactional behavior. Instead, manage transactions manually within your test methods to ensure better control over transaction boundaries.

- **Clean Database After Each Test**: Always clean up the database after each test method to ensure a clean state for subsequent tests. You can achieve this by executing database cleanup logic (e.g., deleting records, truncating tables) in an `@AfterEach` or `@After` method annotated with JUnit.

By following these practices, you can write unit tests that are more predictable, maintainable, and less prone to conflicts or side effects caused by database state changes.

**important

When using `@SpringBootTest`, each test method typically runs within its own transaction, and the changes made within one test method are not automatically propagated to subsequent test methods. However, depending on how the transaction management and data initialization are configured, there are scenarios where changes made in one test method might affect subsequent test methods:

1. **Transactional Behavior**: If tests are annotated with `@Transactional`, changes made within a test method may be rolled back at the end of the test method execution. However, if tests share the same transaction (e.g., due to class-level `@Transactional` annotation), changes might be visible to subsequent test methods.

2. **Database State**: If tests rely on a shared database state and do not properly reset or clean up the database between tests, changes made in one test method may persist and affect subsequent test methods.

3. **Data Initialization**: If tests use data initialization mechanisms (e.g., data loaders, test fixtures) to populate the database before tests run, changes made to the initialized data within one test method may affect subsequent test methods if the data is not reset or reinitialized between tests.

To ensure that each test method runs in isolation and with its own data:

- **Avoid Shared Transactions**: Minimize the use of class-level `@Transactional` annotations and instead annotate individual test methods with `@Transactional` only when necessary.

- **Reset Database State**: Use setup and teardown methods (e.g., `@BeforeEach`, `@AfterEach`) to reset the database state between tests. This can involve clearing the database or reverting changes made during test execution.

- **Isolate Data Initialization**: If tests rely on pre-existing data, ensure that the data initialization mechanism properly isolates data between tests or reinitializes data before each test method runs.

By carefully managing transactions, database state, and data initialization, you can ensure that each test method runs independently and with predictable behavior, regardless of changes made in other test methods.