JPA and Hibernate Advanced

☑ Favorite	☐
↗ Tag	Master Hibernate & JPA with Spring Boot

▼ @DirtiesContext

```
@Test
    @DirtiesContext
    public void deleteById_basic() {
        repository.deleteById(10002L);
        assertNull(repository.findById(10002L));
    }
```

The "DirtiesContext" annotation in Spring is used to indicate that a particular unit test modifies the state of the application's data. By annotating a test method with `@DirtiesContext`, Spring ensures that the application context is reset after the test is executed. This is crucial for maintaining the integrity of subsequent tests that may rely on the initial state of the data.

Consider a scenario where a test deletes certain data that other tests depend on. Without resetting the context, subsequent tests may fail because they expect the data to be present. By using `@DirtiesContext`, Spring automatically resets the context after the test, ensuring that subsequent tests are not affected by the modifications made during the test execution.

In summary, `@DirtiesContext` is used to indicate that a test modifies the application's data state, and Spring resets the context to maintain consistency for subsequent tests. This ensures that tests remain isolated and independent of each other's side effects on the application state.

▼ JPQL vs SQL

- In SQL we query from tables

- In JPQL we query from Entities
    - Queries are converted to SQL queries by JPA

▼ Fetchtype.LAZY

In JPA (Java Persistence API), the `fetch` attribute of the `@OneToOne` annotation specifies how the associated entity should be loaded from the database when the owning entity is retrieved.

When `fetch = FetchType.LAZY` is used, it means that the associated entity (`Passport` in this case) will not be loaded from the database until it is explicitly accessed or referenced by the application code. This can help improve performance by avoiding unnecessary database queries to fetch related entities when they are not immediately needed.

In contrast, when `fetch = FetchType.EAGER` is used, it means that the associated entity will be loaded eagerly along with the owning entity. This can lead to performance issues if the associated entity contains a large amount of data or if it's not always needed when retrieving the owning entity.

So, using `fetch = FetchType.LAZY` is often preferred when dealing with associations that may not always be needed or when performance optimization is a concern.

▼ Default Fetching Strategy

In JPA relationships, the default fetching strategy varies depending on the type of relationship:

1. **One-to-Many**: By default, fetching is lazy. This means that related entities are not loaded from the database until they are explicitly accessed.
2. **Many-to-One**: Fetching is always eager by default. This means that related entities are loaded from the database immediately when the owning entity is fetched.
3. **Many-to-Many**: By default, fetching is lazy. Related entities are not loaded until they are explicitly accessed.

To remember the default fetching strategies:

- For relationships ending in "one" (such as One-to-One or Many-to-One), fetching is eager.
- For relationships ending in "many" (such as One-to-Many or Many-to-Many), fetching is lazy.

Understanding these default fetching behaviors is essential for effectively managing data retrieval and optimizing performance in JPA applications.

▼ PersistenceContext

- **Workspace for Database Operations**: A persistence context serves as a workspace where database operations are performed within a software application. It's like a container that manages the lifecycle of objects retrieved from or persisted to a database.
- Imagine you have a desk where you keep all your school stuff neatly organized. Let's call this desk your "persistence context."

  Now, let's say you have two books: one is your "Student" book, and the other is your "Passport" book. These books represent information about you and your passport.

  When you're doing some work, like studying or writing an essay, you'll take out your Student book from your desk (persistence context) and use it. While you're using it, you might need information from your Passport book, so you'll take that out too and keep it on your desk.

  Now, whenever you write something new in either of these books, you're actually making changes to them. And because they're on your desk (in the persistence context), you can easily see and track these changes.

  So, the persistence context is like your desk where you keep all the important stuff (your entities or books) while you're working on them. It helps you keep track of any changes you make and makes it easy to save those changes when you're done working.