Testing Spring Data JPA Repositories

@DataJpaTest and TestEntityManager, only named in "Testing in Spring MVC" -- closing the gap this project's own Mockito-mocked service tests leave open (nothing verifies a query actually works); testing a derived query method and a custom @Query, and embedded test database vs. real PostgreSQL vs. Testcontainers, grounded in this project's own application-test.yml comment. The 9th and final lesson in the Spring Data JPA category.

Advanced 35 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/testing-spring-data-jpa

---

"Testing in Spring MVC" named `@DataJpaTest` once, in passing, as `@WebMvcTest`'s sibling slice-test annotation — and never came back to it. This category's own service tests (`QuizServiceTest`, `QuestionIngestServiceTest`, and the rest) all mock every repository with Mockito, which is exactly right for testing a service's own logic in isolation — but it leaves a real gap this closing lesson fills: nothing anywhere in this project actually verifies that a derived query method, a hand-written `@Query`, or a `Specification` genuinely does what its name or JPQL claims.

Why Mocking a Repository Isn't Enough

A mocked repository tells you exactly what you told it to return — nothing more.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.when;

// This project's own real service tests (QuizServiceTest,
// QuestionIngestServiceTest, and the rest) mock every repository with
// Mockito, exactly like this -- a perfectly good way to test a SERVICE's
// own logic in isolation. But it has a blind spot worth being explicit
// about.
interface TopicRepositoryMockExample {
    Optional<Object> findBySlug(String slug);
}

@ExtendWith(MockitoExtension.class)
class MockedRepositoryLimitationExample {

    @Mock
    TopicRepositoryMockExample repository;

    @Test
    void thisTestPassesEvenThoughItProvesNothingAboutTheRealQuery() {
        // when(...) tells the mock exactly what to return -- it never
        // asks Spring Data JPA to actually parse "findBySlug" into a
        // query, and never runs anything against a real database.
        when(repository.findBySlug("records")).thenReturn(Optional.of(new Object()));

        Optional<Object> result = repository.findBySlug("records");

        assertThat(result).isPresent(); // passes -- but proves nothing
        // If the REAL findBySlug(...) method were misspelled as
        // "findBySlgu(...)", or if it filtered on the wrong column
        // entirely, this test would still pass -- it only verifies the
        // MOCK's own configured behavior, never the real, generated
        // query. Testing that the query itself is correct needs a real
        // persistence layer running underneath -- covered next.
    }
}
```

`when(repository.findBySlug("records")).thenReturn(...)` configures the mock's behavior directly — it never asks Spring Data JPA to parse `findBySlug` into a real query, and never touches a database. If the REAL method were misspelled, or filtered on the wrong column entirely, this test would still pass, because it only verifies the mock's own configured behavior. Verifying the query itself needs something that actually runs it.

@DataJpaTest: A Slice Test for the Persistence Layer

`@DataJpaTest` is `@WebMvcTest`'s sibling, mentioned but never explained in "Testing in Spring MVC" — the same "slice test" idea, applied to the opposite layer.

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;

@Entity
class TopicDataJpaExample {
    @Id
    @GeneratedValue
    private Long id;
    private String slug;

    TopicDataJpaExample() {
    }

    TopicDataJpaExample(String slug) {
        this.slug = slug;
    }

    String getSlug() {
        return slug;
    }
}

interface TopicDataJpaRepositoryExample extends JpaRepository<TopicDataJpaExample, Long> {
    Optional<TopicDataJpaExample> findBySlug(String slug);
}

// @DataJpaTest is @WebMvcTest's sibling slice test -- instead of loading
// only the web layer, it loads only the PERSISTENCE layer: entities,
// repositories, and the actual database connection, but none of this
// project's controllers or services. Each test method also runs inside
// its own transaction, rolled back automatically afterward -- one test's
// data never leaks into the next.
@DataJpaTest
class DataJpaTestWithTestEntityManagerExample {

    // TestEntityManager -- distinct from the EntityManager covered in
    // "The Persistence Context and Locking" -- is a test-focused wrapper
    // around it, with convenience methods for getting data into the
    // database WITHOUT going through the repository being tested. That
    // separation matters: if setup used the same repository method the
    // test is trying to verify, a bug in that method could hide itself.
```

```
    @org.springframework.beans.factory.annotation.Autowired
    private TestEntityManager entityManager;

    @org.springframework.beans.factory.annotation.Autowired
    private TopicDataJpaRepositoryExample repository;

    @Test
    void findBySlug_returnsTheRealPersistedTopic() {
        // persistAndFlush(...) saves the entity AND forces an immediate
        // flush (covered in "The Persistence Context and Locking") --
        // guaranteeing the row genuinely exists in the database before
        // the test calls the repository method it's actually testing.
        entityManager.persistAndFlush(new TopicDataJpaExample("records"));

        Optional<TopicDataJpaExample> found = repository.findBySlug("records");

        // Unlike MockedRepositoryLimitationExample, this genuinely
        // exercises Spring Data JPA's real query derivation, a real SQL
        // query, and a real database -- if "findBySlug" were misspelled
        // or filtered on the wrong column, THIS test would fail.
        assertThat(found).isPresent();
        assertThat(found.get().getSlug()).isEqualTo("records");
    }
}
```

Where `@WebMvcTest` loads only the web layer, `@DataJpaTest` loads only the PERSISTENCE layer — entities, repositories, and a real database connection, but none of this project's controllers or services. Each test method also runs inside its own transaction, automatically rolled back afterward, so one test's data never leaks into the next.

What @DataJpaTest Actually Sets Up

Beyond the transaction-per-test behavior, `@DataJpaTest` configures a handful of things together: it scans for `@Entity` classes and Spring Data JPA repositories specifically (not `@Controller`s or `@Service`s), configures Hibernate against a real database connection, and — by default — replaces whatever `DataSource` is configured with an embedded, in-memory one, a behavior "Embedded Test Database vs. Real PostgreSQL" covers next.

TestEntityManager: Setting Up Data Without a Repository

Getting data into the database for a test shouldn't use the very repository method the test is trying to verify — a bug in that method could hide itself.

`TestEntityManager` — distinct from the `EntityManager` covered in "The Persistence Context and Locking" — is a test-focused wrapper around it, with convenience methods for setting up data directly. `persistAndFlush(...)`, used in the example above, saves an entity and forces an immediate flush (also covered there), guaranteeing the row genuinely exists before the test calls the repository method it's actually testing.

Testing a Derived Query Method

A derived query method (covered in "Query Methods and JPQL with @Query") can go wrong in two independent ways — filtering the wrong rows, or ordering them incorrectly — and a good test checks both.

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

@Entity
class CodeExampleTestExample {
    @Id
    @GeneratedValue
    private Long id;
    private Long topicId;
    private Integer sortOrder;

    CodeExampleTestExample() {
    }

    CodeExampleTestExample(Long topicId, Integer sortOrder) {
        this.topicId = topicId;
        this.sortOrder = sortOrder;
    }

    Integer getSortOrder() {
        return sortOrder;
    }
}

interface CodeExampleTestRepositoryExample extends JpaRepository<CodeExampleTestExample,
Long> {
    List<CodeExampleTestExample> findByTopicIdOrderBySortOrderAsc(Long topicId);
}

// Testing this project's real CodeExampleRepository.findByTopicIdOrderBySortOrderAsc
// (covered in "Query Methods and JPQL with @Query") -- proving both that
// the FILTER is correct (only this topic's rows) and that the ORDERING is
// correct (sort_order ascending), which a mocked repository could never
// actually verify.
@DataJpaTest
class DerivedQueryMethodTestExample {

    @Autowired
    private TestEntityManager entityManager;
```

```
    @Autowired
    private CodeExampleTestRepositoryExample repository;

    @Test
    void findByTopicIdOrderBySortOrderAsc_filtersAndOrdersCorrectly() {
        // Data for TWO different topics, and deliberately out of order --
        // a passing test here proves the query does real filtering and
        // real ordering, not just "returns whatever was saved."
        entityManager.persist(new CodeExampleTestExample(1L, 2));
        entityManager.persist(new CodeExampleTestExample(1L, 1));
        entityManager.persist(new CodeExampleTestExample(2L, 1)); // a different topic
entirely
        entityManager.flush();

        List<CodeExampleTestExample> result = repository.findByTopicIdOrderBySortOrderAsc
(1L);

        assertThat(result).hasSize(2); // topic 2's row correctly excluded
        assertThat(result.get(0).getSortOrder()).isEqualTo(1); // correctly ordered
        assertThat(result.get(1).getSortOrder()).isEqualTo(2);
    }
}
```

Saving data for TWO different topics, deliberately out of order, and then asserting both that only the right topic's rows come back AND that they're correctly ordered, is what actually proves `findByTopicIdOrderBySortOrderAsc` does real filtering and real ordering — not just "returns whatever was saved," which a smaller, single-row test could accidentally pass without proving much at all.

Testing a Custom @Query

A hand-written `@Query` is exactly as easy to get syntactically wrong as a derived method name — a typo'd property path, a missing join.

```
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;

@Entity
class CategoryQueryTestExample {
    @Id
```

```
    @GeneratedValue
    private Long id;
    private String name;

    CategoryQueryTestExample() {
    }

    CategoryQueryTestExample(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }
}

@Entity
class TopicQueryTestExample {
    @Id
    @GeneratedValue
    private Long id;
    private String slug;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private CategoryQueryTestExample category;

    TopicQueryTestExample() {
    }

    TopicQueryTestExample(String slug, CategoryQueryTestExample category) {
        this.slug = slug;
        this.category = category;
    }

    CategoryQueryTestExample getCategory() {
        return category;
    }
}

interface TopicQueryTestRepositoryExample extends JpaRepository<TopicQueryTestExample,
Long> {

    @Query("select t from TopicQueryTestExample t join fetch t.category where t.slug = :
slug")
    Optional<TopicQueryTestExample> findBySlugWithCategory(String slug);
}

// A hand-written @Query -- like this project's real
// TopicRepository.findBySlugWithCategoryAndCourse, covered in "Transaction
// Management" -- is exactly as easy to get syntactically wrong (a typo'd
// property path, a missing join) as a derived method name. @DataJpaTest
// verifies the JPQL itself actually runs and returns what it should.
@DataJpaTest
class CustomQueryTestExample {

    @Autowired
    private TestEntityManager entityManager;
```

```
    @Autowired
    private TopicQueryTestRepositoryExample repository;

    @Test
    void findBySlugWithCategory_joinsTheRelationshipCorrectly() {
        CategoryQueryTestExample category = entityManager.persistAndFlush(
                new CategoryQueryTestExample("Spring MVC"));
        entityManager.persistAndFlush(new TopicQueryTestExample("records", category));

        // If the join fetch's JPQL had a typo, this test would fail here
        // -- either with no result at all, or with a real
        // LazyInitializationException the moment the category is
        // accessed outside this still-open test transaction.
        Optional<TopicQueryTestExample> found = repository.findBySlugWithCategory
("records");

        assertThat(found).isPresent();
        assertThat(found.get().getCategory().getName()).isEqualTo("Spring MVC");
    }
}
```

This mirrors this project's own real `TopicRepository.findBySlugWithCategoryAndCourse`, covered in "Transaction Management" — if the `join fetch`'s JPQL had a typo, this test would fail immediately, either with no result at all or with a genuine `LazyInitializationException` the moment the relationship is accessed outside the still-open test transaction.

Embedded Test Database vs. Real PostgreSQL

`@DataJpaTest`'s default behavior — replacing the configured `DataSource` with an embedded, in-memory database (H2, commonly) — trades one risk for another.

An embedded database is fast and needs no setup, but it isn't PostgreSQL — a query relying on PostgreSQL-specific behavior (this project's own `QuestionRepository.findRandomPublishedPool`, covered in "Query Methods and JPQL with @Query," uses a native `RANDOM()` query that wouldn't behave identically, or even necessarily run, against every embedded database) can pass against the embedded substitute and still fail against the real thing. Testing against real PostgreSQL directly avoids that gap entirely, at the cost of needing a real PostgreSQL instance available wherever tests run.

> This project's own `application-test.yml` already points its test profile at a real, locally running PostgreSQL instance rather than an embedded substitute — its own comment, written early in the project, already names the tool covered next as the more robust long-term answer.

Testcontainers: The Realistic Middle Ground

This project's own `application-test.yml` has a real comment worth quoting directly: today's tests point at a manually created `learning_test` database on a locally running PostgreSQL server — workable, but it requires that manual setup step, and every test run shares that same database rather than a genuinely clean, disposable one. The comment's own words: "Testcontainers ile her test çalıştırmasında izole, tek kullanımlık bir Postgres container'ı ayağa kaldırmak çok daha sağlam olur" (spinning up an isolated, single-use Postgres container per test run, with Testcontainers, would be far more robust).

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.orm.jpa.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

// This project's OWN application-test.yml has a real comment (written all
// the way back when the test database was first set up) that says this
// out loud: today's tests point at a manually created "learning_test"
// database on a locally running Postgres -- workable, but it requires
// that manual setup step, and every test run shares that same database
// rather than a genuinely clean, disposable one. The comment's own words:
// "Testcontainers ile her test çalıştırmasında izole, tek kullanımlık bir
// Postgres container'ı ayağa kaldırmak çok daha sağlam olur."
//
// AutoConfigureTestDatabase.Replace.NONE turns off @DataJpaTest's OWN
// default behavior of swapping in an embedded, in-memory database --
// without it, @DataJpaTest would try to replace whatever DataSource is
// configured with an embedded one instead of using Testcontainers' real
// PostgreSQL.
@Testcontainers
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class TestcontainersSketchExample {

    // A REAL, disposable PostgreSQL instance, started in a Docker
    // container just for this test class, and torn down afterward -- not
    // a manually created, long-lived database, and not an embedded
    // in-memory substitute that might not behave identically to real
    // PostgreSQL for every query.
    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:16");

    @DynamicPropertySource
    static void configureDataSource(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Test
    void placeholder() {
        // This is a SKETCH of the setup, not a full worked example -- the
        // point is what each piece is FOR, not a deep dive into
        // Testcontainers itself, which is its own, separate topic.
    }
}
```

`@Container` and a `PostgreSQLContainer` start a real, disposable PostgreSQL instance in Docker just for this test class; `@DynamicPropertySource` points Spring's `DataSource` at it; `AutoConfigureTestDatabase.Replace.NONE` stops `@DataJpaTest` from overriding that with its own embedded default. This is a sketch of the SHAPE of the setup, not a full worked

example — Testcontainers itself, with its own configuration and lifecycle concerns, is a large enough topic to deserve its own dedicated lesson rather than a deep dive appended here.

Common Misconceptions

"A passing mocked-repository test proves the query works." It proves the mock behaves as configured — nothing about the real, generated query is exercised at all. **"`@DataJpaTest` loads the whole application, just like a real request would see it."** It doesn't — it's a slice test, deliberately narrower than `@SpringBootTest`, loading only entities and repositories, not controllers or services. **"An embedded test database is just as good as testing against real PostgreSQL."** It's faster and needs no setup, but a PostgreSQL-specific query (a native query, in particular) can behave differently, or not run at all, against an embedded substitute.

Best Practices

- Reach for `@DataJpaTest` specifically to verify a derived query method, a custom `@Query`, or a `Specification` actually does what it claims — mocking a repository can't tell you that.
- Use `TestEntityManager` to set up a test's data, rather than the repository method under test itself, to avoid a bug hiding behind its own setup.
- Test both what a query includes AND what it excludes (or how it orders results) — saving data for more than one case is what actually proves filtering/ordering logic, not just "something came back."
- Reach for Testcontainers over an embedded substitute the moment a query relies on PostgreSQL-specific behavior a generic embedded database might not replicate.

Common Mistakes

- Treating a passing Mockito-mocked repository test as proof a query is correct, when it only proves the mock's own configured behavior.
- Setting up a `@DataJpaTest`'s test data through the very repository method being tested, risking a bug that hides behind its own setup.
- Testing a derived query method with only one saved row, proving nothing about whether filtering or ordering across multiple rows actually works.
- Assuming an embedded test database behaves identically to PostgreSQL for every query, especially a native one relying on PostgreSQL-specific SQL.

Summary, Cheat Sheet, and Glossary

Summary

- Mocking a repository verifies a service's own logic, but proves nothing about whether a real query actually works — `@DataJpaTest` closes that gap.
- `@DataJpaTest` is `@WebMvcTest`'s sibling slice test, loading only entities, repositories, and a real database connection, with each test running in its own rolled-back transaction.
- `TestEntityManager` sets up test data directly, deliberately independent of the repository method under test.
- A good query test checks both filtering and ordering, using data for more than one case.
- `@DataJpaTest` defaults to an embedded database; a query relying on PostgreSQL-specific behavior needs real PostgreSQL, with Testcontainers as the realistic way to get an isolated, disposable instance per test run.

Cheat Sheet

```
// A basic @DataJpaTest
@DataJpaTest
class TopicRepositoryTest {

    @Autowired TestEntityManager entityManager;
    @Autowired TopicRepository repository;

    @Test
    void findBySlug_returnsThePersistedTopic() {
        entityManager.persistAndFlush(new Topic("records"));
        assertThat(repository.findBySlug("records")).isPresent();
    }
}

// Testing filtering AND ordering together
entityManager.persist(new CodeExample(topicId, 2));
entityManager.persist(new CodeExample(topicId, 1));
entityManager.persist(new CodeExample(otherTopicId, 1));
List<CodeExample> result = repository.findByTopicIdOrderBySortOrderAsc(topicId);
assertThat(result).hasSize(2); // filtered
assertThat(result.get(0).getSortOrder()).isEqualTo(1); // ordered

// Real PostgreSQL via Testcontainers, instead of the embedded default
@Testcontainers
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class RealDatabaseTest {
    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:16");

    @DynamicPropertySource
    static void props(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
    }
}
```

Glossary

- **@DataJpaTest**: a slice test loading only the persistence layer (entities, repositories, a database connection), with each test running in its own rolled-back transaction.
- **TestEntityManager**: a test-focused wrapper around `EntityManager`, used to set up or inspect data directly, independent of the repository under test.
- **Embedded test database**: an in-memory database (commonly H2) `@DataJpaTest` substitutes in by default, fast but not identical to PostgreSQL.
- **Testcontainers**: a library that starts a real, disposable database (or other service) in Docker for the duration of a test run, avoiding both the embedded-database gap and a manually managed shared test database.