Testing Spring Data JPA Repositories

"Spring MVC'de Test Yazmak"ın yalnızca isim olarak andığı @DataJpaTest ve TestEntityManager -- bu projenin kendi Mockito-mock'lanmış servis testlerinin bırakmadan geçtiği boşluğu (bir sorgunun gerçekten çalıştığının hiç doğrulanmaması) kapatıyor; derived query/custom @Query testi, ve embedded test veritabanı vs gerçek PostgreSQL vs Testcontainers -- bu projenin kendi application-test.yml'inin gerçek yorumuna dayanarak. Spring Data JPA kategorisinin 9. ve son dersi.

İleri 35 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/testing-spring-data-jpa

---

"Spring MVC'de Test Yazmak", `@DataJpaTest`'i bir kez, geçerken, `@WebMvcTest`'in kardeşi bir slice-test annotation'ı olarak andı -- ve bir daha ona hiç dönmedi. Bu kategorinin kendi servis testleri (`QuizServiceTest`, `QuestionIngestServiceTest` ve geri kalanı), her repository'yi Mockito ile mock'lar, ki bu bir servisin kendi mantığını izole olarak test etmek için tam olarak doğrudur -- ama bu kapanış dersinin doldurduğu gerçek bir boşluk bırakır: bu projede hiçbir yerde, bir derived query metodunun, elle yazılmış bir `@Query`'nin, ya da bir `Specification`'ın adının ya da JPQL'inin iddia ettiği şeyi gerçekten yaptığını hiçbir şey doğrulamaz.

Bir Repository'yi Mock'lamak Neden Yetmez

Mock'lanmış bir repository, sana tam olarak ne söylediysen onu döndürür -- fazlası değil.

```
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(...)`, mock'un davranışını doğrudan yapılandırır -- Spring Data JPA'dan `findBySlug`'ı gerçek bir sorguya ayrıştırmasını asla istemez, ve asla bir veritabanına dokunmaz. GERÇEK metot yanlış yazılmış olsaydı, ya da tamamen yanlış bir sütunda filtreleseydi, bu test yine de geçerdi, çünkü yalnızca mock'un kendi yapılandırılmış davranışını doğrular. Sorgunun kendisini doğrulamak, onu gerçekten çalıştıran bir şeye ihtiyaç duyar.

@DataJpaTest: Persistence Katmanı İçin Bir Slice Test

`@DataJpaTest`, "Spring MVC'de Test Yazmak"ta anılan ama hiç açıklanmayan `@WebMvcTest`'in kardeşidir -- aynı "slice test" fikri, karşıt katmana uygulanmış.

```
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");
    }
}
```

`@WebMvcTest`'in yalnızca web katmanını yüklediği yerde, `@DataJpaTest` yalnızca PERSISTENCE katmanını yükler -- entity'ler, repository'ler, ve gerçek bir veritabanı bağlantısı, ama bu projenin hiçbir controller'ı ya da servisi değil. Her test metodu da kendi transaction'ı içinde çalışır, sonrasında otomatik olarak rollback edilir, bu yüzden bir testin verisi bir sonrakine asla sızmaz.

@DataJpaTest Gerçekte Neyi Kurar

Test-başına-transaction davranışının ötesinde, `@DataJpaTest` bir avuç şeyi birlikte yapılandırır: özellikle `@Entity` sınıflarını ve Spring Data JPA repository'lerini tarar (`@Controller`'ları ya da `@Service`'leri değil), Hibernate'i gerçek bir veritabanı bağlantısına karşı yapılandırır, ve -- varsayılan olarak -- yapılandırılmış olan `DataSource`'u, sırada "Embedded Test Database vs. Gerçek PostgreSQL"in işlediği bir davranışla, embedded, bellek-içi bir taneyle değiştirir.

TestEntityManager: Bir Repository Olmadan Veri Kurmak

Bir test için veritabanına veri koymak, testin doğrulamaya çalıştığı repository metodunun kendisini kullanmamalıdır -- o metottaki bir hata kendini gizleyebilir.

`TestEntityManager` -- "The Persistence Context and Locking"te işlenen `EntityManager`'dan farklı -- onun etrafında, veriyi doğrudan kurmak için kullanışlı metotları olan, test-odaklı bir sarmalayıcıdır. Yukarıdaki örnekte kullanılan `persistAndFlush(...)`, bir entity'yi kaydeder ve (orada da işlenen) anlık bir flush'ı zorlar, test gerçekte test ettiği repository metodunu çağırmadan önce satırın gerçekten var olduğunu garanti eder.

Bir Derived Query Metodunu Test Etmek

Derived bir query metodu ("Query Methods and JPQL with @Query"te işlenen), iki bağımsız şekilde yanlış gidebilir -- yanlış satırları filtreleyerek, ya da onları yanlış sıralayarak -- ve iyi bir test ikisini de kontrol eder.

```
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);
    }
}
```

İKİ farklı topic için, bilinçli olarak sırasız veri kaydetmek, sonra hem yalnızca doğru topic'in satırlarının geri geldiğini HEM de doğru sıralandıklarını doğrulamak, `findByTopicIdOrderBySortOrderAsc`'ın gerçek filtreleme ve gerçek sıralama yaptığını gerçekten kanıtlayan şeydir -- yalnızca "kaydedilen her neyse onu döndürür"ü değil, ki bu daha küçük, tek satırlı bir test yanlışlıkla, gerçekte hiçbir şey kanıtlamadan geçebilirdi.

Custom Bir @Query'yi Test Etmek

Elle yazılmış bir `@Query`, derived bir metot adı kadar sözdizimsel olarak yanlış yazılması kolaydır -- yanlış yazılmış bir property yolu, eksik bir 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");
    }
}
```

Bu, "Transaction Management"te işlenen bu projenin gerçek `TopicRepository.findBySlugWithCategoryAndCourse`'unu yansıtır -- `join fetch`'in JPQL'inde bir yazım hatası olsaydı, bu test hemen başarısız olurdu, ya hiçbir sonuç olmadan ya da ilişkiye hâlâ açık olan test transaction'ının dışında erişildiği anda gerçek bir `LazyInitializationException`'la.

Embedded Test Database vs. Gerçek PostgreSQL

`@DataJpaTest`'in varsayılan davranışı -- yapılandırılmış `DataSource`'u embedded, bellek-içi bir veritabanıyla (genelde H2) değiştirmek -- bir riski başka bir riskle takas eder.

Embedded bir veritabanı hızlıdır ve kurulum gerektirmez, ama PostgreSQL değildir -- PostgreSQL'e özgü bir davranışa dayanan bir sorgu (bu projenin gerçek `QuestionRepository.findRandomPublishedPool`'u, "Query Methods and JPQL with @Query"te işlenen, her embedded veritabanına karşı aynı şekilde davranmayacak, hatta gerekli şekilde çalışmayacak bir native `RANDOM()` sorgusu kullanır) embedded ikame karşısında geçebilir ve yine de gerçek şeye karşı başarısız olabilir. Doğrudan gerçek PostgreSQL'e karşı test etmek, bu boşluğu tamamen atlatır, her testin çalıştığı her yerde gerçek bir PostgreSQL instance'ının mevcut olması gerekliliği pahasına.

> Bu projenin kendi `application-test.yml`'i, test profilini zaten embedded bir ikame yerine gerçek, yerel olarak çalışan bir PostgreSQL instance'ına yönlendiriyor -- projenin erken bir döneminde yazılan kendi yorumu, sırada işlenen aracı, daha sağlam bir uzun-vadeli cevap olarak zaten adlandırıyor.

Testcontainers: Gerçekçi Orta Yol

Bu projenin kendi `application-test.yml`'inde doğrudan alıntılanmaya değer gerçek bir yorum var: bugünkü testler, yerel olarak çalışan bir PostgreSQL sunucusundaki elle oluşturulmuş bir `learning_test` veritabanına yönleniyor -- işe yarıyor, ama bu elle kurulum adımını gerektiriyor, ve her test çalışması, gerçekten temiz, tek kullanımlık bir tane yerine aynı veritabanını paylaşıyor. Yorumun kendi sözleri: "Testcontainers ile her test çalıştırmasında izole, tek kullanımlık bir Postgres container'ı ayağa kaldırmak çok daha sağlam olur."

```
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` ve bir `PostgreSQLContainer`, tam olarak bu test sınıfı için Docker'da gerçek, tek kullanımlık bir PostgreSQL instance'ı başlatır; `@DynamicPropertySource`, Spring'in `DataSource` 'unu ona yönlendirir; `AutoConfigureTestDatabase.Replace.NONE`, `@DataJpaTest` 'in bunu kendi embedded varsayılanıyla geçersiz kılmasını durdurur. Bu, kurulumun ŞEKLİNİN bir taslağıdır,

tam işlenmiş bir örnek değil -- Testcontainers'ın kendisi, kendi yapılandırma ve yaşam döngüsü meseleleriyle, burada eklenmiş bir derin dalışı değil, kendi ayrı, özel dersini hak edecek kadar büyük bir konu.

Yaygın Yanlış Anlamalar

"Geçen, mock'lanmış bir repository testi sorgunun çalıştığını kanıtlar." Mock'un yapılandırıldığı gibi davrandığını kanıtlar -- gerçek, üretilmiş sorgu hakkında hiçbir şey hiç çalıştırılmaz. **"`@DataJpaTest`, tam olarak gerçek bir isteğin göreceği gibi bütün uygulamayı yükler."** Yüklemez -- bilinçli olarak `@SpringBootTest`'ten daha dar bir slice testtir, yalnızca entity'leri ve repository'leri yükler, controller'ları ya da servisleri değil. **"Embedded bir test veritabanı, gerçek PostgreSQL'e karşı test etmek kadar iyidir."** Daha hızlıdır ve kurulum gerektirmez, ama PostgreSQL'e özgü bir sorgu (özellikle bir native sorgu), embedded bir ikame karşısında farklı davranabilir, ya da hiç çalışmayabilir.

Best Practices

- Bir derived query metodunun, custom bir `@Query`'nin, ya da bir `Specification`'ın gerçekten iddia ettiğini yaptığını doğrulamak için özellikle `@DataJpaTest`'e başvur -- bir repository'yi mock'lamak sana bunu söyleyemez.
- Bir testin verisini kurmak için, test edilen repository metodunun kendisi yerine `TestEntityManager` kullan, kendi kurulumunun arkasına gizlenen bir hatadan kaçınmak için.
- Bir sorgunun hem NEYİ içerdiğini HEM DE NEYİ hariç tuttuğunu (ya da sonuçları nasıl sıraladığını) test et -- birden fazla durum için veri kaydetmek, yalnızca "bir şey geri geldi" yi değil, filtreleme/sıralama mantığını gerçekten kanıtlayan şeydir.
- Bir sorgu, generic bir embedded veritabanının tekrarlayamayabileceği PostgreSQL'e özgü bir davranışa dayandığı anda, embedded bir ikame yerine Testcontainers'a başvur.

Yaygın Hatalar

- Geçen, Mockito-mock'lanmış bir repository testini, yalnızca mock'un kendi yapılandırılmış davranışını kanıtlarken, bir sorgunun doğru olduğunun kanıtı olarak ele almak.
- Bir `@DataJpaTest`'in test verisini, test edilen repository metodunun tam kendisi üzerinden kurmak, kendi kurulumunun arkasına gizlenen bir hata riski taşımak.
- Derived bir query metodunu yalnızca tek bir kaydedilmiş satırla test etmek, birden fazla satır arasında filtrelemenin ya da sıralamanın gerçekten çalışıp çalışmadığı hakkında hiçbir şey kanıtlamamak.
- Embedded bir test veritabanının, özellikle PostgreSQL'e özgü SQL'e dayanan bir native sorgu için, her sorguda PostgreSQL'le birebir aynı davrandığını varsaymak.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bir repository'yi mock'lamak bir servisin kendi mantığını doğrular, ama gerçek bir sorgunun gerçekten çalıştığı hakkında hiçbir şey kanıtlamaz -- `@DataJpaTest` bu boşluğu kapatır.
- `@DataJpaTest`, `@WebMvcTest`'in kardeşi slice testtir, yalnızca entity'leri, repository'leri, ve gerçek bir veritabanı bağlantısını yükler, her test kendi rollback edilen transaction'ında çalışır.
- `TestEntityManager`, test verisini doğrudan, bilinçli olarak test edilen repository metodundan bağımsız olarak kurar.
- İyi bir sorgu testi, birden fazla durum için veri kullanarak hem filtrelemeyi hem sıralamayı kontrol eder.
- `@DataJpaTest` varsayılan olarak embedded bir veritabanına düşer; PostgreSQL'e özgü bir davranışa dayanan bir sorgu gerçek PostgreSQL'e ihtiyaç duyar, test çalışması başına izole, tek kullanımlık bir instance elde etmenin gerçekçi yolu olarak Testcontainers'la.

Cheat Sheet

```
// Temel bir @DataJpaTest
@DataJpaTest
class TopicRepositoryTest {

    @Autowired TestEntityManager entityManager;
    @Autowired TopicRepository repository;

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

// Filtrelemeyi VE sıralamayı birlikte test etmek
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); // filtrelendi
assertThat(result.get(0).getSortOrder()).isEqualTo(1); // sıralandı

// Embedded varsayılan yerine Testcontainers ile gerçek PostgreSQL
@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);
    }
}
```

Terimler Sözlüğü

- **@DataJpaTest**: yalnızca persistence katmanını (entity'ler, repository'ler, bir veritabanı bağlantısı) yükleyen, her testin kendi rollback edilen transaction'ında çalıştığı bir slice test.
- **TestEntityManager**: `EntityManager`'ın etrafında, veriyi doğrudan, test edilen repository'den bağımsız olarak kurmak ya da incelemek için kullanılan, test-odaklı bir sarmalayıcı.
- **Embedded test veritabanı**: `@DataJpaTest`'in varsayılan olarak yerine koyduğu, hızlı ama PostgreSQL'le birebir aynı olmayan bellek-içi bir veritabanı (genelde H2).
- **Testcontainers**: bir test çalışması süresince Docker'da gerçek, tek kullanımlık bir veritabanı (ya da başka bir servis) başlatan, hem embedded-veritabanı boşluğundan hem de elle yönetilen paylaşılan bir test veritabanından kaçınan bir kütüphane.