Tests d'intégration JPA - Les fondamentaux

1. Introduction et concepts

Test d'intégration JPA : Teste les entités, repositories et requêtes avec une vraie base de données (ou H2 en mémoire).

Différence avec test unitaire :

- Teste les mappings JPA réels
- Valide les contraintes DB
- Vérifie les requêtes SQL générées
- Détecte les problèmes N+1
- Plus lent qu'un test unitaire

2. Configuration de base

2.1 Dépendances Maven

```
<dependencies>
    <!-- Spring Boot Test (inclut JUnit 5, Mockito, AssertJ) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <!-- H2 pour tests en mémoire -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>test</scope>
    </dependency>

    <!-- Testcontainers (optionnel, pour base réelle) -->
    <dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>postgresql</artifactId>
        <version>1.19.3</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2.2 Configuration de test (application-test.properties)

```
# src/test/resources/application-test.properties

# H2 Database en mémoire
spring.datasource.url=jdbc:h2:mem:testdb
```

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true

# Désactiver cache pour tests prédictibles
spring.jpa.properties.hibernate.cache.use_second_level_cache=false

# Logging SQL détaillé
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
logging.level.org.hibernate.orm.jdbc.bind=TRACE
```

3. Anatomie d'un test JPA

3.1 Test Repository basique

```
@DataJpaTest  // ← Annotation clé
@ActiveProfiles("test")
class ProductRepositoryTest {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private TestEntityManager entityManager;  // ← Utilitaire de test JPA

    @Test
    void shouldSaveAndFindProduct() {
        // Given
        Product product = new Product();
        product.setName("Test Product");
        product.setPrice(new BigDecimal("99.99"));

        // When
        Product saved = productRepository.save(product);
        entityManager.flush();  // Force SQL immédiat
        entityManager.clear();  // Vide le cache (simule nouvelle session)

        // Then
        Product found = productRepository.findById(saved.getId()).orElseThrow();
        assertThat(found.getName()).isEqualTo("Test Product");
        assertThat(found.getPrice()).isEqualByComparingTo("99.99");
    }
}
```

Annotations essentielles :

- `@DataJpaTest` : Configure uniquement la couche JPA (pas de serveur web)
- `@AutoConfigureTestDatabase(replace = NONE)` : Utilise la DB configurée (pas H2 auto)
- `@Sql` : Exécute un script SQL avant le test

3.2 TestEntityManager - Les commandes clés

```
@Test
void demonstrateTestEntityManager() {
    Product product = new Product("Laptop", new BigDecimal("1200"));

    // persist() : INSERT sans flush immédiat
    entityManager.persist(product);

    // flush() : Force l'exécution des SQL en attente
    entityManager.flush();

    // clear() : Vide le contexte de persistance (cache 1er niveau)
    entityManager.clear();

    // find() : SELECT en DB (car cache vidé)
    Product fromDb = entityManager.find(Product.class, product.getId());

    // detach() : Détache une entité du contexte
    entityManager.detach(fromDb);
}
```

4. Tests des associations

4.1 OneToMany bidirectionnel

```
@Test
void shouldCascadeOrderToOrderItems() {
    // Given
    User user = new User("john@test.com");
    entityManager.persist(user);

    Order order = new Order(user);
    OrderItem item1 = new OrderItem(order, "Product A", 2);
    OrderItem item2 = new OrderItem(order, "Product B", 1);

    order.addItem(item1);  // Méthode helper bidirectionnelle
    order.addItem(item2);

    // When
    entityManager.persist(order);  // CASCADE.PERSIST sur items
    entityManager.flush();
    entityManager.clear();

    // Then
    Order found = entityManager.find(Order.class, order.getId());
```

```
    assertThat(found.getItems()).hasSize(2);
    assertThat(found.getItems())
        .extracting(OrderItem::getProductName)
        .containsExactlyInAnyOrder("Product A", "Product B");
}
```

4.2 Problème N+1 - Détection

```
@Test
void shouldDetectNPlusOneProblem() {
    // Given : 3 orders avec items
    createOrdersWithItems(3);
    entityManager.clear();

    // When : Récupération sans FETCH
    List<Order> orders = entityManager
        .createQuery("SELECT o FROM Order o", Order.class)
        .getResultList();

    // Then : Provoque N+1 si on accède aux items
    orders.forEach(order -> {
        // □□ 1 requête par order.getItems() = N+1
        System.out.println("Items count: " + order.getItems().size());
    });

    // Vérifie le nombre de requêtes (avec Hypersistence Utils)
    // assertSelectCount(1 + 3); // 1 pour orders + 3 pour items
}

@Test
void shouldSolveNPlusOneWithJoinFetch() {
    // Given
    createOrdersWithItems(3);
    entityManager.clear();

    // When : Avec JOIN FETCH
    List<Order> orders = entityManager
        .createQuery("SELECT DISTINCT o FROM Order o LEFT JOIN FETCH o.items",
Order.class)
        .getResultList();

    // Then : 1 seule requête
    orders.forEach(order -> {
        System.out.println("Items count: " + order.getItems().size());
    });

    // assertSelectCount(1); // Une seule requête
}
```

5. Tests des requêtes JPQL

5.1 Query basique

```
@Test
void shouldFindProductsByPriceRange() {
    // Given
    entityManager.persist(new Product("Cheap", new BigDecimal("10")));
    entityManager.persist(new Product("Medium", new BigDecimal("50")));
    entityManager.persist(new Product("Expensive", new BigDecimal("200")));
    entityManager.flush();

    // When
    List<Product> products = productRepository.findByPriceBetween(
        new BigDecimal("20"),
        new BigDecimal("100")
    );

    // Then
    assertThat(products)
        .hasSize(1)
        .extracting(Product::getName)
        .containsExactly("Medium");
}
```

5.2 Projection DTO

```
public record ProductSummary(UUID id, String name, BigDecimal price) {}

@Test
void shouldProjectToDTO() {
    // Given
    entityManager.persist(new Product("Test", new BigDecimal("99.99")));
    entityManager.flush();

    // When
    List<ProductSummary> summaries = entityManager
        .createQuery(
            "SELECT new com.example.dto.ProductSummary(p.id, p.name, p.price) " +
            "FROM Product p",
            ProductSummary.class
        )
        .getResultList();

    // Then
    assertThat(summaries).hasSize(1);
    assertThat(summaries.get(0).name()).isEqualTo("Test");
}
```

6. Tests avec données initiales

6.1 Via fichier SQL

```
@Test
@Sql("/test-data/products.sql")  // ← Exécute avant le test
void shouldLoadFromSqlFile() {
```

```
    List<Product> products = productRepository.findAll();
    assertThat(products).hasSizeGreaterThan(0);
}
```

Fichier `src/test/resources/test-data/products.sql` :

```
INSERT INTO product (id, name, price, stock) VALUES
('123e4567-e89b-12d3-a456-426614174000', 'Product 1', 10.00, 100),
('123e4567-e89b-12d3-a456-426614174001', 'Product 2', 20.00, 50);
```

6.2 Via méthode @BeforeEach

```
@DataJpaTest
class OrderRepositoryTest {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private TestEntityManager entityManager;

    private User testUser;

    @BeforeEach
    void setUp() {
        testUser = new User("test@example.com");
        entityManager.persist(testUser);
        entityManager.flush();
    }

    @Test
    void shouldFindOrdersByUser() {
        Order order = new Order(testUser);
        entityManager.persist(order);

        List<Order> orders = orderRepository.findByUser(testUser);
        assertThat(orders).hasSize(1);
    }
}
```

7. Tests des contraintes

7.1 Validation Bean Validation

```
@Test
void shouldFailWhenEmailInvalid() {
    // Given
    User user = new User();
    user.setUsername("john");
    user.setEmail("invalid-email");  // ← Email invalide

    // When/Then
    assertThatThrownBy(() -> {
```

```
        entityManager.persist(user);
        entityManager.flush();  // Validation lors du flush
    })
    .isInstanceOf(ConstraintViolationException.class)
    .hasMessageContaining("email");
}
```

7.2 Contrainte unique

```
@Test
void shouldFailOnDuplicateEmail() {
    // Given
    entityManager.persist(new User("john@test.com"));
    entityManager.flush();
    entityManager.clear();

    // When/Then
    assertThatThrownBy(() -> {
        User duplicate = new User("john@test.com");
        entityManager.persist(duplicate);
        entityManager.flush();
    })
    .isInstanceOf(DataIntegrityViolationException.class);
}
```

8. Testcontainers (DB réelle)

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@Testcontainers
class ProductRepositoryTestcontainersTest {

    @Container
    static PostgreSQLContainer<?> postgres = new
PostgreSQLContainer<>("postgres:15")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

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

    @Autowired
    private ProductRepository productRepository;

    @Test
    void shouldWorkWithRealPostgres() {
        Product product = new Product("Real DB Test", new BigDecimal("99"));
        productRepository.save(product);
```

```
        assertThat(productRepository.findById(product.getId())).isPresent();
    }
}
```

9. Bonnes pratiques

DO

- Utiliser `flush()` et `clear()` pour isoler les tests du cache
- Tester les cas limites (null, contraintes, cascades)
- Vérifier les requêtes SQL générées (`show-sql: true`)
- Utiliser AssertJ pour des assertions lisibles
- Nommer explicitement les tests (`shouldXxxWhenYyy`)

DON'T

- Ne pas tester la logique métier ici (c'est le rôle des tests unitaires)
- Éviter les dépendances entre tests
- Ne pas réutiliser les mêmes données sans `clear()`
- Ne pas oublier `@Transactional` est par défaut avec `@DataJpaTest`

10. Exercice pratique

À implémenter :

Créer les tests d'intégration pour l'entité `Review` :

1. [ ] Test de création d'une review avec associations
2. [ ] Test de la contrainte `rating` entre 1 et 5
3. [ ] Test unicité (user, product)
4. [ ] Test du chargement avec `JOIN FETCH` (éviter N+1)
5. [ ] Test de calcul de moyenne des ratings par produit
6. [ ] Test de la projection `ReviewSummaryDTO`

Ressources

- Spring Boot Testing Documentation
- Baeldung - Spring Boot Testing
- Vlad Mihalcea's Blog