Séance 3 - Tests et CI/CD (4h)

Objectifs pédagogiques

- Comprendre la différence entre tests unitaires et tests d'intégration
- Écrire des tests simples et efficaces avec les bonnes pratiques
- Gérer les profils Spring (dev/test/prod)
- Mettre en place une pipeline CI complète avec GitHub Actions
- Mesurer la couverture de code

Partie 0 : Point de départ (15min)

Point avancement TD2 * Qui a terminé les associations Order/OrderItem/User ? * Qui a résolu des problèmes N+1 ? * Décision : **Ceux qui ont fini peuvent commencer les tests, les autres finalisent le TD2 </WRAP> ===== Partie 1 : Configuration multi-environnements (30min) ===== ==== 1.1 Stratégie de profils Spring ====**

Objectif : Séparer les configurations selon l'environnement (dev, test, prod)

=== Structure des fichiers === <sxh;gutter:false> src/main/resources/ ├── application.properties # Configuration commune ├── application-dev.properties # Développement local ├── application-test.properties # Tests automatisés └── application-prod.properties # Production </sxh> === application.properties (commun) === <sxh properties;gutter:false> # Configuration commune à tous les profils spring.application.name=ecommerce-api server.port=8080 # JPA commun spring.jpa.open-in-view=false spring.jpa.properties.hibernate.jdbc.time_zone=UTC # Validation spring.jackson.deserialization.fail-on-unknown-properties=true </sxh> === application-dev.properties === <sxh properties;gutter:false> # Base H2 fichier pour le dev spring.datasource.url=jdbc:h2:file:./data/ecommerce-dev spring.datasource.username=sa spring.datasource.password= # Console H2 activée spring.h2.console.enabled=true spring.h2.console.path=/h2-console # DDL auto pour en dev spring.jpa.hibernate.ddl-auto=update # Logs verbeux spring.jpa.show-sql=true spring.jpa.properties.hibernate.format_sql=true logging.level.com.ecommerce=DEBUG logging.level.io.hypersistence.utils=DEBUG </sxh> === application-test.properties === <sxh properties;gutter:false> # Base H2 en mémoire pour les tests spring.datasource.url=jdbc:h2:mem:testdb;MODE=PostgreSQL;DB_CLOSE_DELAY=-1 spring.datasource.username=sa spring.datasource.password= # Recréation du schéma à chaque test spring.jpa.hibernate.ddl-auto=create-drop # Logs minimaux (sauf erreurs) spring.jpa.show-sql=false logging.level.com.ecommerce=INFO logging.level.org.hibernate=WARN # Performance tests spring.jpa.properties.hibernate.generate_statistics=true # Désactivation fonctionnalités non nécessaires en test spring.h2.console.enabled=false </sxh> === application-prod.properties === <sxh properties;gutter:false> # Base PostgreSQL (exemple) spring.datasource.url=${DATABASE_URL} spring.datasource.username=${DB_USERNAME} spring.datasource.password=${DB_PASSWORD} # JAMAIS de DDL auto en production spring.jpa.hibernate.ddl-auto=validate # Logs minimaux spring.jpa.show-sql=false logging.level.com.ecommerce=INFO # Sécurité spring.h2.console.enabled=false </sxh> ==== 1.2 Activation des profils ==== <sxh bash;gutter:false> # Dans IntelliJ : Run Configuration > Active profiles: dev # Ou via variable d'environnement export SPRING_PROFILES_ACTIVE=dev #

Via ligne de commande mvn spring-boot:run -Dspring-boot.run.profiles=dev </sxh>



Exercice 1 (15min) :

1. Créer les 4 fichiers de configuration
2. Tester le lancement avec le profil dev
3. Vérifier que la console H2 est accessible sur `/h2-console`
4. Relancer avec le profil `test` et constater les différences de logs

===== Partie 2 : Tests Unitaires (1h15) =====



Principe clé : Un test unitaire teste **UNE** classe isolée, sans base de données, très rapidement

==== 2.1 Dépendances nécessaires (pom.xml) ==== <sxh xml> <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> </dependencies> </sxh> ==== 2.2 Premier test simple : ProductService ==== <sxh java> package com.ecommerce.service; import com.ecommerce.domain.Product; import com.ecommerce.domain.Category; import com.ecommerce.repository.ProductRepository; import com.ecommerce.exception.ProductNotFoundException; import com.ecommerce.exception.InsufficientStockException; import org.junit.jupiter.api.Test; import org.junit.jupiter.api.DisplayName; import org.junit.jupiter.api.extension.ExtendWith; import org.mockito.InjectMocks; import org.mockito.Mock; import org.mockito.junit.jupiter.MockitoExtension; import java.math.BigDecimal; import java.util.Optional; import java.util.UUID; import static org.assertj.core.api.Assertions.*; import static org.mockito.Mockito.*; @ExtendWith(MockitoExtension.class) @DisplayName(“ProductService - Unit Tests”) class ProductServiceTest { @Mock private ProductRepository productRepository; @InjectMocks private ProductService productService; @Test @DisplayName(“Should return product when it exists”) void getProduct_WhenExists_ShouldReturnProduct() { *Given (Arrange) UUID productId = UUID.randomUUID(); Category category = new Category(“Electronics”, “Devices”); Product expectedProduct = Product.builder() .id(productId) .name(“iPhone”) .price(new BigDecimal(“999.99”)) .stock(10) .category(category) .build(); when(productRepository.findById(productId)) .thenReturn(Optional.of(expectedProduct));* When (Act) Product result = productService.getProduct(productId); *Then (Assert) assertThat(result).isNotNull(); assertThat(result.getName()).isEqualTo(“iPhone”); assertThat(result.getPrice()).isEqualByComparingTo(“999.99”); assertThat(result.getStock()).isEqualTo(10); verify(productRepository, times(1)).findById(productId); verifyNoMoreInteractions(productRepository); } @Test @DisplayName(“Should throw exception when product not found”) void getProduct_WhenNotExists_ShouldThrowException() {* Given UUID productId = UUID.randomUUID(); when(productRepository.findById(productId)) .thenReturn(Optional.empty()); *When & Then assertThatThrownBy[1] .isInstanceOf(ProductNotFoundException.class) .hasMessageContaining(productId.toString()); verify(productRepository).findById(productId); } @Test @DisplayName(“Should decrease stock when updating with negative quantity”) void updateStock_WithNegativeQuantity_ShouldDecreaseStock() {* Given UUID productId = UUID.randomUUID(); Product product = Product.builder() .id(productId) .name(“Test Product”) .price(BigDecimal.TEN) .stock(10) .build(); when(productRepository.findById(productId)) .thenReturn(Optional.of(product)); when(productRepository.save(any(Product.class))) .thenReturn(product); *When productService.updateStock(productId, -3);* Then

assertThat(product.getStock()).isEqualTo(7); verify(productRepository).save(product); } @Test @DisplayName(“Should throw exception when insufficient stock”) void updateStock_WithInsufficientStock_ShouldThrowException() { *Given UUID productId = UUID.randomUUID(); Product product = Product.builder() .id(productId) .name(“Test Product”) .price(BigDecimal.TEN) .stock(5) .build(); when(productRepository.findById(productId)) .thenReturn(Optional.of(product));* When & Then assertThatThrownBy[2] .isInstanceOf(InsufficientStockException.class); verify(productRepository, never()).save(any()); } @Test @DisplayName(“Should increase stock when updating with positive quantity”) void updateStock_WithPositiveQuantity_ShouldIncreaseStock() { *Given UUID productId = UUID.randomUUID(); Product product = Product.builder() .id(productId) .name(“Test Product”) .price(BigDecimal.TEN) .stock(10) .build(); when(productRepository.findById(productId)) .thenReturn(Optional.of(product)); when(productRepository.save(any(Product.class))) .thenReturn(product);* When productService.updateStock(productId, 5); *Then assertThat(product.getStock()).isEqualTo(15); verify(productRepository).save(product); } } </sxh> ==== 2.3 Concepts clés ==== <sxh java;gutter:false>* @Mock : Crée un faux objet (ne fait rien par défaut) @Mock private ProductRepository productRepository; *@InjectMocks : Injecte automatiquement les mocks dans la classe testée @InjectMocks private ProductService productService;* when(...).thenReturn(...) : Définit le comportement du mock when(productRepository.findById(id)).thenReturn(Optional.of(product)); *verify(...) : Vérifie qu'une méthode a été appelée (et combien de fois) verify(productRepository, times(1)).save(any()); verify(productRepository, never()).delete(any());* assertThat(...) : Vérifie le résultat (AssertJ - plus lisible que assertEquals) assertThat(result.getStock()).isEqualTo(7); assertThat(result).isNotNull(); assertThat(list).hasSize(3); *assertThatThrownBy : Vérifie qu'une exception est levée assertThatThrownBy*[3] *.isInstanceOf(MyException.class) .hasMessage(“Expected message”); </sxh> ==== 2.4 Tests paramétrés (en plus) ==== <sxh java> import org.junit.jupiter.params.ParameterizedTest; import org.junit.jupiter.params.provider.CsvSource; import org.junit.jupiter.params.provider.ValueSource; @ParameterizedTest @DisplayName(“Should validate price is positive”) @ValueSource(strings = {“-10.00”, “-0.01”, “0.00”}) void createProduct_WithInvalidPrice_ShouldThrowException(String price) {* Given CreateProductDto dto = new CreateProductDto(); dto.setName(“Test”); dto.setPrice(new BigDecimal(price)); dto.setStock(10); *When & Then assertThatThrownBy*[4] *.isInstanceOf(InvalidPriceException.class); } @ParameterizedTest @DisplayName(“Should calculate correct total for different quantities”) @CsvSource({ “1, 10.00, 10.00”, “2, 10.00, 20.00”, “5, 9.99, 49.95” }) void calculateTotal_WithDifferentQuantities_ShouldReturnCorrectAmount(int quantity, String unitPrice, String expectedTotal) {* Given Product product = Product.builder() .price(new BigDecimal(unitPrice)) .build(); *When BigDecimal total = productService.calculateTotal(product, quantity);* Then assertThat(total).isEqualByComparingTo(expectedTotal); } </sxh>

Exercice 2 (45min) :

Créer `UserServiceTest` avec au moins 5 tests :

1. `createUser_WithValidData_ShouldReturnUser`
2. `createUser_WithDuplicateEmail_ShouldThrowException`
3. `getUser_WhenExists_ShouldReturnUser`
4. `getUser_WhenNotExists_ShouldThrowException`
5. `getUserOrders_ShouldReturnOrderHistory`

Template fourni :

```
@ExtendWith(MockitoExtension.class)
@DisplayName("UserService - Unit Tests")
class UserServiceTest {
```

```
    @Mock
    private UserRepository userRepository;

    @Mock
    private OrderRepository orderRepository;

    @InjectMocks
    private UserService userService;

    @Test
    @DisplayName("Should create user with valid data")
    void createUser_WithValidData_ShouldReturnUser() {
        // Given
        CreateUserDto dto = new CreateUserDto("John Doe",
"john@example.com");
        User user = User.builder()
            .id(UUID.randomUUID())
            .name(dto.getName())
            .email(dto.getEmail())
            .build();

when(userRepository.existsByEmail(dto.getEmail())).thenReturn(
false);
when(userRepository.save(any(User.class))).thenReturn(user);

        // When
        User result = userService.createUser(dto);

        // Then
        assertThat(result).isNotNull();
assertThat(result.getEmail()).isEqualTo("john@example.com");
        verify(userRepository).save(any(User.class));
    }

    // TODO: Implémenter les 4 autres tests
}
```

Critères de validation :

- ☐ Tous les tests passent (`mvn test`)
- ☐ Utilisation correcte des mocks
- ☐ Pattern AAA (Arrange, Act, Assert) respecté
- ☐ Messages d'erreur explicites avec `@DisplayName`

===== Partie 3 : Tests d'Intégration (1h15) =====

Test d'intégration : Teste le fonctionnement complet de l'API (Controller → Service → Repository → DB)

==== 3.1 Configuration de base ==== <sxh java> package com.ecommerce.controller; import com.ecommerce.domain.Product; import com.ecommerce.domain.Category; import com.ecommerce.repository.ProductRepository; import

com.ecommerce.repository.CategoryRepository; import org.junit.jupiter.api.BeforeEach; import org.junit.jupiter.api.Test; import org.junit.jupiter.api.DisplayName; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.http.MediaType; import org.springframework.test.context.ActiveProfiles; import org.springframework.test.web.servlet.MockMvc; import org.springframework.transaction.annotation.Transactional; import java.math.BigDecimal; import static org.hamcrest.Matchers.*; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*; import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*; @SpringBootTest @AutoConfigureMockMvc @ActiveProfiles(“test”) @Transactional *Rollback automatique après chaque test @DisplayName(“ProductController - Integration Tests”) class ProductControllerIntegrationTest { @Autowired private MockMvc mockMvc; @Autowired private ProductRepository productRepository; @Autowired private CategoryRepository categoryRepository; private Category electronics; @BeforeEach void setUp() {* Nettoyage (si @Transactional ne suffit pas) productRepository.deleteAll(); categoryRepository.deleteAll(); *Données de test electronics = categoryRepository.save(new Category(“Electronics”, “Electronic devices”)); } @Test @DisplayName(“GET /products/{id} should return 200 when product exists”) void getProduct_WhenExists_ShouldReturn200() throws Exception {* Given Product product = productRepository.save(Product.builder() .name(“iPhone 15”) .price(new BigDecimal(“999.99”)) .stock(50) .category(electronics) .build()); *When & Then mockMvc.perform(get(“/products/{id}”, product.getId())) .andDo(print())* Affiche la requête/réponse (utile pour déboguer) .andExpect(status().isOk()) .andExpect(content().contentType(MediaType.APPLICATION_JSON)) .andExpect(jsonPath(“$.id”).value(product.getId().toString())) .andExpect(jsonPath(“$.name”).value(“iPhone 15”)) .andExpect(jsonPath(“$.price”).value(999.99)) .andExpect(jsonPath(“$.stock”).value(50)) .andExpect(jsonPath(“$.category.name”).value(“Electronics”)); } @Test @DisplayName(“GET /products/{id} should return 404 when product not found”) void getProduct_WhenNotExists_ShouldReturn404() throws Exception { *When & Then mockMvc.perform(get(“/products/{id}”, “00000000-0000-0000-0000-000000000000”)) .andExpect(status().isNotFound()) .andExpect(jsonPath(“$.message”).exists()); } @Test @DisplayName(“POST /products should return 201 with valid data”) void createProduct_WithValidData_ShouldReturn201() throws Exception {* Given String requestBody = “”“ { “name”: “iPad Pro”, “price”: 799.99, “stock”: 30, “categoryId”: ”%s“ } ”“”.formatted(electronics.getId()); *When & Then mockMvc.perform(post(“/products”) .contentType(MediaType.APPLICATION_JSON) .content(requestBody)) .andDo(print()) .andExpect(status().isCreated()) .andExpect(header().exists(“Location”)) .andExpect(jsonPath(“$.id”).exists()) .andExpect(jsonPath(“$.name”).value(“iPad Pro”)) .andExpect(jsonPath(“$.price”).value(799.99)) .andExpect(jsonPath(“$.stock”).value(30)); } @Test @DisplayName(“POST /products should return 400 with invalid data”) void createProduct_WithInvalidData_ShouldReturn400() throws Exception {* Given - prix négatif String requestBody = “”“ { “name”: “Invalid Product”, “price”: -10.00, “stock”: 10, “categoryId”: ”%s“ } ”“”.formatted(electronics.getId()); *When & Then mockMvc.perform(post(“/products”) .contentType(MediaType.APPLICATION_JSON) .content(requestBody)) .andExpect(status().isBadRequest()) .andExpect(jsonPath(“$.errors”).isArray()); } @Test @DisplayName(“PUT /products/{id}/stock should update stock correctly”) void updateStock_WithValidQuantity_ShouldReturn200() throws Exception {* Given Product product = productRepository.save(Product.builder() .name(“Test Product”) .price(BigDecimal.TEN) .stock(10) .category(electronics) .build()); String requestBody = “”“ { “quantity”: 5 } ”“”; *When & Then mockMvc.perform(put(“/products/{id}/stock”, product.getId()) .contentType(MediaType.APPLICATION_JSON) .content(requestBody)) .andExpect(status().isOk()) .andExpect(jsonPath(“$.stock”).value(15)); } @Test @DisplayName(“GET /products should return paginated list”) void getProducts_ShouldReturnPaginatedList() throws Exception {* Given productRepository.save(Product.builder() .name(“Product 1”) .price(BigDecimal.TEN) .stock(10)

.category(electronics) .build()); productRepository.save(Product.builder() .name(“Product 2”) .price(BigDecimal.valueOf(20)) .stock(20) .category(electronics) .build()); *When & Then mockMvc.perform(get(“/products”) .param(“page”, “0”) .param(“size”, “10”)) .andExpect(status().isOk()) .andExpect(jsonPath(“$.content”).isArray()) .andExpect(jsonPath(“$.content”, hasSize(2))) .andExpect(jsonPath(“$.totalElements”).value(2)); } @Test @DisplayName(“GET /products should filter by category”) void getProducts_WithCategoryFilter_ShouldReturnFilteredList() throws Exception {* Given Category books = categoryRepository.save(new Category(“Books”, “Books category”)); productRepository.save(Product.builder() .name(“iPhone”) .price(BigDecimal.valueOf(999)) .stock(10) .category(electronics) .build()); productRepository.save(Product.builder() .name(“Java Book”) .price(BigDecimal.valueOf(50)) .stock(20) .category(books) .build()); *When & Then mockMvc.perform(get(“/products”) .param(“categoryId”, electronics.getId().toString())) .andExpect(status().isOk()) .andExpect(jsonPath(“$.content”, hasSize(1))) .andExpect(jsonPath(“$.content[0].name”).value(“iPhone”)); } } </sxh> ==== 3.2 Concepts clés ==== <sxh java;gutter:false>* @SpringBootTest : Lance toute l'application Spring @SpringBootTest *@AutoConfigureMockMvc : Configure MockMvc pour simuler les requêtes HTTP @AutoConfigureMockMvc* @ActiveProfiles(“test”) : Utilise application-test.properties @ActiveProfiles(“test”) *@Transactional : Rollback automatique après chaque test @Transactional* MockMvc : Simule des requêtes HTTP sans démarrer le serveur mockMvc.perform(get(“/products/123”)) .andExpect(status().isOk()) .andExpect(jsonPath(“$.name”).value(“iPhone”)); *jsonPath : Parcourt la réponse JSON avec des expressions jsonPath(“$.name”)* Champ direct jsonPath(“$.category.name”) *Objet imbriqué jsonPath(“$.items[0].name”)* Premier élément d'un tableau jsonPath(“$.items”, hasSize(3)) *Taille du tableau </sxh> ==== 3.3 Test avec détection N+1 ==== <sxh java> import io.hypersistence.utils.jdbc.validator.SQLStatementCountValidator; import static io.hypersistence.utils.jdbc.validator.SQLStatementCountValidator.*; @Test @DisplayName(“GET /users/{id}/orders should not trigger N+1 queries”) void getUserOrders_ShouldNotTriggerNPlusOne() throws Exception {* Given User user = userRepository.save(new User(“John”, “john@example.com”)); for (int i = 0; i < 5; i++) { Order order = new Order(user); order.addItem(new OrderItem(product1, 1, product1.getPrice())); order.addItem(new OrderItem(product2, 2, product2.getPrice())); orderRepository.save(order); } *When SQLStatementCountValidator.reset(); mockMvc.perform(get(“/users/{id}/orders”, user.getId())) .andExpect(status().isOk()) .andExpect(jsonPath(“$”, hasSize(5))) .andExpect(jsonPath(“$[0].items”, hasSize(2)));* Then - Vérifier le nombre de requêtes SQL assertSelectCount(2); *1 pour User + 1 pour Orders avec items (JOIN FETCH) } </sxh>*



Exercice 3 (1h) :

Créer `UserControllerIntegrationTest` et `OrderControllerIntegrationTest` avec :

UserController (30min) :



1. `POST /users` - Création valide → 201
2. `POST /users` - Email déjà utilisé → 409 Conflict
3. `GET /users/{id}` - Utilisateur existant → 200
4. `GET /users/{id}` - Utilisateur inexistant → 404
5. `GET /users/{id}/recommendations` - Retourne des produits → 200

OrderController (30min) :

1. `POST /orders` - Création valide → 201
2. `POST /orders` - Stock insuffisant → 400
3. `GET /orders/{id}` - Commande existante → 200
4. `GET /users/{userId}/orders` - Historique → 200

5. **Bonus :** Test N+1 sur l'historique des commandes

Critères de validation :



- ☐ Tous les tests passent (`mvn verify`)
- ☐ `@BeforeEach` pour préparer les données
- ☐ Vérification des codes HTTP corrects
- ☐ Vérification du contenu JSON retourné
- ☐ Au moins 1 test de performance (N+1)

===== Partie 4 : Couverture de code avec JaCoCo (20min) ===== ==== 4.1 Configuration Maven ==== <sxh xml> <!– pom.xml –> <build> <plugins> <!– JaCoCo pour la couverture de code –> <plugin> <groupId>org.jacoco</groupId> <artifactId>jacoco-maven-plugin</artifactId> <version>0.8.11</version> <executions> <execution> <goals> <goal>prepare-agent</goal> </goals> </execution> <execution> <id>report</id> <phase>test</phase> <goals> <goal>report</goal> </goals> </execution> <execution> <id>jacoco-check</id> <goals> <goal>check</goal> </goals> <configuration> <rules> <rule> <element>PACKAGE</element> <limits> <limit> <counter>LINE</counter> <value>COVEREDRATIO</value> <minimum>0.70</minimum> </limit> </limits> </rule> </rules> </configuration> </execution> </executions> </plugin> <!– Surefire pour les tests unitaires –> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-surefire-plugin</artifactId> <version>3.2.2</version> <configuration> <includes> <include>**/*Test.java</include> </includes> <excludes> <exclude>**/*IntegrationTest.java</exclude> </excludes> </configuration> </plugin> <!– Failsafe pour les tests d'intégration –> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-failsafe-plugin</artifactId> <version>3.2.2</version> <configuration> <includes> <include>**/*IntegrationTest.java</include> </includes> </configuration> <executions> <execution> <goals> <goal>integration-test</goal> <goal>verify</goal> </goals> </execution> </executions> </plugin> </plugins> </build> </sxh> ==== 4.2 Commandes Maven ==== <sxh bash;gutter:false> # Tests unitaires uniquement (rapides) mvn clean test # Tests unitaires + rapport de couverture mvn clean test jacoco:report # Tous les tests (unitaires + intégration) mvn clean verify # Voir le rapport de couverture open target/site/jacoco/index.html </sxh> ==== 4.3 Exclusion de certaines classes ==== <sxh xml> <configuration> <excludes> <!– Exclure les entités JPA –> <exclude>**/domain/**</exclude> <!– Exclure les DTOs –> <exclude>**/dto/**</exclude> <!– Exclure la classe main –> <exclude>**/EcommerceApplication.class</exclude> </excludes> </configuration> </sxh>

Exercice 4 (10min) :



1. Ajouter la configuration JaCoCo dans `pom.xml`
2. Lancer `mvn clean test jacoco:report`
3. Ouvrir `target/site/jacoco/index.html` dans un navigateur
4. Identifier les classes avec une couverture < 70%
5. Ajouter des tests pour améliorer la couverture

===== Partie 5 : GitHub Actions - Pipeline CI/CD complète (40min) ===== ==== 5.1 Workflow complet ==== Créer `.github/workflows/ci.yml` : <sxh yaml> name: CI/CD Pipeline on: push: branches: [main, develop] pull_request: branches: [main, develop] jobs: # Job 1 : Tests unitaires (rapides) unit-tests: name: Unit Tests runs-on: ubuntu-latest steps: - name: Checkout code uses: actions/checkout@v4 - name: Set up JDK 21 uses: actions/setup-java@v4 with: java-version: '21' distribution: 'temurin' cache: maven - name: Run unit tests run: mvn clean test - name: Upload test results if: always() uses: actions/upload-artifact@v3 with: name: unit-test-

results path: target/surefire-reports/ # Job 2 : Tests d'intégration (plus longs) integration-tests: name: Integration Tests runs-on: ubuntu-latest needs: unit-tests # Attend que les tests unitaires passent steps: - name: Checkout code uses: actions/checkout@v4 - name: Set up JDK 21 uses: actions/setup-java@v4 with: java-version: '21' distribution: 'temurin' cache: maven - name: Run integration tests run: mvn clean verify -DskipUnitTests - name: Upload test results if: always() uses: actions/upload-artifact@v3 with: name: integration-test-results path: target/failsafe-reports/ # Job 3 : Analyse de couverture coverage: name: Code Coverage runs-on: ubuntu-latest needs: integration-tests steps: - name: Checkout code uses: actions/checkout@v4 - name: Set up JDK 21 uses: actions/setup-java@v4 with: java-version: '21' distribution: 'temurin' cache: maven - name: Generate coverage report run: mvn clean verify jacoco:report - name: Upload coverage to Codecov uses: codecov/codecov-action@v3 with: files: ./target/site/jacoco/jacoco.xml flags: unittests name: codecov-umbrella fail_ci_if_error: false - name: Upload JaCoCo report uses: actions/upload-artifact@v3 with: name: jacoco-report path: target/site/jacoco/ # Job 4 : Build (optionnel - pour vérifier que l'app compile) build: name: Build Application runs-on: ubuntu-latest needs: coverage steps: - name: Checkout code uses: actions/checkout@v4 - name: Set up JDK 21 uses: actions/setup-java@v4 with: java-version: '21' distribution: 'temurin' cache: maven - name: Build with Maven run: mvn clean package -DskipTests - name: Upload artifact uses: actions/upload-artifact@v3 with: name: ecommerce-api path: target/*.jar </sxh> ==== 5.2 Configuration pour séparer les tests ==== <sxh xml> <!– pom.xml - Ajout de propriétés –> <properties> <skipUnitTests>false</skipUnitTests> <skipIntegrationTests>false</skipIntegrationTests> </properties> <build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-surefire-plugin</artifactId> <configuration> <skipTests>${skipUnitTests}</skipTests> </configuration> </plugin> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-failsafe-plugin</artifactId> <configuration> <skipTests>${skipIntegrationTests}</skipTests> </configuration> </plugin> </plugins> </build> </sxh> ==== 5.3 Badges pour le README ==== <sxh markdown;gutter:false> # E-Commerce API ![CI/CD](https://github.com/VOTRE-USERNAME/VOTRE-REPO/actions/workflows/ci.yml/badge.svg) ![Coverage](https://codecov.io/gh/VOTRE-USERNAME/VOTRE-REPO/branch/main/graph/badge.svg) ## Description API REST pour un système e-commerce avec Spring Boot 3. ## Badges de statut - **Build** : Statut de la compilation - **Tests** : Résultat des tests automatisés - **Coverage** : Pourcentage de code couvert par les tests ## Commandes ```bash # Tests unitaires uniquement mvn test # Tests d'intégration uniquement mvn verify -DskipUnitTests # Tous les tests mvn verify # Rapport de couverture mvn test jacoco:report ``` </sxh> ==== 5.4 Protection de branche (optionnel) ==== Dans GitHub : - **Settings** → **Branches** → **Add rule** - Branch name pattern : `main` - Cocher : * ☑ Require status checks to pass before merging * ☑ Require branches to be up to date before merging * Sélectionner : `Unit Tests`, `Integration Tests`, `Code Coverage` - **Create**

Exercice 5 (30min) :

1. Créer le fichier `.github/workflows/ci.yml`
2. Commit et push sur GitHub
3. Vérifier dans l'onglet “Actions” que les 4 jobs s'exécutent
4. Ajouter les badges dans `README.md`
5. **(Bonus)** Configurer la protection de branche `main`
6. **(Bonus)** Créer une PR et vérifier que les tests sont obligatoires

Vérifications :

- ☐ Les tests unitaires passent en < 1min
- ☐ Les tests d'intégration passent en < 3min
- ☐ Le rapport de couverture est généré
- ☐ L'artifact `.jar` est uploadé

===== Récapitulatif des commandes ===== <sxh bash;gutter:false> # ========== Développement local ========== # Lancer l'app en mode dev mvn spring-boot:run -Dspring-boot.run.profiles=dev # Lancer l'app en mode test mvn spring-boot:run -Dspring-boot.run.profiles=test # ========== Tests ========== # Tests unitaires uniquement (rapides, < 10s) mvn clean test # Tests d'intégration uniquement mvn clean verify -DskipUnitTests # Tous les tests mvn clean verify # Tests avec rapport de couverture mvn clean test jacoco:report # ========== Couverture de code ========== # Générer le rapport JaCoCo mvn jacoco:report # Ouvrir le rapport open target/site/jacoco/index.html # Vérifier le seuil de couverture mvn jacoco:check # ========== Build ========== # Compiler sans tests mvn clean package -DskipTests # Build complet mvn clean install </sxh> ===== Livrables attendus (réaliste pour 4h) =====

Must have (priorité absolue)

Configuration (30min) :

- ☐ 4 profils configurés : commun, dev, test, prod
- ☐ Application démarre avec chaque profil
- ☐ Variables d'environnement documentées

Tests unitaires (1h15) :

- ☐ `ProductServiceTest` complet (≥5 tests)
- ☐ `UserServiceTest` complet (≥5 tests)
- ☐ Utilisation correcte des mocks
- ☐ Tests paramétrés pour au moins 1 cas
- ☐ Tous les tests passent (`mvn test`)



Tests d'intégration (1h15) :



- ☐ `ProductControllerIntegrationTest` complet (≥6 tests)
- ☐ `UserControllerIntegrationTest` complet (≥5 tests)
- ☐ `OrderControllerIntegrationTest` complet (≥4 tests)
- ☐ Vérification des codes HTTP (200, 201, 400, 404, 409)
- ☐ Au moins 1 test de détection N+1
- ☐ Tous les tests passent (`mvn verify`)

Couverture + CI/CD (1h) :

- ☐ JaCoCo configuré avec seuil minimum 70%
- ☐ Workflow GitHub Actions complet (4 jobs)
- ☐ Séparation unit tests / integration tests
- ☐ Badges CI/CD dans le README
- ☐ Pipeline qui passe au vert sur GitHub

Nice to have (bonus si temps)

- Protection de branche `main` configurée
- Tests de contrat avec Spring Cloud Contract
- SonarCloud intégré pour la qualité de code
- Tests de charge basiques avec JMeter
- Documentation Swagger/OpenAPI testée

===== Aide-mémoire : Différences clés ===== ^ Aspect ^ Test Unitaire ^ Test d'Intégration ^ | **Vitesse** | ⚡ Très rapide (<10ms) | 🐢 Plus lent (100-500ms) | | **Base de données** | 🚫 Non (mocks)

| ✅ Oui (H2 en mémoire) | | **Contexte Spring** | ❌ Non | ✅ Oui (toute l'app) | | **Annotations** | `@ExtendWith(MockitoExtension.class)` | `@SpringBootTest` | | **Ce qu'on teste** | Logique métier isolée | Flux complet de bout en bout | | **Quand ça échoue** | Bug dans la logique | Bug d'intégration/config | | **Commande Maven** | `mvn test` | `mvn verify` | | **Fichier de tests** | `*Test.java` | `*IntegrationTest.java` | ===== Bonnes pratiques à retenir =====





Tests unitaires

- ✅ **Rapides** : < 10ms par test
- ✅ **Isolés** : pas de dépendances externes (DB, réseau)
- ✅ **AAA Pattern** : Arrange, Act, Assert
- ✅ **1 test = 1 comportement** : ne pas tester plusieurs choses
- ✅ **Nommage explicite** : `methodName_WhenCondition_ShouldExpectedBehavior`
- ✅ **Mocks minimalistes** : seulement les dépendances nécessaires

Tests d'intégration

- ✅ **Réalistes** : données de test cohérentes
- ✅ **Nettoyage** : `@Transactional` ou `@BeforeEach` avec `deleteAll()`
- ✅ **Vérifications complètes** : code HTTP + contenu + headers
- ✅ **Performance** : détecter les N+1 avec Hypersistence
- ✅ **Cas d'erreur** : tester les 400, 404, 409, 500

CI/CD

- ✅ **Fail fast** : tests unitaires avant intégration
- ✅ **Parallélisation** : jobs indépendants
- ✅ **Artifacts** : conserver les rapports et JARs
- ✅ **Protection** : branche `main` protégée
- ✅ **Documentation** : badges visibles

===== Ressources essentielles ===== * Spring Boot Testing - Documentation officielle * JUnit 5 User Guide * Mockito Documentation * AssertJ Documentation * JaCoCo Maven Plugin * GitHub Actions - Documentation * Baeldung - Testing in Spring Boot

[1)]
) → productService.getProduct(productId
[2)]
) → productService.updateStock(productId, -10
[3)]
) → service.doSomething(
[4)]
) → productService.createProduct(dto

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/eadl/bloc3/dev_av/td3?rev=1761667564

Last update: **2025/10/28 17:06**