3 - Tests et CI/CD

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 ?
- Ceux qui ont fini peuvent commencer les tests, les autres finalisent le TD2

Partie 1 : Configuration multi-environnements (30min)

1.1 Stratégie de profils Spring

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

Profiles

La création de profiles permet de gérer des configurations différentes, et des fichiers de configuration spécifiques à chaque profile.

pom.xml - Configuration des profils Maven

```
<profiles>
    <profile>
        <id>dev</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <activeProfile>dev</activeProfile>
        </properties>
```

```
    </profile>
    <profile>
        <id>test</id>
        <properties>
            <activeProfile>test</activeProfile>
        </properties>
    </profile>
    <profile>
        <id>prod</id>
        <properties>
            <activeProfile>prod</activeProfile>
        </properties>
    </profile>
</profiles>
```

application.properties (commun)

```
# Récupération du profile Maven pour def du profile Spring
spring.profiles.active=@activeProfile@

# 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
```

application-dev.properties

```
# 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
```

application-test.properties

```
# 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
```

application-prod.properties

```
# 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
```

1.2 Activation des profils

```
# 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
```

```
# Avec profil maven
mvn spring-boot:run -P dev
```

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)

```
<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>
    <!-- MockK pour Kotlin (optionnel, alternative à Mockito) -->
    <dependency>
        <groupId>com.ninja-squad</groupId>
        <artifactId>springmockk</artifactId>
        <version>4.0.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2.2 Premier test simple : ProductService

```
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 io.mockk.every
import io.mockk.mockk
```

```
import io.mockk.verify
import io.mockk.slot
import org.assertj.core.api.Assertions.*
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.DisplayName
import org.junit.jupiter.api.BeforeEach
import java.math.BigDecimal
import java.util.*

@DisplayName("ProductService - Unit Tests")
class ProductServiceTest {

    private lateinit var productRepository: ProductRepository
    private lateinit var productService: ProductService
    private lateinit var category: Category

    @BeforeEach
    fun setUp() {
        productRepository = mockk()
        productService = ProductService(productRepository)
        category = Category(name = "Electronics", description = "Electronic
products")
        category.id = UUID.randomUUID()
    }

    @Test
    @DisplayName("Should create product with valid data")
    fun `createProduct with valid data should return product`() {
        // Given
        val productSlot = slot<Product>()
        val savedProduct = Product(
            name = "iPhone 15",
            price = BigDecimal("999.99"),
            stock = 10,
            category = category
        ).apply { id = UUID.randomUUID() }

        every { productRepository.save(capture(productSlot)) } returns savedProduct

        // When
        val result = productService.createProduct(
            name = "iPhone 15",
            price = BigDecimal("999.99"),
            stock = 10,
            categoryId = category.id!!
        )

        // Then
        assertThat(result).isNotNull
        assertThat(result.name).isEqualTo("iPhone 15")
        assertThat(result.price).isEqualByComparingTo("999.99")
        assertThat(result.stock).isEqualTo(10)
        verify(exactly = 1) { productRepository.save(any()) }
    }

    @Test
```

```
    @DisplayName("Should throw exception when product not found")
    fun `getProduct when not exists should throw exception`() {
        // Given
        val productId = UUID.randomUUID()
        every { productRepository.findById(productId) } returns Optional.empty()

        // When & Then
        assertThatThrownBy { productService.getProduct(productId) }
            .isInstanceOf(ProductNotFoundException::class.java)
            .hasMessageContaining(productId.toString())
    }

    @Test
    @DisplayName("Should decrease stock when sufficient")
    fun `decreaseStock with sufficient stock should update product`() {
        // Given
        val productId = UUID.randomUUID()
        val product = Product(
            name = "Test Product",
            price = BigDecimal("10.00"),
            stock = 10,
            category = category
        ).apply { id = productId }

        every { productRepository.findById(productId) } returns
Optional.of(product)
        every { productRepository.save(any()) } returns product

        // When
        productService.decreaseStock(productId, 3)

        // Then
        assertThat(product.stock).isEqualTo(7)
        verify(exactly = 1) { productRepository.save(product) }
    }

    @Test
    @DisplayName("Should throw exception when insufficient stock")
    fun `decreaseStock with insufficient stock should throw exception`() {
        // Given
        val productId = UUID.randomUUID()
        val product = Product(
            name = "Test Product",
            price = BigDecimal("10.00"),
            stock = 2,
            category = category
        ).apply { id = productId }

        every { productRepository.findById(productId) } returns
Optional.of(product)

        // When & Then
        assertThatThrownBy { productService.decreaseStock(productId, 5) }
            .isInstanceOf(InsufficientStockException::class.java)
            .hasMessageContaining("Insufficient stock")
    }
```

```
    @Test
    @DisplayName("Should update stock correctly")
    fun `updateStock should change product stock and save`() {
        // Given
        val productId = UUID.randomUUID()
        val product = Product(
            name = "Test Product",
            price = BigDecimal("10.00"),
            stock = 10,
            category = category
        ).apply { id = productId }

        every { productRepository.findById(productId) } returns
Optional.of(product)
        every { productRepository.save(any()) } returns product

        // When
        productService.updateStock(productId, 5)

        // Then
        assertThat(product.stock).isEqualTo(15)
        verify(exactly = 1) { productRepository.save(product) }
    }
}
```

2.3 Concepts clés

```
// MockK pour Kotlin (alternative à Mockito)
import io.mockk.*

// Créer un mock
val repository = mockk<ProductRepository>()

// Définir le comportement du mock
every { repository.findById(id) } returns Optional.of(product)
every { repository.save(any()) } returns product

// Capturer un argument
val productSlot = slot<Product>()
every { repository.save(capture(productSlot)) } returns product

// Vérifier qu'une méthode a été appelée
verify(exactly = 1) { repository.save(any()) }
verify(atLeast = 1) { repository.findById(any()) }
verify(exactly = 0) { repository.delete(any()) }

// AssertJ - Assertions plus lisibles
assertThat(result.stock).isEqualTo(7)
assertThat(result).isNotNull()
assertThat(list).hasSize(3)
assertThat(price).isEqualByComparingTo("999.99")

// Vérifier qu'une exception est levée
```

```
assertThatThrownBy { service.doSomething() }
    .isInstanceOf(MyException::class.java)
    .hasMessage("Expected message")
    .hasMessageContaining("partial")
```

2.4 Tests paramétrés (en plus)

```
import org.junit.jupiter.params.ParameterizedTest
import org.junit.jupiter.params.provider.CsvSource
import org.junit.jupiter.params.provider.ValueSource

class ProductValidationTest {

    @ParameterizedTest
    @DisplayName("Should validate price is positive")
    @ValueSource(strings = ["-10.00", "-0.01", "0.00"])
    fun `createProduct with invalid price should throw exception`(price: String) {
        // Given
        val productService = ProductService(mockk())

        // When & Then
        assertThatThrownBy {
            productService.createProduct(
                name = "Test",
                price = BigDecimal(price),
                stock = 10,
                categoryId = UUID.randomUUID()
            )
        }.isInstanceOf(InvalidPriceException::class.java)
    }

    @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"
    )
    fun `calculateTotal with different quantities should return correct amount`(
        quantity: Int,
        unitPrice: String,
        expectedTotal: String
    ) {
        // Given
        val product = Product(
            name = "Test",
            price = BigDecimal(unitPrice),
            stock = 100,
            category = mockk()
        )
        val productService = ProductService(mockk())

        // When
```

```
        val total = productService.calculateTotal(product, quantity)

        // Then
        assertThat(total).isEqualByComparingTo(expectedTotal)
    }
}
```

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 :

```
@DisplayName("UserService - Unit Tests")
class UserServiceTest {
    private lateinit var userRepository: UserRepository
    private lateinit var orderRepository: OrderRepository
    private lateinit var userService: UserService
    @BeforeEach
    fun setUp() {
        userRepository = mockk()
        orderRepository = mockk()
        userService = UserService(userRepository, orderRepository)
    }
    @Test
    @DisplayName("Should create user with valid data")
    fun `createUser with valid data should return user`() {
        // Given
        val email = "john@example.com"
        val user = User(name = "John Doe", email = email).apply {
            id = UUID.randomUUID()
        }
        every { userRepository.existsByEmail(email) } returns false
        every { userRepository.save(any()) } returns user

        // When
        val result = userService.createUser(name = "John Doe", email =
email)

        // Then
        assertThat(result).isNotNull()
        assertThat(result.email).isEqualTo(email)
        verify(exactly = 1) { userRepository.save(any()) }
    }
    // 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

```
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.hamcrest.Matchers.*
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.*
import org.springframework.transaction.annotation.Transactional
import java.math.BigDecimal

@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("test")
@Transactional
@DisplayName("ProductController - Integration Tests")
class ProductControllerIntegrationTest {

    @Autowired
    private lateinit var mockMvc: MockMvc

    @Autowired
    private lateinit var productRepository: ProductRepository

    @Autowired
    private lateinit var categoryRepository: CategoryRepository
```

```
    private lateinit var electronics: Category

    @BeforeEach
    fun setUp() {
        productRepository.deleteAll()
        categoryRepository.deleteAll()
        electronics = categoryRepository.save(
            Category(
                name = "Electronics",
                description = "Electronic products"
            )
        )
    }

    @Test
    @DisplayName("POST /products should create a new product")
    fun `createProduct should return 201 with created product`() {
        // Given
        val requestBody = """
            {
                "name": "iPhone 15",
                "price": 999.99,
                "stock": 10,
                "categoryId": "${electronics.id}"
            }
        """.trimIndent()

        // When & Then
        mockMvc.post("/products") {
            contentType = MediaType.APPLICATION_JSON
            content = requestBody
        }.andExpect {
            status { isCreated() }
            jsonPath("$.name") { value("iPhone 15") }
            jsonPath("$.price") { value(999.99) }
            jsonPath("$.stock") { value(10) }
            jsonPath("$.category.name") { value("Electronics") }
        }
    }

    @Test
    @DisplayName("GET /products/{id} should return product when exists")
    fun `getProduct when exists should return 200 with product`() {
        // Given
        val product = productRepository.save(
            Product(
                name = "MacBook Pro",
                price = BigDecimal("1999.99"),
                stock = 5,
                category = electronics
            )
        )

        // When & Then
        mockMvc.get("/products/${product.id}")
            .andExpect {
```

```
                status { isOk() }
                jsonPath("$.id") { value(product.id.toString()) }
                jsonPath("$.name") { value("MacBook Pro") }
                jsonPath("$.price") { value(1999.99) }
            }
    }

    @Test
    @DisplayName("GET /products/{id} should return 404 when not found")
    fun `getProduct when not exists should return 404`() {
        // Given
        val nonExistentId = java.util.UUID.randomUUID()

        // When & Then
        mockMvc.get("/products/$nonExistentId")
            .andExpect {
                status { isNotFound() }
            }
    }

    @Test
    @DisplayName("PUT /products/{id} should update product")
    fun `updateProduct should return 200 with updated product`() {
        // Given
        val product = productRepository.save(
            Product(
                name = "iPad",
                price = BigDecimal("699.99"),
                stock = 15,
                category = electronics
            )
        )

        val updateRequest = """
            {
                "name": "iPad Pro",
                "price": 899.99,
                "stock": 20
            }
        """.trimIndent()

        // When & Then
        mockMvc.put("/products/${product.id}") {
            contentType = MediaType.APPLICATION_JSON
            content = updateRequest
        }.andExpect {
            status { isOk() }
            jsonPath("$.name") { value("iPad Pro") }
            jsonPath("$.price") { value(899.99) }
            jsonPath("$.stock") { value(20) }
        }
    }

    @Test
    @DisplayName("DELETE /products/{id} should delete product")
    fun `deleteProduct should return 204`() {
```

```
        // Given
        val product = productRepository.save(
            Product(
                name = "AirPods",
                price = BigDecimal("199.99"),
                stock = 50,
                category = electronics
            )
        )

        // When & Then
        mockMvc.delete("/products/${product.id}")
            .andExpect {
                status { isNoContent() }
            }

        // Verify deletion
        mockMvc.get("/products/${product.id}")
            .andExpect {
                status { isNotFound() }
            }
    }

    @Test
    @DisplayName("GET /products should return paginated list")
    fun `getProducts should return paginated results`() {
        // Given
        productRepository.save(
            Product(
                name = "iPhone",
                price = BigDecimal("999"),
                stock = 10,
                category = electronics
            )
        )
        productRepository.save(
            Product(
                name = "MacBook",
                price = BigDecimal("1999"),
                stock = 5,
                category = electronics
            )
        )

        // When & Then
        mockMvc.get("/products") {
            param("page", "0")
            param("size", "10")
        }.andExpect {
            status { isOk() }
            jsonPath("$.content") { isArray() }
            jsonPath("$.content.length()") { value(2) }
            jsonPath("$.totalElements") { value(2) }
        }
    }
```

```
    @Test
    @DisplayName("GET /products should filter by category")
    fun `getProducts with category filter should return filtered list`() {
        // Given
        val books = categoryRepository.save(
            Category("Books", "Books category")
        )
        productRepository.save(
            Product(
                name = "iPhone",
                price = BigDecimal("999"),
                stock = 10,
                category = electronics
            )
        )
        productRepository.save(
            Product(
                name = "Java Book",
                price = BigDecimal("50"),
                stock = 20,
                category = books
            )
        )

        // When & Then
        mockMvc.get("/products") {
            param("categoryId", electronics.id.toString())
        }.andExpect {
            status { isOk() }
            jsonPath("$.content.length()") { value(1) }
            jsonPath("$.content[0].name") { value("iPhone") }
        }
    }
}
```

3.2 Concepts clés

```
// @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 DSL Kotlin : Simule des requêtes HTTP sans démarrer le serveur
mockMvc.get("/products/123")
    .andExpect {
        status { isOk() }
```

```
        jsonPath("$.name") { value("iPhone") }
    }

mockMvc.post("/products") {
    contentType = MediaType.APPLICATION_JSON
    content = jsonBody
}.andExpect {
    status { isCreated() }
}

// 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.length()")    // Taille du tableau
```

3.3 Test avec détection N+1

```
import io.hypersistence.utils.jdbc.validator.SQLStatementCountValidator
import io.hypersistence.utils.jdbc.validator.SQLStatementCountValidator.*

@Test
@DisplayName("GET /users/{id}/orders should not trigger N+1 queries")
fun `getUserOrders should not trigger NPlusOne`() {
    // Given
    val user = userRepository.save(User("John", "john@example.com"))
    repeat(5) {
        val order = Order(user = user)
        order.addItem(OrderItem(product1, 1, product1.price))
        order.addItem(OrderItem(product2, 2, product2.price))
        orderRepository.save(order)
    }

    // When
    SQLStatementCountValidator.reset()
    mockMvc.get("/users/${user.id}/orders")
        .andExpect {
            status { isOk() }
            jsonPath("$.length()") { value(5) }
            jsonPath("$[0].items.length()") { value(2) }
        }

    // Then - Vérifier le nombre de requêtes SQL
    assertSelectCount(2) // 1 pour User + 1 pour Orders avec items (JOIN FETCH)
}
```

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

```
<build>
    <plugins>
        <!-- JaCoCo Maven Plugin -->
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.11</version>
            <executions>
                <!-- Préparation de l'agent JaCoCo -->
                <execution>
                    <id>prepare-agent</id>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
                <!-- Génération du rapport après les tests -->
                <execution>
                    <id>report</id>
                    <phase>test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                </execution>
                <!-- Vérification du seuil minimum de couverture -->
                <execution>
```

```
                    <id>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>
            <configuration>
                <excludes>
                    <!-- Exclure les entités JPA -->
                    <exclude>**/domain/**</exclude>
                    <!-- Exclure les DTOs -->
                    <exclude>**/dto/**</exclude>
                    <!-- Exclure la classe main -->
                    <exclude>**/EcommerceApplicationKt.class</exclude>
                    <!-- Exclure les configurations -->
                    <exclude>**/config/**</exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>
```

4.2 Commandes Maven

```
# Exécuter les tests et générer le rapport
mvn clean test jacoco:report

# Vérifier que le seuil de couverture est atteint
mvn jacoco:check

# Tous les tests (unitaires + intégration)
mvn clean verify

# Voir le rapport de couverture
open target/site/jacoco/index.html
```

4.3 Exclusion de certaines classes

Déjà configuré dans la section précédente. Les exclusions communes pour un projet Kotlin :

- Entités JPA (`domain/*`)
- DTOs (`dto/*`)
- Configuration (`config/*`)
- Classe main (`*ApplicationKt.class`)

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` :

```
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 -DskipIntegrationTests
```

```
      - 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
  integration-tests:
    name: Integration Tests
    runs-on: ubuntu-latest
    needs: unit-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: 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 : Couverture de code
  code-coverage:
    name: Code Coverage
    runs-on: ubuntu-latest
    needs: [unit-tests, 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: Run tests with coverage
        run: mvn clean verify jacoco:report

      - name: Check coverage threshold
        run: mvn jacoco:check

      - name: Upload coverage report
```

```
        uses: actions/upload-artifact@v3
        with:
          name: coverage-report
          path: target/site/jacoco/

      - name: Comment PR with coverage
        if: github.event_name == 'pull_request'
        uses: codecov/codecov-action@v3
        with:
          files: target/site/jacoco/jacoco.xml
          fail_ci_if_error: true

  # Job 4 : Build et packaging
  build:
    name: Build Application
    runs-on: ubuntu-latest
    needs: [unit-tests, integration-tests, code-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 JAR
        uses: actions/upload-artifact@v3
        with:
          name: ecommerce-api
          path: target/*.jar
```

5.2 Séparation des tests (pom.xml)

```
<properties>
    <skipUnitTests>false</skipUnitTests>
    <skipIntegrationTests>false</skipIntegrationTests>
</properties>

<build>
    <plugins>
        <!-- Tests unitaires avec Surefire -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <skipTests>${skipUnitTests}</skipTests>
                <includes>
```

```
                    <include>**/*Test.kt</include>
                    <include>**/*Test.class</include>
                </includes>
                <excludes>
                    <exclude>**/*IntegrationTest.kt</exclude>
                    <exclude>**/*IntegrationTest.class</exclude>
                </excludes>
            </configuration>
        </plugin>

        <!-- Tests d'intégration avec Failsafe -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <configuration>
                <skipTests>${skipIntegrationTests}</skipTests>
                <includes>
                    <include>**/*IntegrationTest.kt</include>
                    <include>**/*IntegrationTest.class</include>
                </includes>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

5.3 Badges pour le README

```
# E-Commerce API

![CI/CD](https://github.com/VOTRE-USERNAME/VOTRE-REPO/actions/workflows/ci.yml/badg
e.svg)
![Coverage](https://codecov.io/gh/VOTRE-USERNAME/VOTRE-REPO/branch/main/graph/badge
.svg)
![Kotlin](https://img.shields.io/badge/Kotlin-1.9.22-blue.svg)
![Spring Boot](https://img.shields.io/badge/Spring%20Boot-3.2.0-green.svg)

## Description
API REST pour un système e-commerce avec Spring Boot 3 et Kotlin.

## 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
```

Technologies

- Kotlin 1.9.22
- Spring Boot 3.2.0
- Spring Data JPA
- H2 Database
- JUnit 5
- MockK
- JaCoCo

5.4 Protection de branche (optionnel)

Dans GitHub :

1. **Settings → Branches → Add rule**
2. Branch name pattern : `main`
3. Cocher :
   - ☐ Require status checks to pass before merging
   - ☐ Require branches to be up to date before merging
   - Sélectionner : `Unit Tests`, `Integration Tests`, `Code Coverage`
4. **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

```
# ========== 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

mvn spring-boot:run -P 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
```

Livrables attendus

A faire en priorité

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 (MockK)
- 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

En +, Si vous avez le 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	Pas d'annotation Spring	`@SpringBootTest`
Mocking	MockK / Mockito	Vrai composants Spring
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.kt`	`*IntegrationTest.kt`

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** : backticks pour noms descriptifs en Kotlin
- **Mocks minimalistes** : seulement les dépendances nécessaires
- **MockK pour Kotlin** : meilleure intégration que Mockito

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
- **DSL Kotlin** : utiliser les extensions MockMvc pour Kotlin

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
- MockK Documentation (Kotlin)
- Mockito Documentation
- AssertJ Documentation
- JaCoCo Maven Plugin
- GitHub Actions - Documentation
- Baeldung - Testing in Spring Boot with Kotlin
- Kotlin Testing with JUnit