JUnit et tests unitaires

M2I Formations 2022

Samih Habbani

Objectifs de la formation

- Mettre en place un scénario de tests unitaires
- Aborder les concepts clés de Junit5
- Configurer un projet avec Maven et Junit5
- TDD - Test Driven Development
- Approche du framework Mockito

Les points de la formation

- Qu'est-ce qu'un test unitaire?
- Pourquoi mettre en place des tests unitaires?
- F.I.R.S.T Principles
- Isolation du code
- Pyramide de tests
- Introduction à Junit 5
- Concepts avancés en Junit 5
- Test Driven Development
- Introduction au framework Mockito

Pré-requis pour ce cours

- Avoir des connaissances en JAVA
- Avoir des connaissances en POO

A qui est destiné ce cours

- Développeurs full-stack
- Architectes

Junit 5 & Mockito

Créer des tests unitaires

1.

Introduction aux tests unitaires

Qu’est-ce qu’un test unitaire?

- Petite méthode qui permet de tester son code
- Petit morceau de code qu’on met en place de manière isolée pour tester du code
- Il valide toujours un résultat attendu
- Simple
- Qui teste une méthode en particulier dans notre code
- JUnit5 pour le faire
- Dans un projet Java on a plusieurs classes avec plusieurs méthodes, on va mettre en place des scénarios de tests pour toutes ces méthodes de classe
- On le fait pas manuellement car l’objectif des tests unitaires c’est d’éviter les régressions

Pourquoi écrire des tests unitaires?

- Pour éviter les régressions de code
- Pour tester l'ensemble de notre code à chaque build
- Pour s'assurer que le retour attendu pour chacune de nos méthodes est celui que l'on attend
- Test Driven Development - développer étape par étape en contrôlant le développement
- Etre sur le code fonctionne
- Qu'il fonctionne ou pas avec des paramètres valides ou invalides
- Qu'il fonctionne maintenant et dans le futur

F.I.R.S.T principles

- Independant - Nos tests unitaires sont indépendants des uns des autres
- Repeatable - Doivent pouvoir être exécutés plusieurs fois et produire le même résultat
- Self-Validating - Testent eux-mêmes le code et décident de la validation des résultats
- Thorough & Timely - On doit pouvoir tester nos méthodes avec des paramètres valides ou invalides avec des valeurs min et max. Timely, signifie que l'on développer nos méthodes de tests pendant qu'on développe le code : objectif avoir des tests complets

Isolation du code

- Un test unitaire va tester une méthode dans une classe A mais cette méthode instancie un objet de Type classe B et fait appel à des méthodes de la classe B.
- Pour éviter ce scénario, on doit isoler le code que l'on test
- Comment ? Injection de dépendance
- On va injecter lors de nos tests dans la méthode de la classe A des objets Mock ou Fake de type classe B pour isoler le code et ne tester que la méthode de la classe A.
- Objets Mock ou Fake (Framework Mockito c'est son rôle)

Pyramide de tests

- 3 types de tests :
- - Tests unitaires - on test un morceau de code isolé avec des dépendances fake ou mock
- - Test d'intégration - le code de l'application est testé sans être mocké donc avec une BDD et des connections HTTP
- - Tests End to End / Test UI - tester les fonctionnalités du début jusqu'à la fin d'une application

QUIZ 1

Créer des tests unitaires

Junit 5 - c’est quoi?

- Junit plateform + Junit Jupiter + Junit Vintage
- Junit plateform - fondation qui permet de lancer des tests sous des frameworks dans la JVM
- Junit Jupiter - combinaison de nouveaux programmes de modèle et de modèle d’extension qui permettent d’écrire des tests c’est une extension de Junit5
- Junit Vintage - permet de tester des méthodes de test plus ancienne en Junit 3 et Junit 4

Junit 5 et outils de build

- IntelliJ
- Maven

Installations

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
```

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M6</version>
</plugin>
```

Lancer le build avec exécution des tests

- mvn package

2.

Concepts de base en Junit5

Créer une première méthode de test

```
    @Test
    void testDemo() {
        System.out.println("Demo Test");
    }
```

Assertions et messages

```
assertEquals(expectedResult, result, "4/2 did not produce 2");
```

Autres assertions

- assertNull()
- assertThrows()
- assertEquals()
- fail()
- assertTrue()
- assertFalse()
- …

Lazy Assertion Messages

```
assertEquals(2, result, () -> number1 + " - " + number2 + " did not
produce " + expectedResult);
```

Nommage de tests unitaires

```
@Test
void
testIntegerDivision_WhenDividendIsDividedByZero_shouldThrownArithmet
icException() {}
```

Annotation @DisplayName

Permet de nommer nos méthodes de tests.

```
@DisplayName("Division by zero")
@Test
void
testIntegerDivision_WhenDividendIsDividedByZero_shouldThrownArithmet
icException() {}
```

Structure de méthode de test - Arrange, Act, Assert

```
    @DisplayName("Test 4/2 = 2")
    @Test
    void testIntegerDivision_WhenFourDividedByTwo_ShouldReturnTwo_2() {

        // Arrange
         int number1 = 4;
         int number2 = 2;
         int expectedResult = 2;

         // Act
         int result = calculator.integerDivision(number1, number2);

         // Assert
         assertEquals(expectedResult, result, "4/2 did not produce 2");

    }
```

Junit 5 Lifecycle

```java
@BeforeAll
static void setup() {
    System.out.println("Executing @BeforeAll method");
}

@AfterAll
static void cleanup() {
    System.out.println("Executing @AfterAll method");
}

@BeforeEach
void beforeEachTestMethod() {
    calculator = new Calculator();
    System.out.println("Executing @BeforeEach method");
}

@AfterEach
void afterEachTestMethod() {
    System.out.println("Executing @AfterEach method");
}
```

Valider le retour d'une exception

```java
@BeforeAll
static void setup() {
    System.out.println("Executing @BeforeAll method");
}

@AfterAll
static void cleanup() {
    System.out.println("Executing @AfterAll method");
}

@BeforeEach
void beforeEachTestMethod() {
    calculator = new Calculator();
    System.out.println("Executing @BeforeEach method");
}

@AfterEach
void afterEachTestMethod() {
    System.out.println("Executing @AfterEach method");
}
```

QUIZ 2

Créer des tests unitaires

Désactiver les tests unitaires

```
@DisplayName("Division by zero")
@Test
void testIntegerDivision_WhenDividendIsDividedByZero_shouldThrownArithmeticException() {

    System.out.println("Running Division by zero");

    // Arrange
    int dividend = 4;
    int divisor = 0;
    String expectedExceptionMessage = "/ by zero";

    // Act / Assert
    ArithmeticException actualException = assertThrows(
            ArithmeticException.class,
            () -> {
                calculator.integerDivision(dividend, divisor);
            }, "Division by zero should have thrown an Arithmetic exception");

    assertEquals(expectedExceptionMessage, actualException.getMessage());

}
```

3.

Concepts avancés en Junit5

@Parameterized & @MethodSource

- Permet d'exécuter une même méthode de test avec plusieurs jeux de données différents
- Accompagné d'une méthode qui renvoie un stream de donnée

```
    @DisplayName("Test integer substraction [number1, number2, expectedResult]")
    @ParameterizedTest
     @MethodSource()

    void integerSubstractionParameters(int number1, int number2, int expectedResult) {

        System.out.println("Running Test" + number1 + " - " + number2  + " = " + expectedResult);
        int result = calculator.integerSubstraction(number1, number2);

        assertEquals(expectedResult, result, () -> number1 + " - " + number2 + " did not produce " +
expectedResult);
    }

    private static Stream<Arguments> integerSubstractionParameters() {
        return Stream.of(
                Arguments.of(33,1,32),
                Arguments.of(3,1,2),
                Arguments.of(10,3,7)
        );
    }
```

@Parameterized & @CsvSource

- Permet de faire la même chose que l'annotation @MethodSource sans avoir à créer de fonction externe

```
    @DisplayName("Test integer substraction [number1, number2, expectedResult]")
    @ParameterizedTest
    @CsvSource({
            "33, 1, 32",
            "24, 1, 23",
            "54, 1, 53"
    })

    void integerSubstractionParameters(int number1, int number2, int
expectedResult) {

        System.out.println("Running Test" + number1 + " - " + number2  + " = " +
expectedResult);
        int result = calculator.integerSubstraction(number1, number2);

        assertEquals(expectedResult, result, () -> number1 + " - " + number2 + "
did not produce " + expectedResult);
    }
```

@Parameterized + CSV File

- Permet d'isoler les jeux de données à tester pour une méthode dans une fichier csv

```
    @DisplayName("Test integer substraction [number1, number2,
expectedResult]")
    @ParameterizedTest
    @CsvFileSource(resources = "/integerSubstraction.csv")
    void integerSubstractionParameters(int number1, int number2, int
expectedResult) {

        System.out.println("Running Test" + number1 + " - " + number2  +
" = " + expectedResult);
        int result = calculator.integerSubstraction(number1, number2);

        assertEquals(expectedResult, result, () -> number1 + " - " +
number2 + " did not produce " + expectedResult);
    }
```

@Parameterized & @ValueSource

- Permet de passer en paramètre d'entrée des strings

```
    @ParameterizedTest
    @ValueSource(strings = {"John", "Kate", "Alice"})
   void valueSourceDemonstration(String firstName) {
        System.out.println(firstName);
        assertNotNull(firstName);
    }
```

Répétition de tests

- Permet d'exécuter plusieurs fois le même test

```
 @DisplayName("Division by Zero")
 @RepeatedTest(value = 3, name="{displayName}; Repetition 
{currentRepetition} of {totalRepetitions}")
 void 
testIntegerDivision_WhenDividendIsDividedByZero_ShouldThrowArithmeticExce
ption(){}
```

Ordre aléatoire d'exécution de test

- Permet d'exécuter les méthodes d'une classe par ordre random

```
@TestMethodOrder(MethodOrderer.Random.class)
public class MethodOrderedRandomnlyTest { }
```

Exécution de test ordonnée par nom

- Permet d'exécuter les méthodes d'une classe par ordre alphabétique de nom

```
@TestMethodOrder(MethodOrderer.MethodName.class)
public class MethodOrderedByNameTest {}
```

Exécution de test ordonnée par index

- Permet d'exécuter les méthodes d'une classe par ordre d'index avec l'annotation @Order

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class MethodOrderedByNameTest {}
```

Ordre d’exécution des classes test

- Combiner la config global dans un fichier junit-platform dans un dossier root resources
- Et L’utilisation des annotations @Order au niveau de nos classes

```
junit.jupiter.testclasss.order.default=org.junit.jupiter.api.ClassOrderer$OrderAnn
otation
```

```
@Order(2)
public class OrderServiceTest {}
```

```
@Order(1)
public class ProductServiceTest {}
```

```
@Order(3)
public class UserServiceTest {}
```

Test Instance Lifecycle

- Pour partager une seule instance de classe à nos méthodes test :
  @TestInstance(TestInstance.Lifecycle.*PER_CLASS*)
- Par défaut, la configuration est la suivante, une nouvelle instance de classe our chaque exécution de méthode test :
  @TestInstance(TestInstance.Lifecycle.*PER_METHOD*)

QUIZ 3

Créer des tests unitaires

4.

TDD - Test Driven Development

5.

Approche du framework Mockito

THE END.

Avec tous nos remerciements et toutes nos félicitations pour avoir suivi ce cursus.

Samih Habbani : s.habbani@coderbase.io