Chapitre 4

Les tests

Enseignante: Dr-Ing. Amina JARRAYA
Email : amina.jarraya@ensi-uma.tn
Niveau: II3- GL

Plan

1. Introduction aux tests
2. Les approches de test – statique et dynamique
3. Les tests statiques avec SonarQube
4. Mise en place de SonarQube
5. Les tests dynamiques – les tests unitaires avec Junit
6. Mise en place de Junit
7. Les tests unitaires dans une application micro-service (springboot, Junit, Mockito, MockMVC, H2 database)

1. Introduction aux tests

Les tests en génie logiciel

- **Les tests** en génie logiciel sont l'analyse systématique d'un logiciel pour détecter des différences entre son comportement réel et ses exigences, dans le but de vérifier sa conformité et d'évaluer sa qualité, sa fiabilité et ses performances.

- Ce processus vise à identifier les erreurs, lacunes ou exigences manquantes pour livrer un produit logiciel sans bugs, qui répond aux attentes des utilisateurs et fonctionne comme prévu.

Pourquoi tester les logiciels ?

- **Assurer la qualité** : Les tests garantissent que le logiciel est fiable et performant.
- **Identifier les défauts** : Ils permettent de détecter les bugs, les erreurs et les manques dès les premières phases du développement.
- **Vérifier les exigences** : Les tests valident que le logiciel fait ce qu'il est censé faire et qu'il répond aux spécifications initiales.
- **Améliorer la fiabilité** : En corrigeant les erreurs découvertes, on augmente la robustesse et la fiabilité du produit.

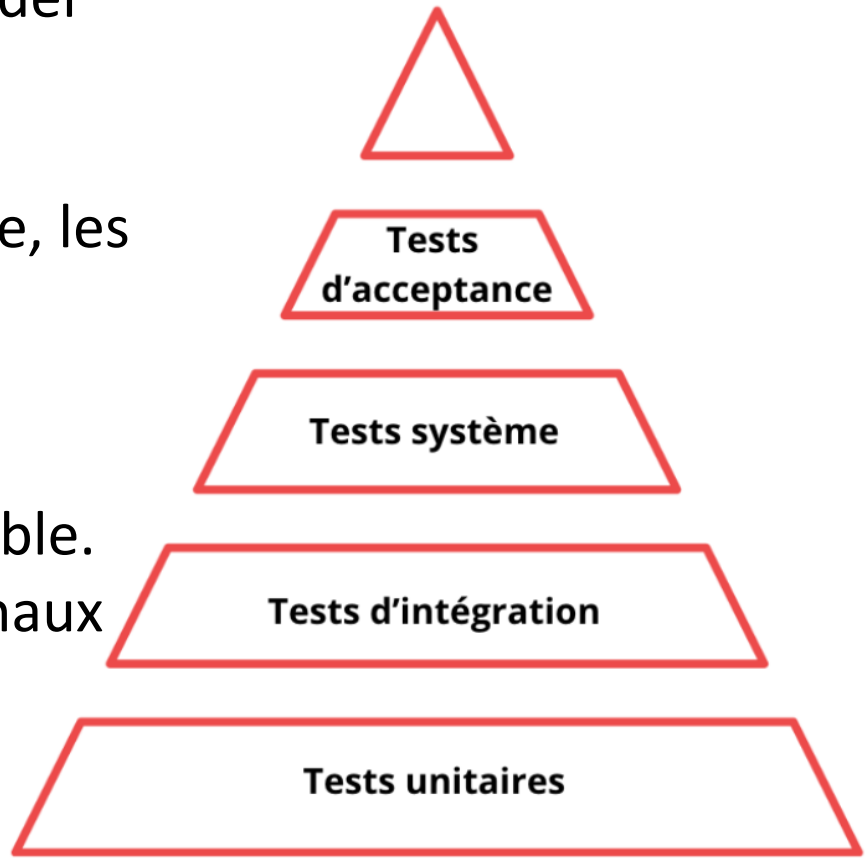
Types courants de tests

- **Tests fonctionnels** : Vérifient que les fonctionnalités du logiciel fonctionnent comme prévu.
- **Tests non fonctionnels** : Évaluent des aspects tels que la performance, la sécurité ou l'utilisabilité.
- **Tests unitaires** : Testent les composants individuels (unités) du code.
- **Tests d'intégration** : Valident l'interaction entre différents modules d'un système.
- **Tests d'utilisabilité** : Évaluent la facilité d'utilisation du produit pour les utilisateurs finaux.

Les phases de test

- Il existe différentes phases de test, comme les tests unitaires, d'intégration, de système et d'acceptation, qui visent à valider des aspects spécifiques du logiciel de manière progressive.

  - **Tests unitaires** :Vérifient les plus petites parties du code, les unités.
  - **Tests d'intégration** :Testent l'intégration et la communication entre plusieurs modules.
  - **Tests de système** :Évaluent le système dans son ensemble.
  - **Tests d'acceptation** :Sont réalisés par les utilisateurs finaux pour valider que le logiciel répond à leurs besoins.



2. Les approches de test statique et dynamique

Les approches de test - statique et dynamique

- En test logiciel, les tests statiques et les tests dynamiques sont deux approches complémentaires pour vérifier la qualité d'un produit.
- Les tests statiques consistent à examiner le code, la documentation, ou les spécifications sans les exécuter.
- Les tests dynamiques impliquent l'exécution du code pour observer son comportement.

Les tests dynamiques

- Les tests dynamiques impliquent l'exécution du code, soit manuellement par des tests unitaires, d'intégration, ou fonctionnels, soit automatiquement par des tests d'acceptation ou des tests de performance.

- Ils permettent de trouver les erreurs qui ne se manifestent que lors de l'exécution du code, comme les erreurs de logique, les problèmes de performance, ou les incompatibilités avec l'environnement.

- **Exemples:**
- Tests unitaires, tests d'intégration, tests fonctionnels, tests de performance, tests d'interface utilisateur.

- **Avantages des tests dynamiques:**
- Détection des erreurs d'exécution, Vérification du comportement réel du code, Possibilité de tester différents scénarios d'utilisation, Tests de performance et de compatibilité.

Les tests statiques

- Les tests statiques peuvent inclure des revues de code, des analyses statiques automatisées (par exemple, avec des outils comme SonarQube ou CodeClimate), et des revues de documentation (expression de besoins, cahier des charges, etc.).

- Ils permettent de détecter les problèmes dès le début du développement, avant l'exécution du code, ce qui peut réduire les coûts de correction.

- **Exemples:**
- Détection de problèmes de syntaxe, de style de code, de vulnérabilités potentielles, de non-conformité aux exigences.

- **Avantages des tests statiques:**
- Détection précoce des erreurs, Réduction des coûts de correction, Possibilité d'automatisation, Analyse de grandes quantités de code.

Les tests statiques

- Les tests statiques peuvent inclure des revues de code, des analyses statiques automatisées (par exemple, avec des outils comme SonarQube ou CodeClimate), et des revues de documentation (expression de besoins, cahier des charges, etc.).

- Ils permettent de détecter les problèmes dès le début du développement, avant l'exécution du code, ce qui peut réduire les coûts de correction.

Les deux types de tests sont complémentaires et devraient être utilisés ensemble pour garantir la meilleure qualité possible du logiciel.

- **E**
- D
  r

- **Avantages des tests statiques:**
- Détection précoce des erreurs, Réduction des coûts de correction, Possibilité d'automatisation, Analyse de grandes quantités de code.

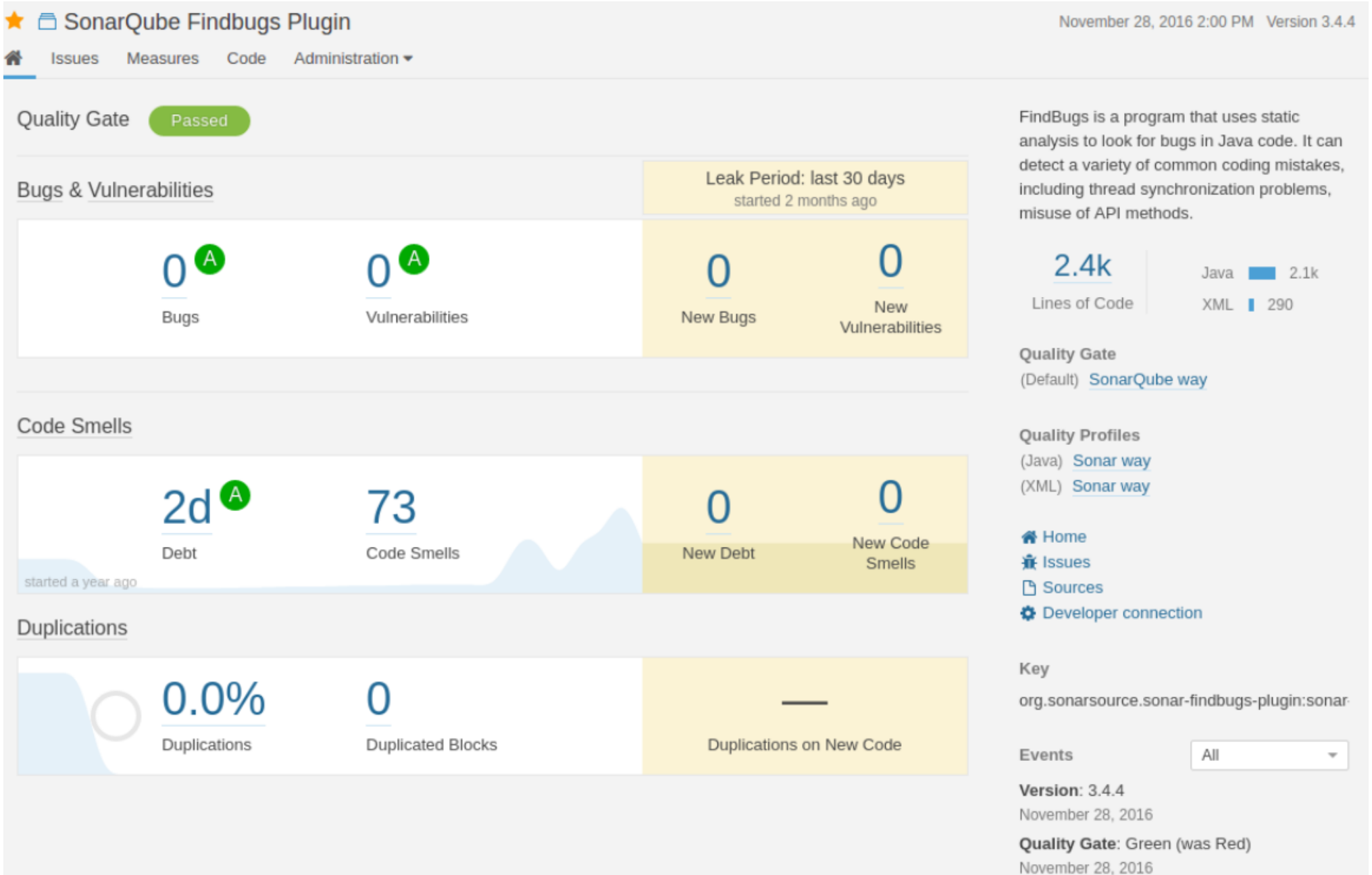
3. Les tests statiques avec SonarQube

Qu’est ce que Sonarqube?

- SonarQube est une plateforme open source, développé par la société SonarSource, dédiée à **l'analyse statique de la qualité et de la sécurité du code source**.

- Elle permet aux développeurs d'identifier et de corriger les problèmes potentiels dans leur code, tels que les bogues, les vulnérabilités de sécurité, les odeurs de code et les duplications. En somme, SonarQube est un outil essentiel pour améliorer la qualité et la fiabilité des logiciels.

Qu’est ce que Sonarqube?



Caractéristiques de Sonarqube

SONARQUBE peut être utilisé avec une vingtaine de langages (Java, .Net (C#), Python, PHP, Cobol, JavaScript, …)

•SonarQube est conçu pour détecter les défauts de codage. Il analyse le code source à la recherche de problèmes tels que les bugs, les vulnérabilités de sécurité et les "code smells" (mauvaises pratiques de codage qui affectent la maintenabilité).

• Il nous permet de choisir les règles à activer lors de l’analyse de notre code.

• SONARQUBE peut être installé en mode standalone, ou en tant que plugin intégré à un IDE comme STS (Eclipse).

Comment ça fonctionne Sonarqube ?

- SonarQube fonctionne en deux parties principales :
  - **un scanner** qui analyse le code
  - **un serveur centralisé** pour stocker les résultats et générer des rapports. Il aide à identifier les problèmes de qualité du code, les bugs, les vulnérabilités et les mauvaises pratiques.

- Voici comment SonarQube fonctionne en détail :

1. Analyse du code source:

- SonarQube utilise un moteur d'analyse (scanner) pour parcourir le code source de votre projet.
- Il recherche des "bad patterns", des erreurs potentielles et des vulnérabilités, sans exécuter le code.
- L'analyse se fait fichier par fichier, en appliquant des règles prédéfinies.
- Exemples de problèmes détectés : injections SQL, code mort, doublons, etc.
- L'analyse est personnalisable et peut s'adapter aux besoins de chaque projet.

Comment ça fonctionne Sonarqube ?

- SonarQube fonctionne en deux parties principales :
  - **un scanner** qui analyse le code
  - **un serveur centralisé** pour stocker les résultats et générer des rapports. Il aide à identifier les problèmes de qualité du code, les bugs, les vulnérabilités et les mauvaises pratiques.

- Voici comment SonarQube fonctionne en détail :

2. Intégration avec les outils de test:

- SonarQube s'intègre avec des outils d'exécution de tests comme Jacoco (Java), Istanbul (JavaScript/TypeScript).
- Il analyse les rapports de ces outils pour mesurer la couverture du code.
- Cela permet d'évaluer l'efficacité des tests et d'identifier les zones du code qui ne sont pas suffisamment testées.

Comment ça fonctionne Sonarqube ?

- SonarQube fonctionne en deux parties principales :
  - **un scanner** qui analyse le code
  - **un serveur centralisé** pour stocker les résultats et générer des rapports. Il aide à identifier les problèmes de qualité du code, les bugs, les vulnérabilités et les mauvaises pratiques.

- Voici comment SonarQube fonctionne en détail :

3. Serveur centralisé:

- Le serveur SonarQube stocke les résultats des analyses et génère des rapports.
- Il offre une interface web pour visualiser les résultats et les métriques.
- Les rapports sont détaillés et permettent de suivre l'évolution de la qualité du code dans le temps.
- Le serveur peut être intégré à des pipelines CI/CD pour une analyse continue du code.

Comment ça fonctionne Sonarqube ?

- SonarQube fonctionne en deux parties principales :
  - **un scanner** qui analyse le code
  - **un serveur centralisé** pour stocker les résultats et générer des rapports. Il aide à identifier les problèmes de qualité du code, les bugs, les vulnérabilités et les mauvaises pratiques.

- Voici comment SonarQube fonctionne en détail :

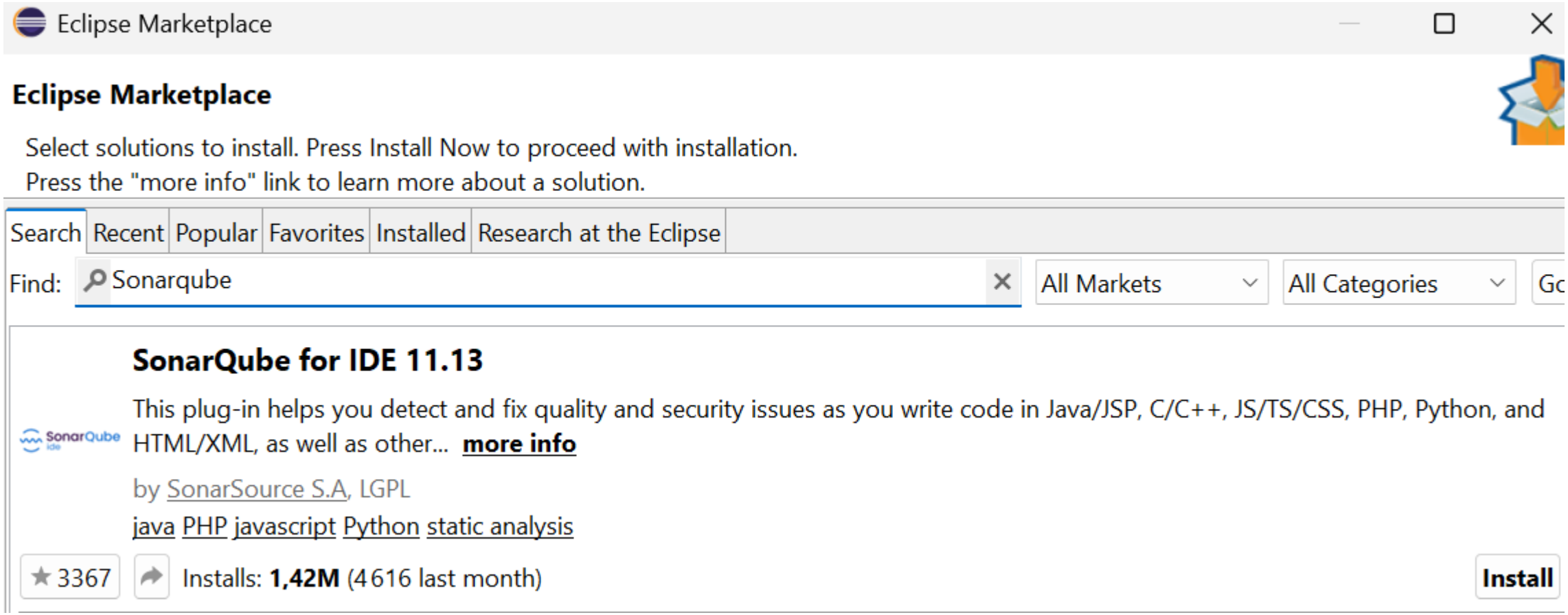
4. Analyse dynamique (optionnelle):

- En plus de l'analyse statique, SonarQube peut être complété par des outils d'analyse dynamique, qui testent le code en fonctionnement.
- Cela permet de détecter des problèmes qui ne sont pas visibles lors de l'analyse statique, comme des erreurs de configuration.

4. Mise en place de SonarQube

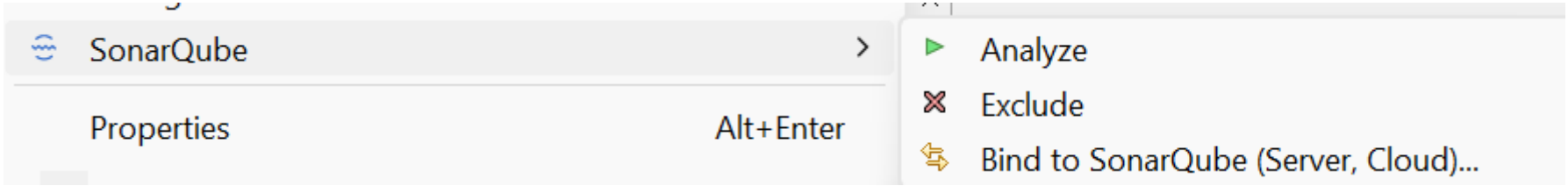
Installer Sonarqube via un IDE

- Dans Eclipse, aller Settings -> Plugins -> MarketPlace, chercher « sonar », installer le plugin « Sonarqube », accepter l'installation, accepter de redémarrer Eclipse à la fin de l'installation.



Lancer SonarQube sous Eclipse

- Ouvrir un de vos projets sur IntelliJ, bouton droit et choisir « SonarLint » -> Analyze, voir le résultat :



Exemple

```
21 @RestController
22 public class CountryController {
23
24     @Autowired
25     CountryService countryservice;
26
27     @GetMapping("/getcountries")
28     public ResponseEntity <List <Country>> getCountries() {
29         try {
30         List <Country> countries = countryservice.getAllCountries();
31         return new ResponseEntity <List <Country>>(countries, HttpStatus.FOUND);
```

Problems | Javadoc | Declaration | **Console** | Terminal | PlantUML | SonarQube Report | SonarQube Rule Description

7 items

Resource	Date	Description
CountryControl	14 minutes ago	Replace the type specification in this constructor call with the diamond operator ("<>").
CountryControl	17 minutes ago	Remove this field injection and use constructor injection instead.
CountryControl	17 minutes ago	Replace the type specification in this constructor call with the diamond operator ("<>").
CountryControl	17 minutes ago	Replace the type specification in this constructor call with the diamond operator ("<>").

Exemple

```java
@RestController
public class CountryController {

    @Autowired
    CountryService countryservice;

    @GetMapping("/getcountries")
    public ResponseEntity <List <Country>> getCountries() {
        try {
        List <Country> countries = countryservice.getAllCountries();
        return new ResponseEntity <List <Country>>(countries, HttpStatus.FOUND);
```

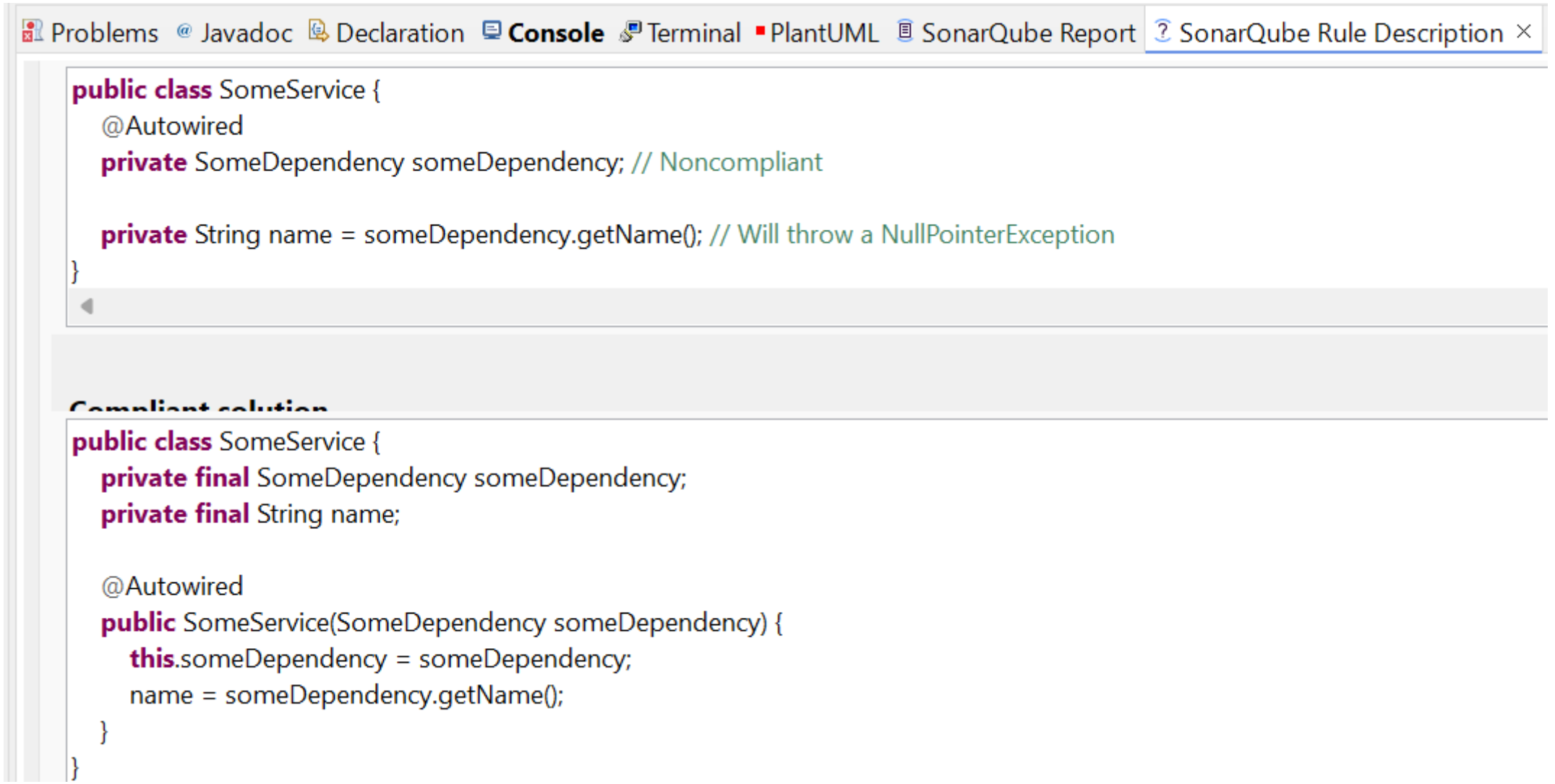
Problems | Javadoc | Declaration | **Console** | Terminal | PlantUML | SonarQube Report | SonarQube Rule Description

Consistency | Not conventional Maintainability Reliability

Why is this an issue? | How can I fix it? | More Info

Dependency injection frameworks such as Spring, Quarkus, and others support dependency injection by using annotations such as `@Inject` and `@Autowired`. These annotations can be used to inject beans via constructor, setter, and field injection.

Exemple


Problems
Javadoc
Declaration
Console
Terminal
PlantUML
SonarQube Report
SonarQube Rule Description
public class SomeService {
@Autowired
private SomeDependency someDependency; // Noncompliant
private String name = someDependency.getName(); // Will throw a NullPointerException
}
public class SomeService {
private final SomeDependency someDependency;
private final String name;
@Autowired
public SomeService(SomeDependency someDependency) {
this.someDependency = someDependency;
name = someDependency.getName();
}
}

Les couleurs des messages affichés

- En SonarQube, les couleurs associées aux messages (erreurs, avertissements, etc.) suivent une logique visuelle pour communiquer rapidement l'état du code analysé. Voici la signification de chaque couleur :

- **Bleu**: Indique des informations ou des détails supplémentaires.
- **Jaune**/**Orange**: Signale des avertissements ou des problèmes potentiels qui ne sont pas critiques, mais qui nécessitent une attention.
- **Rouge**: Représente des erreurs ou des problèmes critiques qui doivent être corrigés immédiatement.
- **Vert**: Indique une réussite ou une validation, signifiant que la partie du code analysée est conforme aux règles.

- Ces couleurs aident les développeurs à prioriser les actions à mener et à identifier rapidement les zones du code qui nécessitent une intervention.

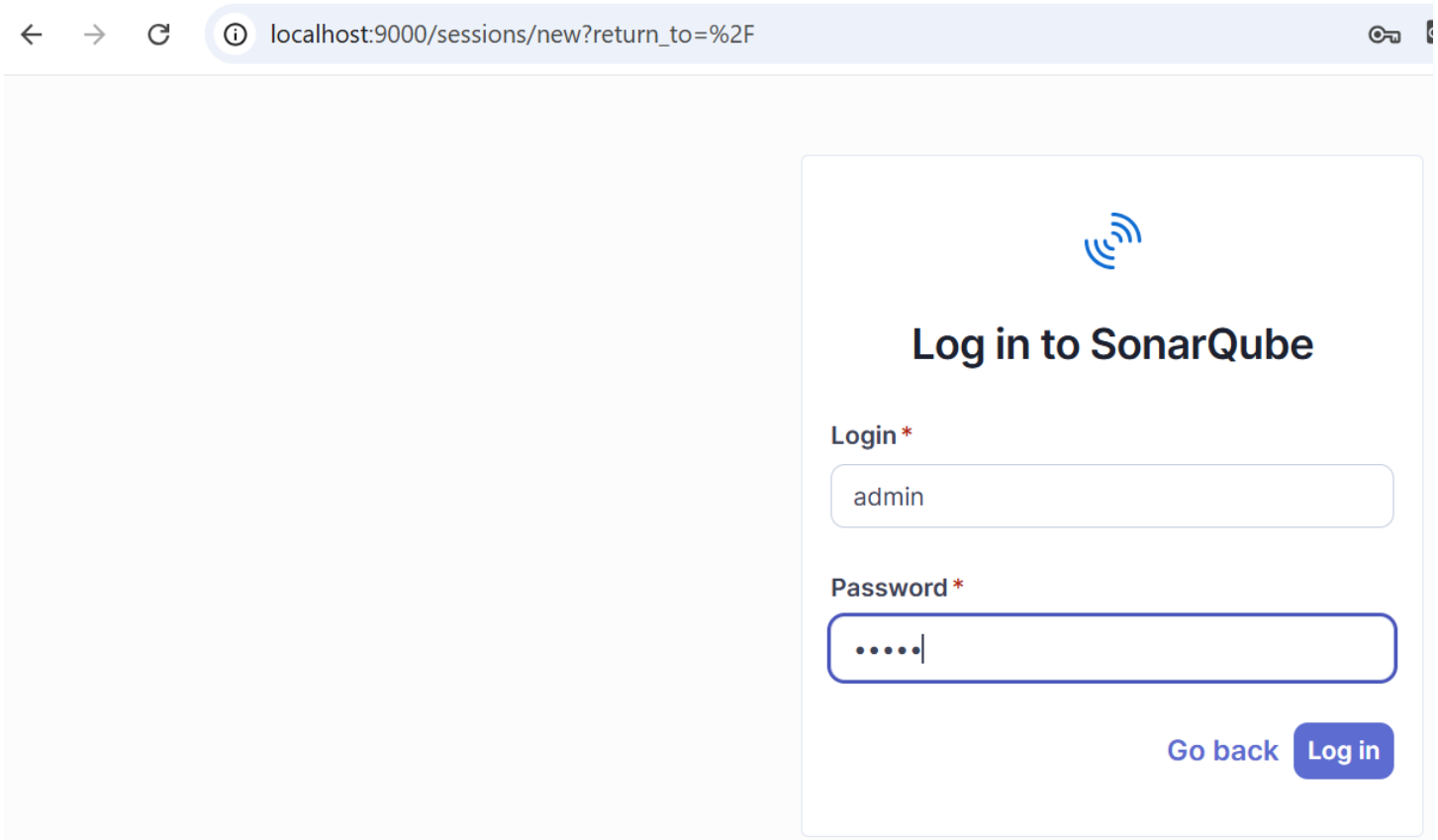
Installer SonarQube via Docker

```
root@DESKTOP-SKOOCDK:/# docker pull sonarqube
Using default tag: latest
latest: Pulling from library/sonarqube
e4b4015f66e4: Pull complete
f62b74addfec: Pull complete
8cce32af8d45: Pull complete
b71466b94f26: Pull complete
4f4fb700ef54: Pull complete
95c1ceb63b80: Pull complete
f8ac95b0cea9: Pull complete
5a04f5b1da55: Pull complete
Digest: sha256:6d64b6cdec730c81a8ccbf266716f342d5c9d8a8c93426a2377e239d3ac79a38
Status: Downloaded newer image for sonarqube:latest
docker.io/library/sonarqube:latest
root@DESKTOP-SKOOCDK:/#
```

Installer SonarQube Server via Docker

- Lancer Sonarqube Server : docker run -p 9000:9000 sonarqube
- (-p pour exposer le port 9000 et pour que sonarqube soit accessible de l'extérieur)

- Sur votre machine Windows, aller à l'url http://localhost:9000 et se connecter avec admin/admin :



Intégrer Sonarqube pour les projets Maven

- Il faut utiliser une version de Maven inférieur à 4.

```
C:\Users\HP-EliteBook>mvn -version
Apache Maven 3.9.11 (3e54c93a704957b63ee3494413a2b544fd3d825b)
Maven home: C:\Users\HP-EliteBook\apache-maven-3.9.11
Java version: 21.0.7, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-21
Default locale: fr_FR, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

- Rajouter la ligne suivante dans le fichier **settings.xml** sous C:\Users\HP-EliteBook\apache-maven-3.9.11\conf

<pluginGroup>org.sonarsource.scanner.maven</pluginGroup>

```
<pluginGroups>
  <!-- pluginGroup
   | Specifies a further group identifier to use for plugin lookup.
  <pluginGroup>com.your.plugins</pluginGroup>
  -->
  <pluginGroup>org.sonarsource.scanner.maven</pluginGroup>
</pluginGroups>
```

Intégrer Sonarqube pour les projets Maven

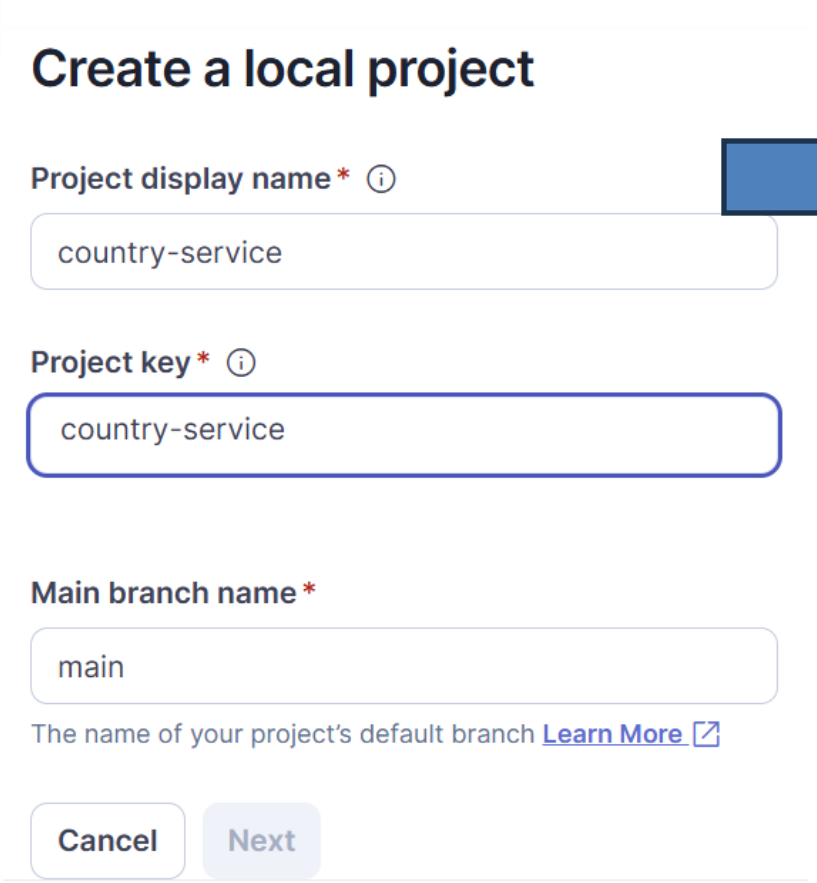
- Rajouter le bloc suivant dans le fichier **settings.xml** sous C:\Users\HP-EliteBook\apache-maven-3.9.11\conf

```
<profile>
<id>sonar</id>
 <activation>
<activeByDefault>true</activeByDefault>
 </activation>
<properties>
<!-- Optional URL to server. Default value is http://localhost:9000 -->
<sonar.host.url>
http://localhost:9000
</sonar.host.url>
</properties>
</profile>
```

```
    <profile>
      <id>env-dev</id>

      <activation>
        <property>
          <name>target-env</name>
          <value>dev</value>
        </property>
      </activation>

      <properties>
        <tomcatPath>/path/to/tomcat/instance</tomcatPath>
      </properties>
    </profile>
    -->

    <profile>
       <id>sonar</id>
       <activation>
          <activeByDefault>true</activeByDefault>
       </activation>
            <properties>
                <!-- Optional URL to server. Default value is http://localhost:9000 -->
                <sonar.host.url>
                  http://localhost:9000
                </sonar.host.url>
            </properties>
    </profile>
  </profiles>
```

Intégrer Sonarqube pour les projets Maven

- Créer un nouveau projet local dans le serveur de sonarqube sur l'addresse : http://localhost:9000/ puis générer votre token




```
mvn clean verify sonar:sonar
-Dsonar.projectKey=country-service
-Dsonar.projectName='country-service'
-Dsonar.host.url=http://localhost:9000
-Dsonar.token=sqp_9898a696e013340b624c422f6a36c4c332a22b32
```

Intégrer Sonarqube pour les projets Maven

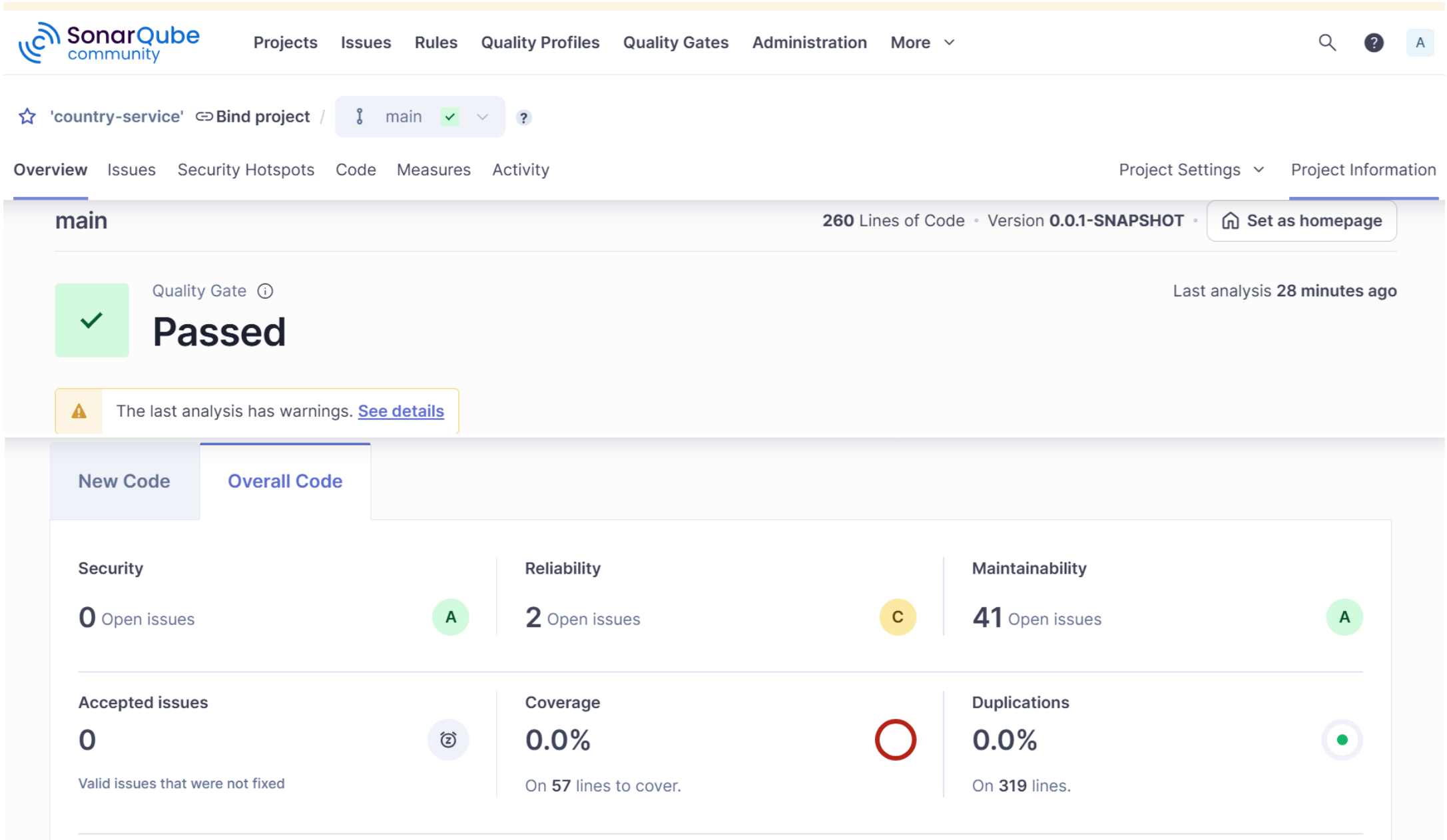
- Taper la commande générée dans le terminal de votre projet :

```
mvn clean verify sonar:sonar
-Dsonar.projectKey=country-service
-Dsonar.projectName='country-service’
-Dsonar.host.url=http://localhost:9000
-Dsonar.token=sqp_9898a696e013340b624c422f6a36c4c332a22b32
```

```
[INFO] ANALYSIS SUCCESSFUL, you can find the results at: http://localhost:9000/dashboard?id=country-service
[INFO] Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
[INFO] More about the report processing at http://localhost:9000/api/ce/task?id=7b1a9c72-134e-4d43-bb62-36e126e5587f
[INFO] Analysis total time: 42.943 s
[INFO] SonarScanner Engine completed successfully
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:59 min
[INFO] Finished at: 2025-08-19T12:13:32+02:00
[INFO] ------------------------------------------------------------------------
```

- Accéder à l’adresse sélectionné

Analyse du projet dans le serveur du SonarQube



5. Les tests dynamiques – les tests unitaires avec JUnit

Qu’est ce que les tests unitaires ?

- **Test unitaire**
  - Procédure permettant de vérifier le bon fonctionnement
    . d'une partie précise d'un logiciel
    . d'une portion d'un programme (appelée « unité » ou « module »)

- **Objectifs**
  - S’assurer qu’une unité fonctionnelle ne comporte pas d’erreurs
  - Vérifier que les classes collaborent bien
  - Garantir l’identification des erreurs au fur et à mesure des modifications du code

Principe des tests unitaires

- **Test unitaire repose sur deux principes simples**

  1. **SI** ça fonctionne une fois, **ALORS** ça fonctionnera les autres fois
  2. **SI** ça fonctionne pour quelques valeurs bien identifiées, **ALORS** ça fonctionnera pour toutes les autres

- **Exemple**
- Soit la méthode *String concatene(String t1, String t2)* qui concatène les chaines de caractères t1 et t2
  - SANS TESTS UNITAIRES

    Afficher le résultat de la méthode dans le programme lors de l'appel de la fonction
  - AVEC LES TESTS UNITAIRES

    Créer une classe dédiée qui se chargera de faire un ensemble de tests avec différentes valeurs

Doit-on tester tous les cas possibles ?

Principe

Schéma de test de classique:

- Un état de départ
- Un état attendu
- Un état d'arrivée
- Objectif : calcule l'état d'arrivé, et vérifie qu'il correspond à l'état attendu

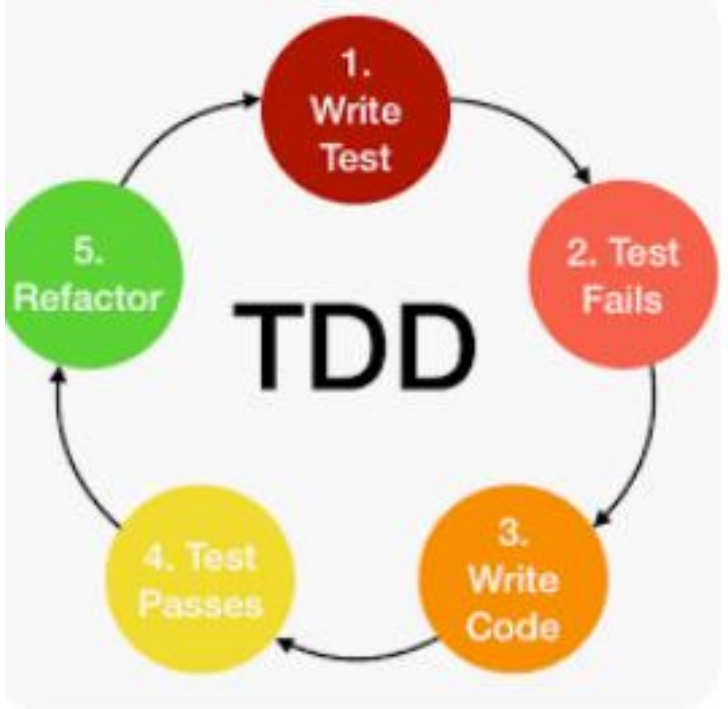
```
public boolean testConcatene(){
        String t1 = "Bonjour " ;
        String t2 = "Toto";
        String attendu = "Bonjour Toto";
        String arrivee = concatene(t1, t2);
        Return arrivee.equals(attendu);
}
```

Pour coder ce test, il n'est pas nécessaire de savoir comment est codé la méthode *concatene*

Principe

- **Test Driven Development (TDD)**
  - Proposé par Kent Beck, 2002
  - Ecrire les tests avant de coder l'application (Méthodes Agiles XP, Scrum, etc.)
- Une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.

- **Avantages du TDD**
  - Ecrire les tests en premier (Manipuler le programme avant même son existence)
  - Permet de segmenter les taches
  - Garanti que l'ensemble du programme est testé
  - Facilite le travail en équipe et la collaboration entre les modules
  - Augmente la confiance du programmeur lors de la modification du code
  - Evite la régression



TEST UNITAIRE : OUTIL DE TEST

PHP	JS	SQL	JAVA
PHPUnit SimpleTest	JSUnit	SQLUnit	JUnit

5.1. Les Tests unitaires avec JUnit

Présentation de JUnit

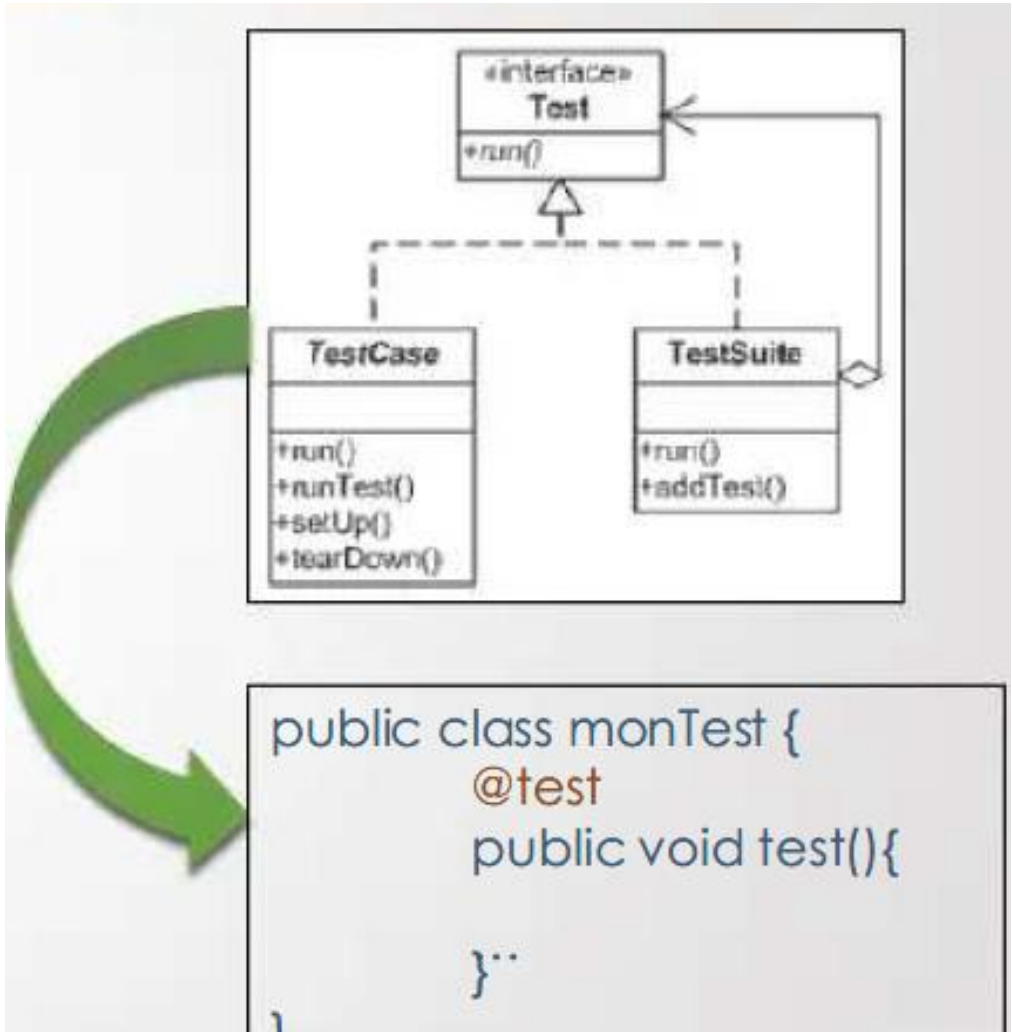
- Framework permettant la mise en place de Test Unitaires en JAVA
  - Open Source: www.junit.org
  - Créé par Kent Beck (eXtreme Programming) et Erich Gamma (Design patterns)
  - Intégré par défaut dans la plupart des IDE (i.e, Eclipse, NetBeans, BlueJ, etc.)

En 2013, une étude menée sur les projets hébergés sur GitHub montre que 31% des projets utilisent JUnit

Versions de JUnit

- **Version 3.8**
  - Basé sur un ensemble de Classes et d'interfaces à implémenter
  - Mise en place des tests en implémentant des classes respectant le Framework

- **Version 4 et 5**
  - Plus souple
  - Basé sur des annotations (Java 5+)
  - Permettent de marquer les méthodes dédiées au test



La classe dans JUnit

- A chaque classe, on associe une classe de test
- Dans Junit3, une classe de test hérite de la classe junit.framework.TestCase pour hériter des méthodes de tests
- Depuis la version 4
  - Inutile de préciser la relation d'héritage
  - les méthodes de tests sont identifiés par des **annotations** Java

- Classe de Test
  - En général

<nomDeLaClasseTestée>TEST.java (Ex. CounteurTest.java)

- Importer packages :

Import org.junit.Test;

Import static org.junit.Assert.*;

Version 3

```
import junit.framework.TestCase;

public class MyTestClass extends TestCase {
    ,
}
```

Version 4 et 5

```
//Classe qui permet de manipuler un compteur
public class Compteur{
        int valeur;
        Compteur(){...}
        Compteur(int init){...}
        int increment(){...}
        int decrement(){...}
        int getValeur(){...}
}
```

```
import org.junit.Test;
import static org.junit.Assert.*;

//Classe dédiée au test de la classe Compteur
public class CompteurTest {

}
```

Les méthodes dans JUnit

- Méthodes de test
  - Le nom Commence par **test**, suivi du nom de la méthode testée

  Ex. **testIncrement**()
  - Visibilité **public**
  - Type de retour: **void**
  - Pas de paramètres
  - Peut lever une exception
  - Annotée **@Test**
  - Utilise **des instructions de test**

```
import org.junit.Test;
import static org.junit.Assert.*;


//Classe dédiée au test de la classe Compteur
public class CompteurTest {
        @Test
        public void testIncrement(){
        //La méthode increment() est testée ici

        }
        @Test
        public void testDecrement(){
        //La méthode decrement() est testée ici

        }
}
```

Exemple introductif de cas de test

- Le programme suivant fournit des méthodes statiques pour faire des calculs arithmétiques d'addition et de division.

```
1    public final class Calculator {
2      public static int add( int operand1, int operand2 ) {
3        return operand1 + operand2;
4      }
5      public static int div( int operand1 , int operand2 ) {
6        return operand1 / operand2;
7      }
8    }
```

- La classe de test suivante code un exemple de cas de test pour les deux opérations add et div sous la forme de deux méthodes de test unitaire.

```
1    public final class CalculatorTest {
2      @Test
3      public void testAdd() {
4        assertEquals(3, Calculator.add(1, 2));
5      }
6      @Test
7      public void testDiv() {
8        assertEquals(1, Calculator.div(3, 2));
9      }
10   }
```

Les éléments de base de Junit – les annotations

Version 4

Table 1 : Liste des annotations JUnit.

Annotation	Description
@Test **public void** should_do_when_case()	Marque une méthode comme contenant du code de test.
@Before **public void** setup()	Marque une méthode pour qu'elle s'exécute avant chaque méthode de test de la classe.
@After **public void** tearDown()	Marque une méthode pour qu'elle s'exécute après chaque méthode de test de la classe.
@BeforeClass **public static void** setupBeforeClass()	Marque une méthode statique pour qu'elle s'exécute avant la première méthode de test de la classe. Elle ne s'exécute qu'une seule fois par classe.
@AfterClass **public static void** tearDownAfterClass()	Marque une méthode statique pour qu'elle s'exécute après la dernière méthode de test de la classe. Elle ne s'exécute qu'une seule fois par classe.
@Test(expected=*Exception.class*) **public void** should_do_when_case()	Marque une méthode de test pour vérifier que la méthode testée lève bien l'exception attendue.
@Test(timeout=*100*) **public void** should_do_when_case()	Marque une méthode de test pour vérifier que la méthode testée s'exécute dans le temps imparti.

Les éléments de base de Junit – les assertions

- Une assertion automatise le verdict d'un test unitaire pour décider si la méthode testée passe avec succès ou échoue le test.

Table 2 : Liste des assertions de JUnit.

Fonction	Description
assertTrue(*condition*)	Vérifie que la condition en paramètre est vraie.
assertFalse(*condition*)	Vérifie que la condition en paramètre est fausse.
assertEquals(*expected*, *actual*)	Teste si les valeurs sont égales au sens de la méthode `equals()`. Note : pour les tableaux, seule la référence est vérifiée pas le contenu.
assertArrayEquals(*expected*, *actual*)	Teste si les valeurs des éléments sont les mêmes dans les deux tableaux.
assertEquals(*expected*, *actual*, *tolerance*)	Vérifie l'égalité entre doubles ou flottants à une valeur de précision près.
assertNull(*object*)	Vérifie que l'objet en paramètre est bien « null ».
assertNotNull(*object*)	Vérifie que l'objet en paramètre n'est pas « null ».
assertSame(*expected*, *actual*)	Vérifie que les deux variables pointent vers le même objet en utilisant l'opérateur ==.
assertNotSame(*expected*, *actual*)	Vérifie que les deux variables ne pointent pas vers le même objet.

Les éléments de base de Junit – les assertions

- La Table 2 récapitule les principales assertions JUnit. Ce sont des méthodes statiques de la classe Assert. Chaque méthode existe en deux versions, avec ou sans un premier paramètre contenant le message qui sera affiché en cas d'erreurs. Par exemple, la première assertion de la Table 2 existe sous la forme assertTrue(condition) et assertTrue(message, condition).

Table 2 : Liste des assertions de JUnit.

Fonction	Description
assertTrue(*condition*)	Vérifie que la condition en paramètre est vraie.
assertFalse(*condition*)	Vérifie que la condition en paramètre est fausse.
assertEquals(*expected*, *actual*)	Teste si les valeurs sont égales au sens de la méthode `equals()`. Note : pour les tableaux, seule la référence est vérifiée pas le contenu.
assertArrayEquals(*expected*, *actual*)	Teste si les valeurs des éléments sont les mêmes dans les deux tableaux.
assertEquals(*expected*, *actual*, *tolerance*)	Vérifie l'égalité entre doubles ou flottants à une valeur de précision près.
assertNull(*object*)	Vérifie que l'objet en paramètre est bien « null ».
assertNotNull(*object*)	Vérifie que l'objet en paramètre n'est pas « null ».
assertSame(*expected*, *actual*)	Vérifie que les deux variables pointent vers le même objet en utilisant l'opérateur ==.
assertNotSame(*expected*, *actual*)	Vérifie que les deux variables ne pointent pas vers le même objet.

Exemple

Principales instructions de tests

```
import org.junit.Test;
import static org.junit.Assert.*;

//Classe dédiée au test de la classe Compteur
public class CompteurTest {
        @Test
        public void testIncrement(){
                int attendue;

                attendue = 1;
                Compteur c1 = new Compteur();
                c1.incrementer();
                assertTrue(c1.getValeur() == attendue);

                attendue = 11;
                Compteur c2 = new Compteur(10);
                c2.incrementer();
                assertEquals("Echec test 2",
c2.getValeur(), attendue);
        }
}
```

Valeur attendue

Création des objets pour le test

Appel de la méthode à tester

Vérification que le résultat correspond bien au résultat attendu.

Si ce n'est pas le cas:

1. Lance une *AssertionFailedError* et la méthode de test s'arrête
2. L'exécuteur de Test JUnit attrape cet objet et indique que la méthode a échoué
3. La méthode de test suivante est exécutée

Exemple

La méthode *fail()*

```
import org.junit.Test;
import static org.junit.Assert.*;

//Classe dédiée au test de la classe Compteur
```

fail() est l'instruction la plus importante !
Les autres ne sont que des raccourcis d'écriture

```
                    attendue = 11;
                    Compteur c2 = new Compteur(10);
                    c2.incrementer();
                    assertEquals("Echec test 2",
c2.getValeur(), attendue);
                }
}
```

Exemple

La méthode *fail()*

```
import org.junit.Test;
import static org.junit.Assert.*;

//Classe dédiée au test de la classe Compteur
public class CompteurTest {
        @Test
        public void testIncrement(){
                int attendue;

                attendue = 1;
                Compteur c1 = new Compteur();
                c1.incrementer();
                assertTrue(c1.getValeur() == attendue);

                attendue = 11;
                Compteur c2 = new Compteur(10);
                c2.incrementer();
                assertEquals("Echec test 2",
c2.getValeur(), attendue);
        }
}
```

```
attendue = 1;
Compteur c1 = new Compteur();
c1.incrementer();
if(c1.getValeur() != attendue){
        fail("ECHEC DU TEST");
}
```

Exemple

Plusieurs méthodes de tests

```
import org.junit.Test;
import static org.junit.Assert.*;
public class CompteurTest {
        @Test
        public void testIncrement(){
                Compteur c = new Compteur(10);
                c.incrementer();
                int attendu = 11;
                assertTrue(c.getValeur() == attendu);

        }
        @Test
        public void testDecrement(){
                Compteur c = new Compteur(10);
                c.decrementer();
                int attendu = 9;
                assertTrue(c.getValeur() == attendu);

        }
}
```

Exemple

Plusieurs méthodes de tests

```
import org.junit.Test;
import static org.junit.Assert.*;
public class CompteurTest {
        @Test
        public void testIncrement(){
                Compteur c = new Compteur(10);
                c.incrementer();
                int attendu = 11;
                assertTrue(c.getValeur() == attendu);

        }
        @Test
        public void testDecrement(){
                Compteur c = new Compteur(10);
                c.decrementer();
                int attendu = 9;
                assertTrue(c.getValeur() == attendu);

        }
}
```

Partie commune à chaque test !

Exemple

Plusieurs méthodes de tests

```
import org.junit.Test;
import static org.junit.Assert.*;
public class CompteurTest {
        @Test
        public void testIncrement(){
                Compteur c = new Compteur(10);
                c.incrementer();
                int attendu = 11;
                assertTrue(c.getValeur() == attendu);

        }
        @Test
        public void testDecrement(){
                Compteur c = new Compteur(10);
                c.decrementer();
                int attendu = 9;
                assertTrue(c.getValeur() == attendu);

        }
}
```

```
//Attributs
private Compteur c;
```

```
//Initialiser le compteur avant chaque test
@Before
public void setUp(){
        super.setUp();
        c = new Compteur(10);
}
```

Exemple

- D’autres annotations peuvent être utilisées

Annotation	Description
@Test	Méthode de test
@Before	Méthode exécutée avant chaque test *(méthode **setUp()** avec Junit 3.8)*
@After	Méthode exécutée après chaque test *(méthode **tearDown()** avec Junit 3.8)*
@BeforeClass	Méthode exécutée avant le premier test doit être ***static*** *(méthode **setUpBeforeClass()** avec Junit 3.8)*
@AfterClass	Méthode exécutée après le dernier test doit être ***static*** *(méthode **tearDownAfterClass()** avec Junit 3.8)*

Ordre d'exécution des annotations

Ordre d'exécution

1. La méthode annotée **@BeforeClass**
2. Pour chaque méthode annotée **@Test** (ordre indéterminé)
    1. Les méthodes annotées **@Before**
    2. La méthode annotée **@Test**
    3. Les méthodes annotées **@After** (ordre indéterminé)
3. La méthode annotée **@AfterClass**

```
public class CompteurTest {
        private Compteur c;

        @Before
        public void setUp(){
                super.setUp();
                c = new Compteur(10);
        }

        @Test
        public void testIncrement(){
                c.incrementer();
                int attendu = 11;
                assertTrue(c.getValeur() == attendu);
        }
        @Test
        public void testDecrement(){
                c.decrementer();
                int attendu = 9;
                assertTrue(c.getValeur() == attendu);
        }
}
```

Exemple

```
1   public final class JUnitTest {
2     private List<String> _list;
3
4     @BeforeClass
5     public static void oneTimeSetUp() {
6       System.out.println("@BeforeClass - oneTimeSetUp");
7     }
8
9     @AfterClass
10    public static void oneTimeTearDown() {
11      System.out.println("@AfterClass - oneTimeTearDown");
12    }
13
14    @Before
15    public void setUp() {
16      _list = new ArrayList<>();
17      System.out.println("@Before - setUp");
18    }
19
20    @After
21    public void tearDown() {
22      _list.clear();
23      System.out.println("@After - tearDown");
24    }
25
26    @Test
27    public void testEmptyList() {
28      assertTrue(_list.isEmpty());
29      System.out.println("@Test - testEmptyList");
30    }
31
32    @Test
33    public void testOneItemList() {
34      _list.add("itemA");
35      assertEquals(1, _list.size());
36      System.out.println("@Test - testOneItemList");
37    }
38  }
```

- Le code présente les annotations et les assertions de JUnit à travers un cas de test de la classe List de Java.

Exécution

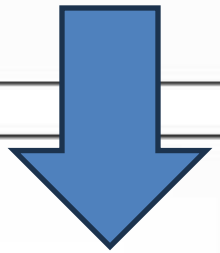
```
@BeforeClass - oneTimeSetUp
@Before - setUp
@Test - testEmptyList
@After - tearDown
@Before - setUp
@Test - testOneItemList
@After - tearDown
@AfterClass - oneTimeTearDown
```

Dans la mesure du possible, il faut limiter les méthodes de test à une seule assertion de manière à avoir une bonne identification de l'erreur.

Les exceptions

- Comment vérifier qu'une méthode lève bien une exception?

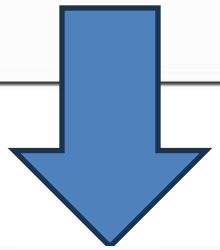
```
public class CompteurTest {

        private Compteur c;
        //La méthode décrémenter doit lever une CompteurZeroException si le compteur est à zéro

        @Test
        public void testDecrementAPartirdeZero(){
                c = new Compteur(0);
                c.decrementer();
        }
}
```



```
public class CompteurTest {

        private Compteur c;
        //La méthode décrémenter doit lever une CompteurZeroException si le compteur est à zéro

        @Test
        public void testDecrementAPartirdeZero(){
                c = new Compteur(0);
                try{
                        c.decrementer();
                        fail("ECHEC décrémentation d'un compteur à zero !");
                }
                catch(CompteurZeroException  e){
                        //OK L'exception est correctement lancée
                }
        }
}
```

Les exceptions

```
public class CompteurTest {

        private Compteur c;
        //La méthode décrémenter doit lever une CompteurZeroException si le compteur est à zéro

        @Test
        public void testDecrementAPartirdeZero(){
                c = new Compteur(0);
                c.decrementer();
        }
}
```



Une autre solution: utilisation des annotations

```
public class CompteurTest {

        private Compteur c;
        //La méthode décrémenter doit lever une CompteurZeroException si le compteur est à zéro

        @Test (expected=CompteurZeroException .class)
        public void testDecrementAPartirdeZero(){
                c = new Compteur(0);
                c.decrementer();
        }
}
```

Tester la durée d'une méthode

- Le paramètre timeout de l'annotation @Test permet de tester la durée maximale de l'exécution d'une méthode en millisecondes. Par exemple, la méthode à tester est :

```
1   public final class JUnit2 {
2     public static void infiniteLoop() {
3       while (true);
4     }
5   }
```

Une méthode de test

```
1   public final class JUnit2Test {
2     @Test(timeout = 1000)
3     public void should_throw_timeout_when_infinite_loop() {
4       JUnit2.infiniteLoop();
5     }
6   }
```

- Dans le cas précédent, la méthode **infinityLoop()** ne finira jamais, donc le moteur JUnit marquera le test en échec après 1000 millisecondes et lèvera l'exception suivante :

```
java.lang.Exception: test timed out after 1000 milliseconds
```

Tester la durée d'une méthode – autre exemple

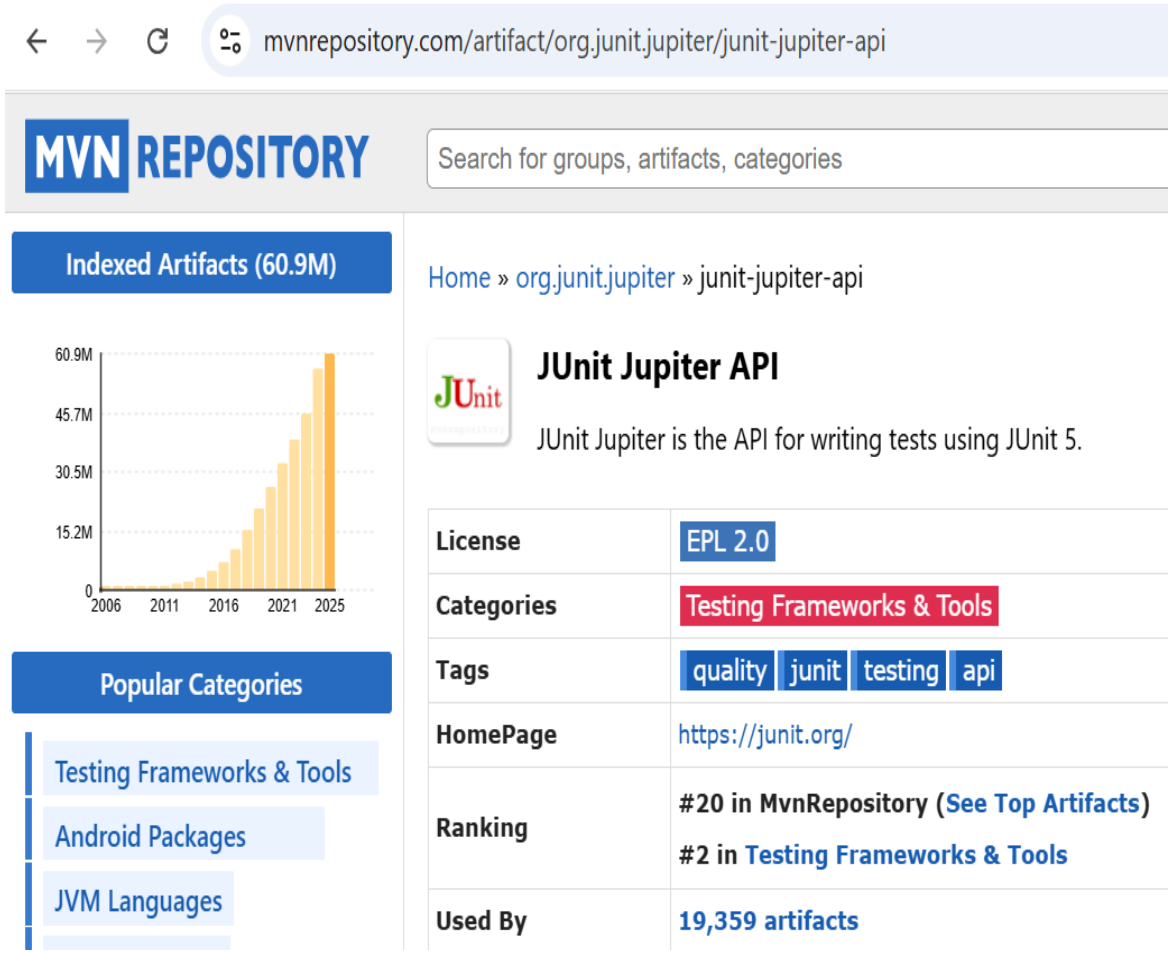
```
public class CompteurTest {

        private Compteur c;
        //La méthode doit échouer si le temps d'exécution dépasse un seuil

        @Test (timeout=10000)
        public void testIncrementMultiple(){
                c = new Compteur(0);
                c.incrementer(1000000);
        }
}
```

6. Mise en place de JUnit

Exemple

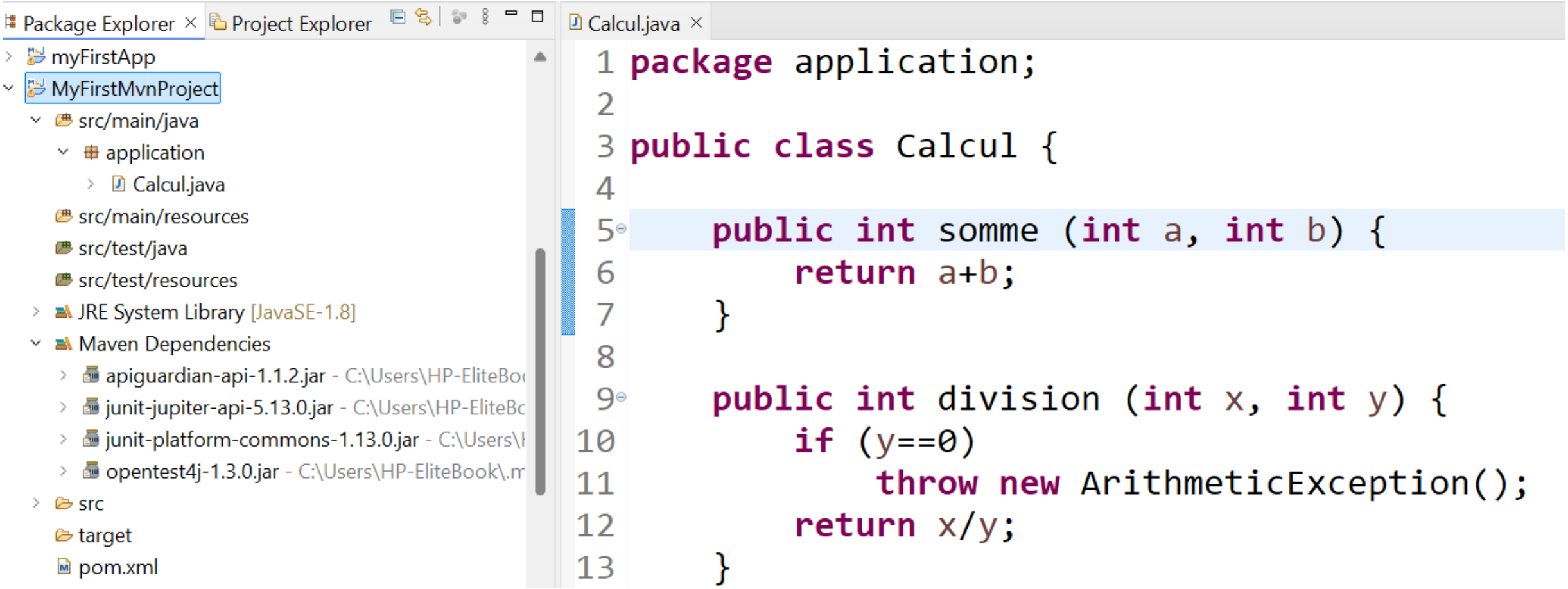
- Etape 1 : Créer un projet **Maven** et rajouter la dépendance **Junit 5**

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.13.0</version>
    <scope>test</scope>
</dependency>
```



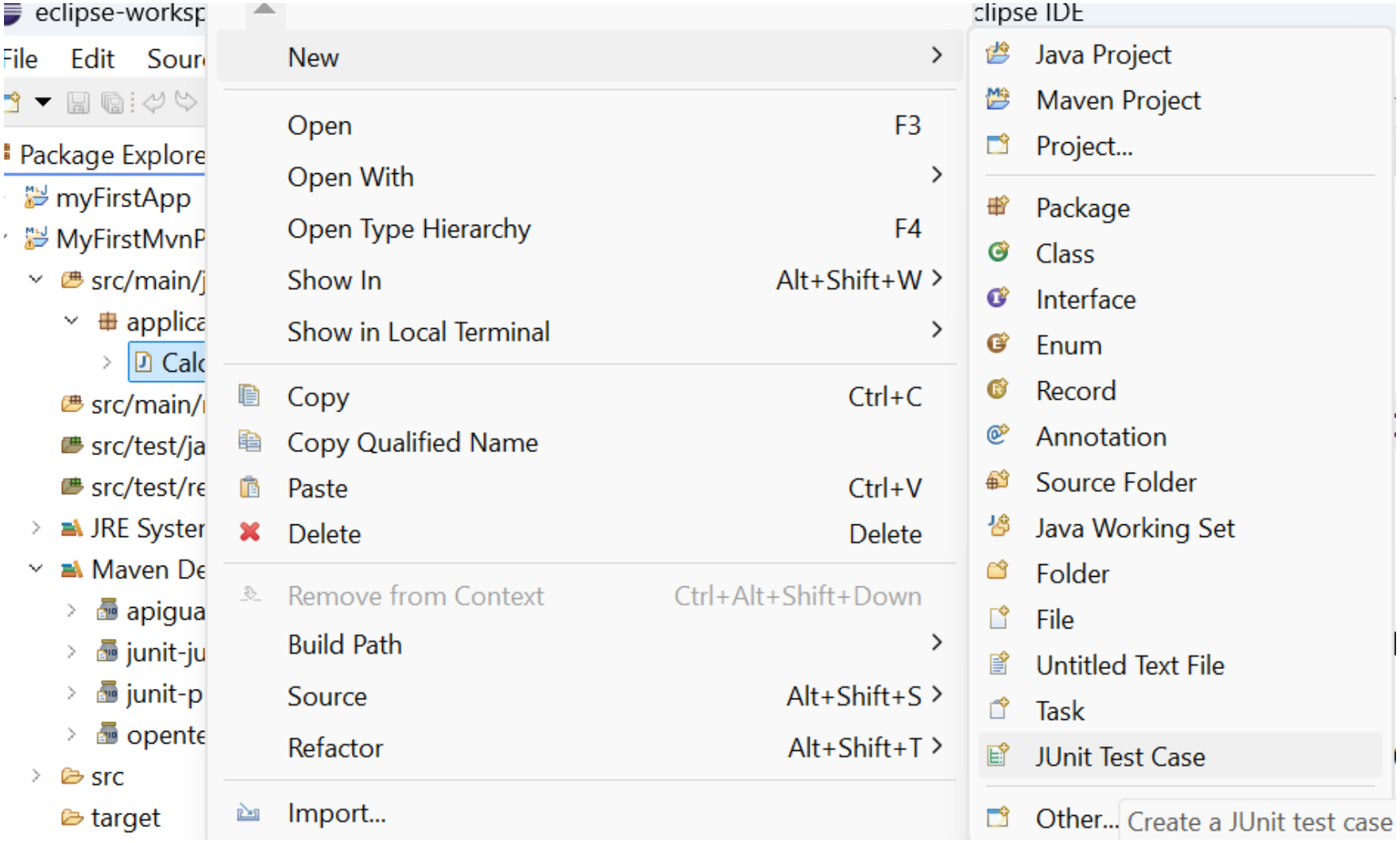
Exemple

- Etape 2 : Créer la classe Calcul

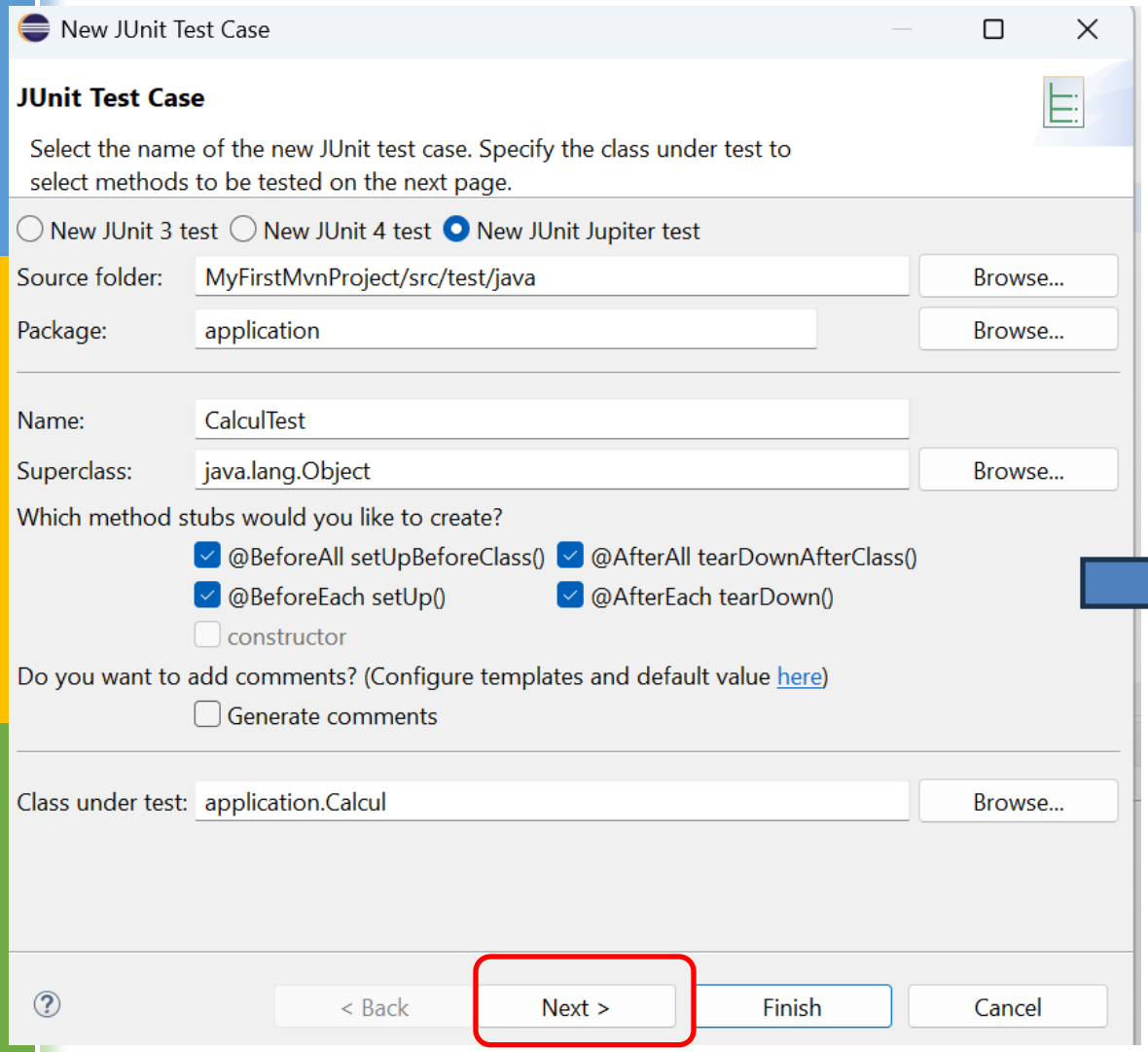


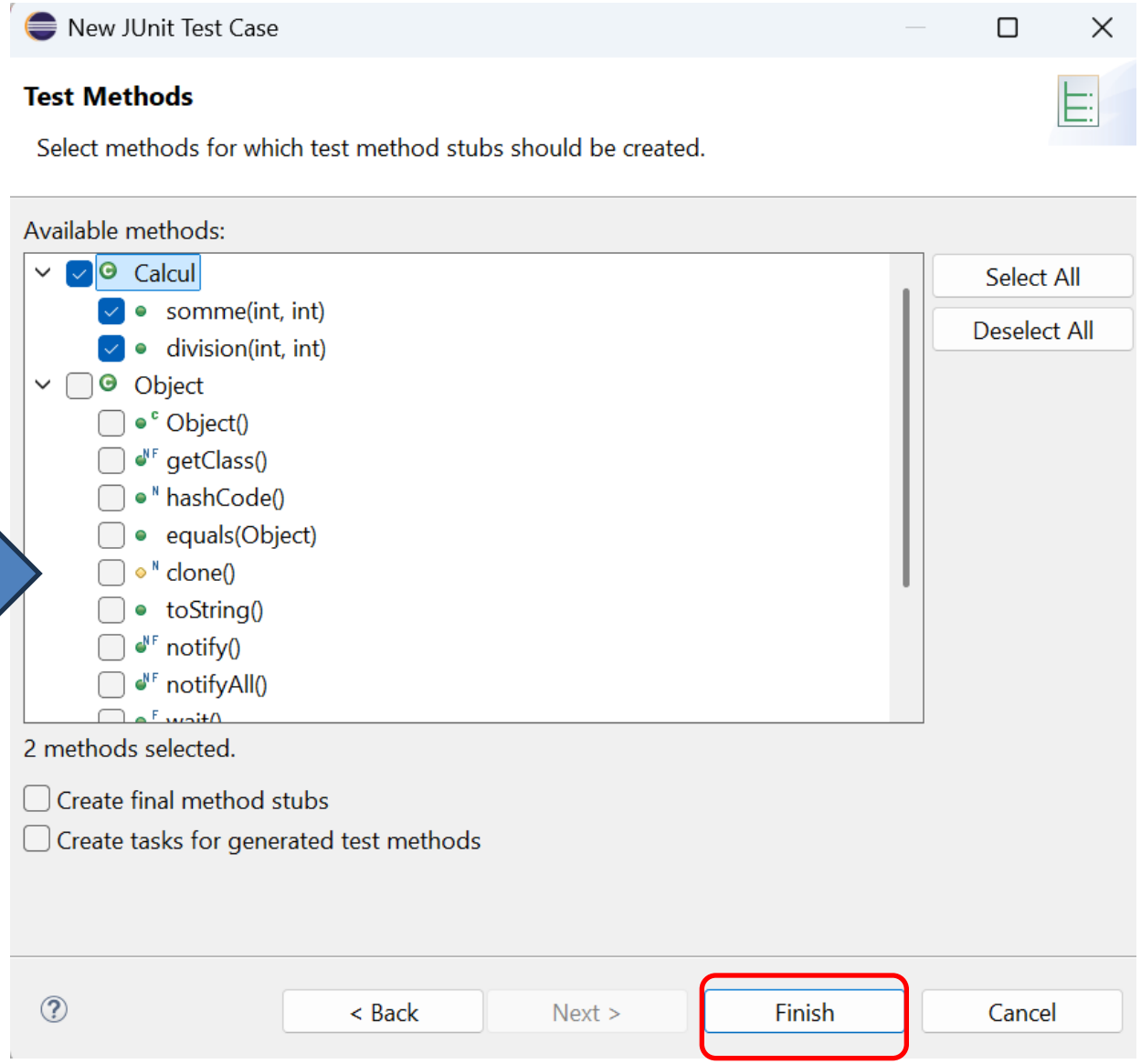
Exemple

- Etape 3 : Créer Junit Test Case

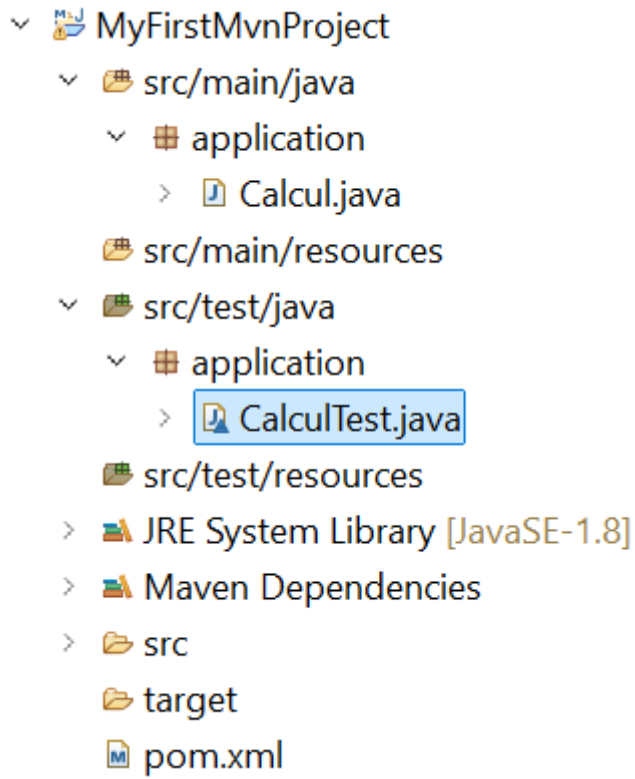


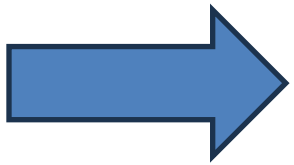
Exemple


New JUnit Test Case
JUnit Test Case
Select the name of the new JUnit test case. Specify the class under test to select methods to be tested on the next page.
New JUnit 3 test
New JUnit 4 test
New JUnit Jupiter test
Source folder:
MyFirstMvnProject/src/test/java
Browse...
Package:
application
Browse...
Name:
CalculTest
Superclass:
java.lang.Object
Browse...
Which method stubs would you like to create?
@BeforeAll setUpBeforeClass()
@AfterAll tearDownAfterClass()
@BeforeEach setUp()
@AfterEach tearDown()
constructor
Do you want to add comments? (Configure templates and default value here)
Generate comments
Class under test:
application.Calcul
Browse...
< Back
Next >
Finish
Cancel


New JUnit Test Case
Test Methods
Select methods for which test method stubs should be created.
Available methods:
Calcul
somme(int, int)
division(int, int)
Object
Object()
getClass()
hashCode()
equals(Object)
clone()
toString()
notify()
notifyAll()
Select All
Deselect All
2 methods selected.
Create final method stubs
Create tasks for generated test methods
< Back
Next >
Finish
Cancel

Exemple





Calcul.java | CalculTest.java

```
1  package application;
2
3  import static org.junit.jupiter.api.Assertions.*;
10
11 class CalculTest {
12
13     @BeforeAll
14     static void setUpBeforeClass() throws Exception {
15     }
16
17     @AfterAll
18     static void tearDownAfterClass() throws Exception {
19     }
20
21     @BeforeEach
22     void setUp() throws Exception {
23     }
24
25     @AfterEach
26     void tearDown() throws Exception {
27     }
28
29     @Test
30     void testSomme() {
31         fail("Not yet implemented");
32     }
33
34     @Test
35     void testDivision() {
36         fail("Not yet implemented");
37     }
38
```

Exemple

La méthode annotée

- **@Test** : pour identifier les méthodes des cas de tests
- **@BeforeAll** : sera exécutée seulement avant le premier test
- **@AfterAll** : sera exécutée seulement après le dernier test
- **@BeforeEach** : sera exécutée avant chaque test
- **@AfterEach** : sera exécutée après chaque test

Calcul.java | CalculTest.java

```
1  package application;
2
3  import static org.junit.jupiter.api.Assertions.*;
10
11 class CalculTest {
12
13     @BeforeAll
14     static void setUpBeforeClass() throws Exception {
15     }
16
17     @AfterAll
18     static void tearDownAfterClass() throws Exception {
19     }
20
21     @BeforeEach
22     void setUp() throws Exception {
23     }
24
25     @AfterEach
26     void tearDown() throws Exception {
27     }
28
29     @Test
30     void testSomme() {
31         fail("Not yet implemented");
32     }
33
34     @Test
35     void testDivision() {
36         fail("Not yet implemented");
37     }
38
```

Junit 4 vs Junit 5 - annotations

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	`@Test`	`@Test`
Execute before all test methods in the current class	`@BeforeClass`	`@BeforeAll`
Execute after all test methods in the current class	`@AfterClass`	`@AfterAll`
Execute before each test method	`@Before`	`@BeforeEach`
Execute after each test method	`@After`	`@AfterEach`
Disable a test method / class	`@Ignore`	`@Disabled`

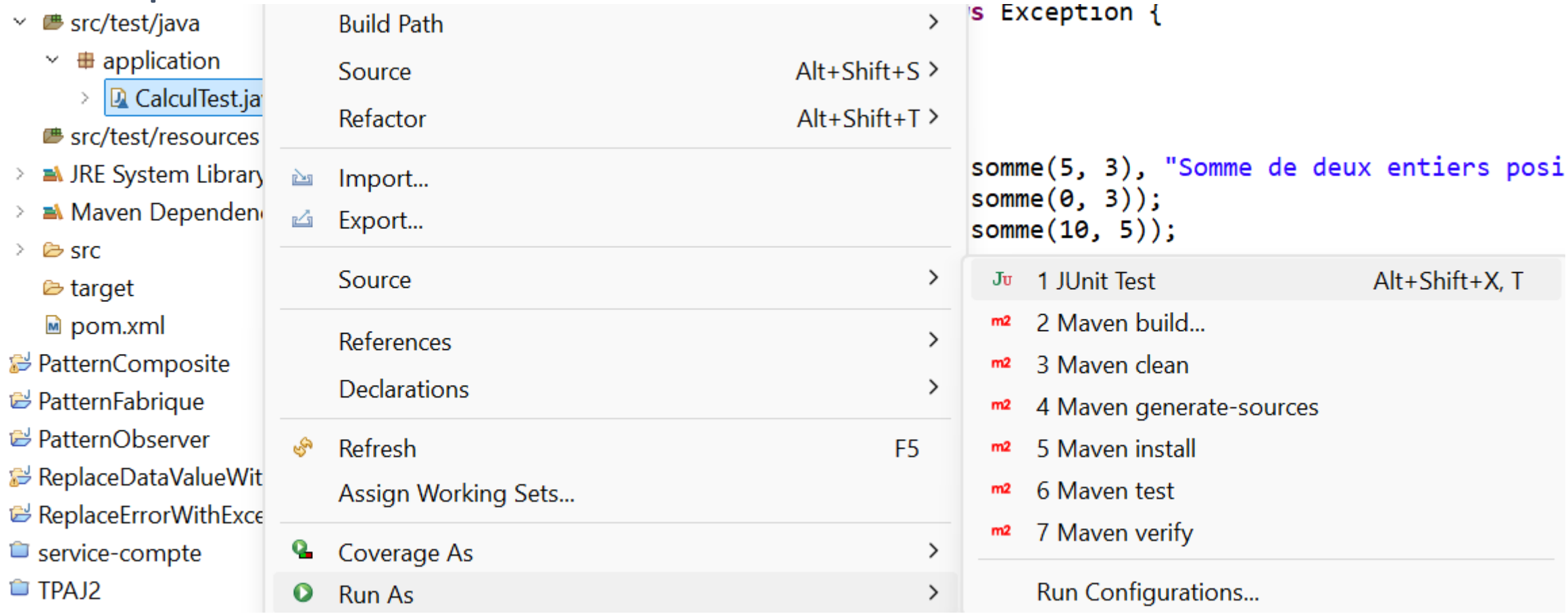
Exemple

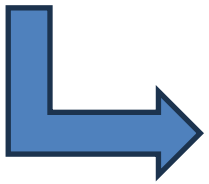
- Step 4 : Implémenter la méthode **testSomme()** et la méthode **testDivision()** en testant chaque fois les cas particuliers

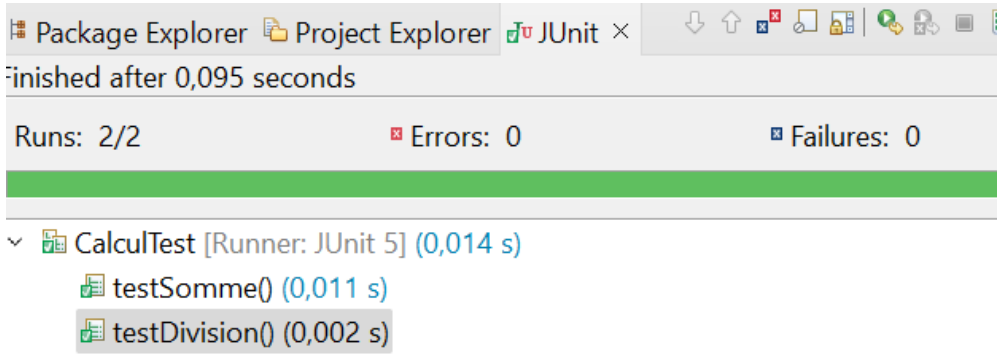
Calcul.java | CalculTest.java

```java
package application;

import static org.junit.jupiter.api.Assertions.*;

class CalculTest {

    Calcul c;

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
        c=new Calcul();
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testSomme() {
        assertEquals(8,c.somme(5, 3), "Somme de deux entiers positifs 5+3=8");
        assertEquals(3,c.somme(0, 3));
        assertTrue(15==c.somme(10, 5));

    }

    @Test
    void testDivision() {
        assertEquals(5,c.division(15, 3));
        assertThrows(ArithmeticException.class, ()->{c.division(5, 0);});
    }

}
```

Exemple - exécution

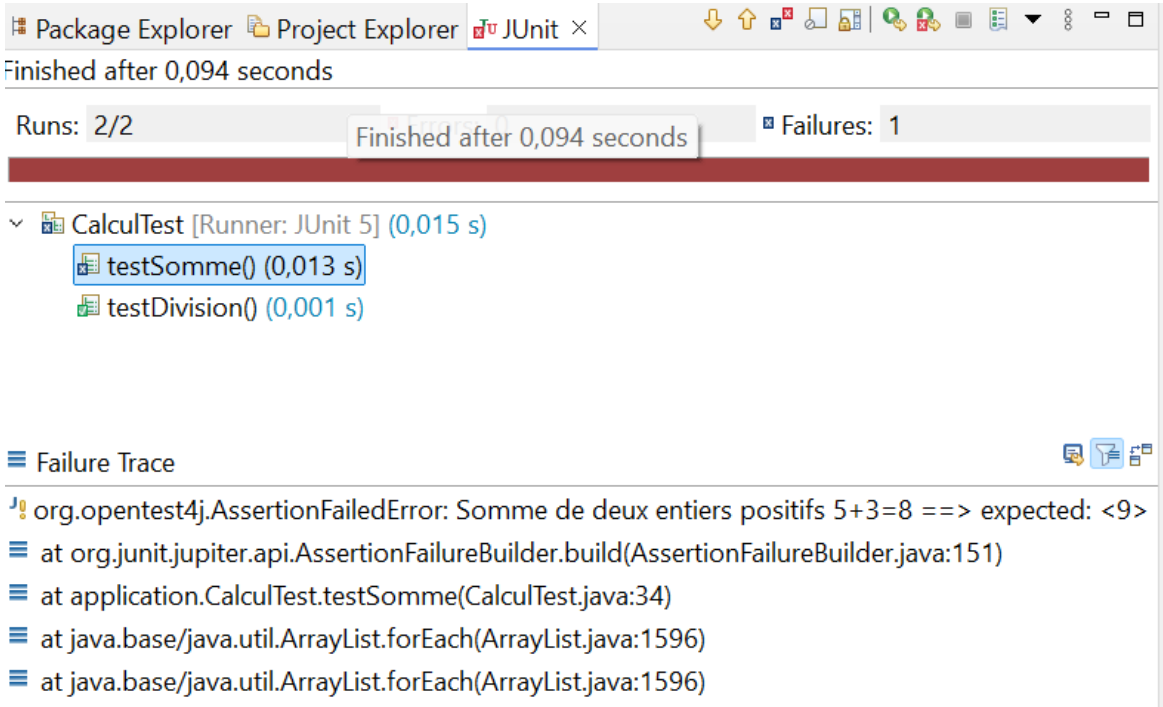


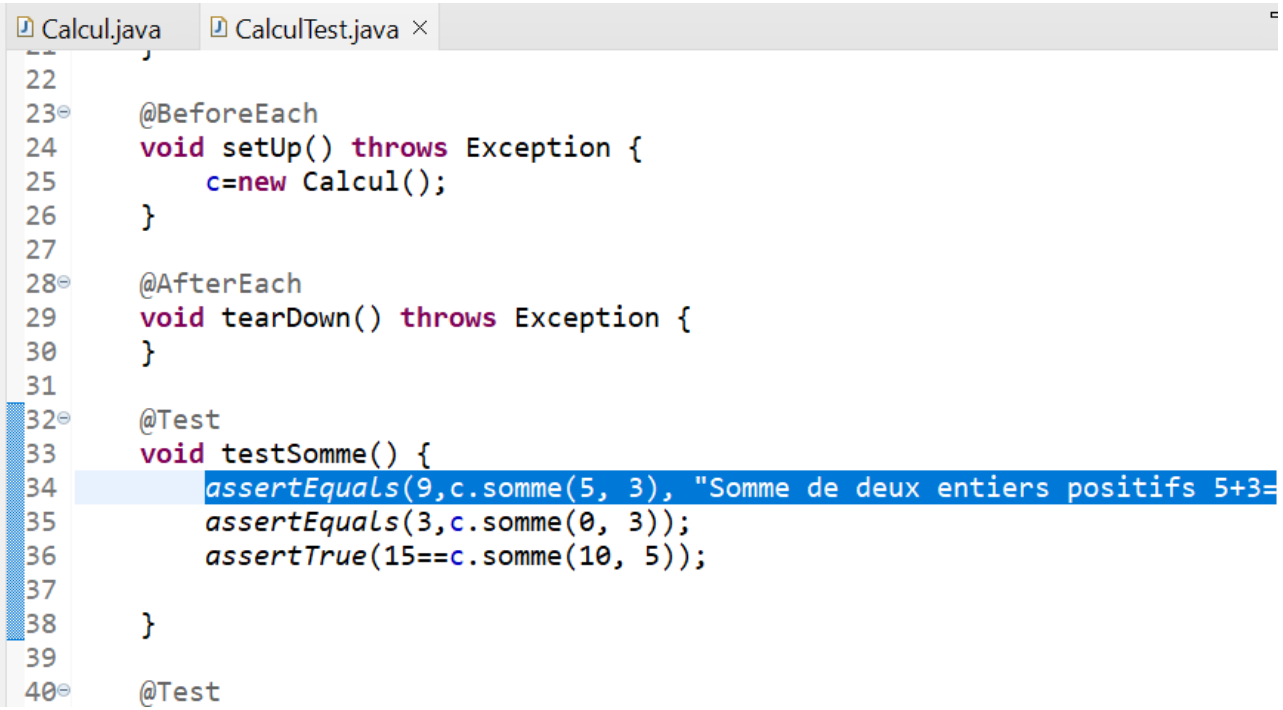




Exemple

- Provoquer une erreur dans le code de la méthode **testsomme**()

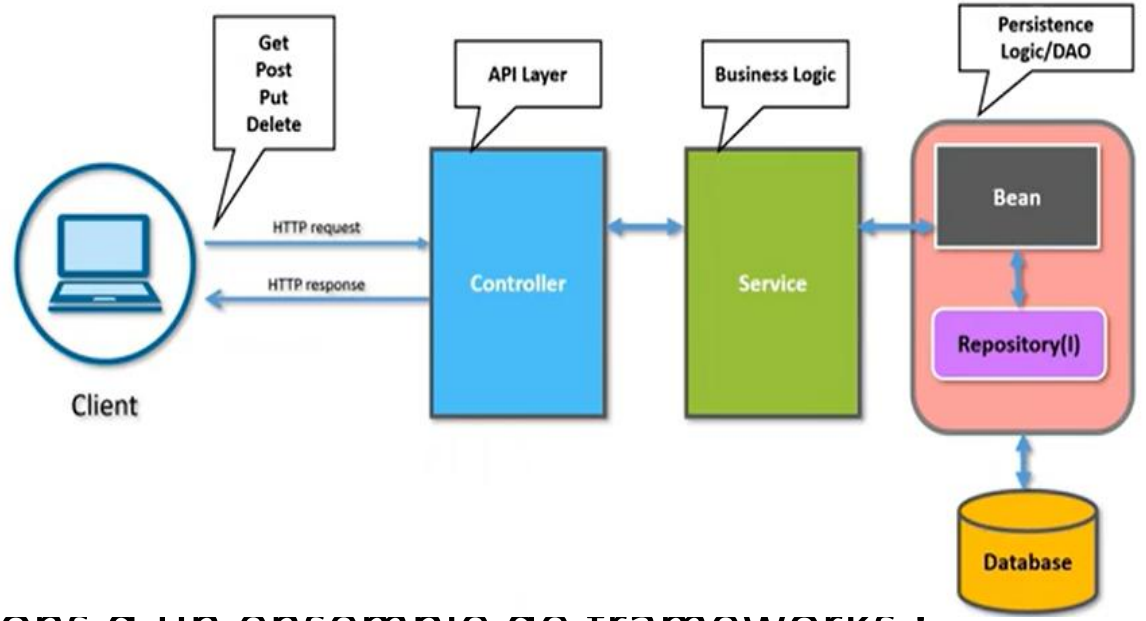




7. Les tests unitaires dans une application micro-service (springboot, Junit, Mockito, MockMVC, H2 database)

Comment tester les micro-services avec une api Rest ?

- Les tests unitaires peuvent être appliqués sur les couches intermédiaires (la couche contrôleur et la couche service).
  - Tester toutes les méthodes dans la couche service
  - Tester toutes les méthodes dans la couche contrôleur



- Pour tester les différentes couches, nous disposons d'un ensemble de frameworks :
  - **Junit**: écrire des cas de test unitaires
  - **Mockito** : simuler les dépendances externes (exemple, les méthodes de la couche service dépendent de la couche DAO. Cette dernière représente une dépendance externe. Ce framework peut être utilisé dans n'importe quel projet java.
  - **MockMVC** : invoquer les méthodes du contrôleur sans démarrer le serveur Spring Boot
  - **H2 database** : utilisé dans les tests afin de simuler une base de données réelle

Remarque : La dépendance **spring-boot-starter-test** inclut Mockito et MockMVC.

La couche service – les tests unitaires

@Mock

- Créer des objets simulés (mocks) de classes ou d'interfaces. Lorsqu'une méthode est appelée sur un objet mocké, elle ne s'exécute pas réellement, mais retourne des valeurs prédéfinies. Cela permet de tester une classe isolée de ses dépendances externes, en remplaçant ces dépendances par des objets mockés.

```
23 @SpringBootTest(classes= {ServiceMockitoTests.class})
24 @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
25
26 public class ServiceMockitoTests {
27
28     @Mock
29     CountryRepository countryrep;
30
31     @InjectMocks
32     CountryService countryService;
```

@InjectMocks

- Créer une instance de la classe que vous testez.
- Elle injecte automatiquement les objets simulés (créés avec @Mock) dans les champs de la classe testée.

La couche service – les tests unitaires

when().thenReturn() est une méthode qui permet de définir le comportement simulé d'un objet mocké. Plus précisément, when() est utilisé pour spécifier quelle méthode d'un objet mocké sera appelée, et thenReturn() est utilisé pour indiquer quelle valeur cet appel de méthode doit retourner.

```
@Test
@Order(1)
public void test_getAllCountries() {
    List<Country> mycountries = new ArrayList<Country>();
    mycountries.add(new Country(1,"India","Delhi"));
    mycountries.add(new Country(2,"USA","Washington"));

    when(countryrep.findAll()).thenReturn(mycountries);//Mocking
    assertEquals(2, countryService.getAllCountries().size());

}
```

Les données simulées

La couche contrôleur – les tests unitaires

- Il existe deux types de tests unitaires :
  - Tester les méthodes du contrôleur **== > Framework Mockito**
  - Tester les méthodes du contrôleur en passant les URLs (get, post, put et delete) **== > Framework Mockito et MockMVC**
    - MockMVC permet de tester les méthodes en passant les URLs sans démarrer le serveur Spring Boot

La couche contrôleur – les tests unitaires

```java
@SpringBootTest(classes= {ControllerMockitoTests.class})
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class ControllerMockitoTests {

    @Mock
    CountryService countryService;

    @InjectMocks
    CountryController countryController;

    public List<Country> mycountries;
    Country country;

    @Test
    @Order(1)
    public void test_getAllCountries() {
        mycountries = new ArrayList<Country>();
        mycountries.add(new Country(1,"India","Delhi"));
        mycountries.add(new Country(2,"USA","Washington"));

        when(countryService.getAllCountries()).thenReturn(mycountries);//Mocking
        ResponseEntity<List<Country>> res = countryController.getCountries();

        assertEquals(HttpStatus.FOUND, res.getStatusCode());
        assertEquals(2, res.getBody().size());

    }
```

- Tester les méthodes du contrôleur **avec le Framework Mockito**
- La classe **ResponseEntity** en Spring est utilisée pour représenter une réponse HTTP complète, permettant de contrôler le code de statut, les en-têtes et le corps de la réponse (exemple code 302 ==FOUND)

La couche contrôleur – les tests unitaires

- Tester les méthodes du contrôleur en passant les URLs **avec le Framework MockMVC et Mockito**

```
@SpringBootTest(classes= {ControllerMockMvcTests.class})
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@ComponentScan(basePackages = "com.countryservice.demo")
@AutoConfigureMockMvc

public class ControllerMockMvcTests {

    @Autowired
    MockMvc mockMvc;

    @Mock
    CountryService countryService;

    @InjectMocks
    CountryController countryController;

    public List<Country> mycountries;
    Country country;

    @BeforeEach
    public void setUp() {
        mockMvc=MockMvcBuilders.standaloneSetup(countryController).build();
    }
```

@ComponentScan indique à Spring de rechercher des classes annotées avec des stéréotypes de composant (tels que @Component, @Service, @Repository, @Controller) dans un package spécifique.

@AutoConfigureMockMvc automatise l'injection de MockMvc dans votre classe de test, vous permettant de tester vos contrôleurs Spring de manière isolée et efficace.

MockMvcBuilders.standaloneSetup() signifie la création d'une instance de MockMvc de manière autonome, sans dépendre d'un contexte web complet

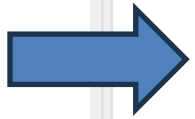
La couche contrôleur – les tests unitaires

- Exemple de test unitaire de la méthode getCountryByID() en passant l'URL /getcountries/{id}

```
@Test
@Order(2)
public void test_getCountryByID() throws Exception {
    country = new Country(1,"India","Delhi");

    int countryID=1;
    when(countryService.getCountryById(countryID)).thenReturn(country);//Mocking

    this.mockMvc.perform(get("/getcountries/{id}", countryID))
    .andExpect(status().isFound())
    .andExpect(MockMvcResultMatchers.jsonPath(".idCountry").value(1)) //private names attributes of country
    .andExpect(MockMvcResultMatchers.jsonPath(".name").value("India"))
    .andExpect(MockMvcResultMatchers.jsonPath(".capital").value("Delhi"))
    .andDo(print());
}
```

- Au niveau de la console :

```
MockHttpServletRequest:
      HTTP Method = GET
      Request URI = /getcountries/1
       Parameters = {}
          Headers = []
             Body = <no character encoding se
    Session Attrs = {}
```



```
MockHttpServletResponse:
           Status = 302
    Error message = null
          Headers = [Content-Type:"application/json"]
     Content type = application/json
             Body = {"idCountry":1,"name":"India","capital":"Delhi"}
    Forwarded URL = null
   Redirected URL = null
          Cookies = []
```

La couche de persistance

- Comment tester les différentes méthodes du repository tout en isolant la base de données MySQL ?
- Il suffit d'utiliser **H2 database** qui est une base de données JAVA SQL volatile, rapide, légère.
- Pour cela, il suffit de :
  - Rajouter la dépendance dans le fichier POM.xml

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

  - Modifier le fichier application.properties

```
#MySQL
#spring.application.name=country-service
#spring.datasource.url =jdbc:mysql://localhost:3306/countries
#spring.datasource.username=root
#spring.datasource.password=root
#spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

#H2 databse
spring.application.name=country-service
spring.datasource.url =jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

Travail à faire

Couche service

- Vous allez implémenter tous les tests unitaires des méthodes existantes dans la couche service

  == > Le fichier ServiceMockitoTests.java

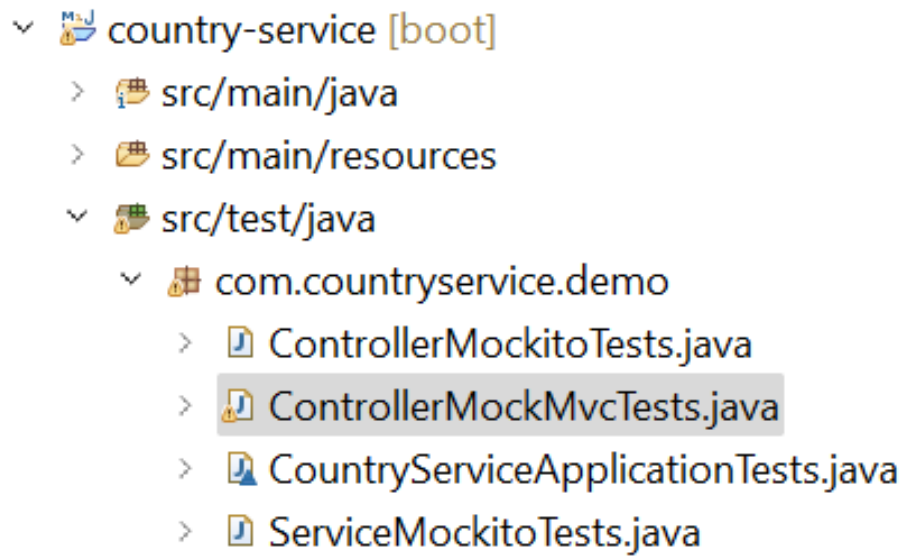
Couche contrôleur

- Vous allez implémenter tous les tests unitaires des méthodes existantes dans la couche contrôleur
  - En invoquant seulement les méthodes

  == > Le fichier ControllerMockitoTests.java

  - En invoquant les méthodes avec les URLs associés

  == > Le fichier ControllerMockMvcTests.java



Couche persistance

- Vous allez appliqué les modifications présentées précédemment.

Maven

- Essayez de lancer cette commande sur le projet spring boot: **mvn clean package**
- Qu'est ce que vous remarquez ?

- *"Apprendre par le projet, c'est découvrir par l'action, créer par la compréhension, et réussir par la persévérance."*

amina.jarraya@ensi-uma.tn

ENSI Manouba