FMS ACADEMY BY FMS

FACYLITIES MULTI SERVICES

Une entreprise ordinaire à l'agilité extraordinaire

FORMATION JAVA SPRING ANGULAR

JAVA TESTS 1

mohamed.el-babili@fms-ea.com

33+ 628 111 476

Version : 4.0

DMAJ : 06/01/24

Module : DEV-TEST-001
Maven & test unitaire

01 Connaître le processus de développement d'un projet avec Maven
02 Utiliser Maven pour injecter des dépendances
03 Utiliser Maven pour générer une application et autres fonctionnalités
04 Mise en œuvre de tests unitaires

SOMMAIRE

Pourquoi tester une application ?

Pour répondre à des impératifs (d'agilité) :

- Garantir la qualité des applications (bugs & confiance…)
- Assurer la conformité aux attentes du client (spécification fonctionnelles)
- Anticiper sur les cas d'utilisations et sur l'évolution d'une application
- Respecter les contraintes techniques & délais de livraison (réduction des couts)
- Eviter les régressions (fiabilité des applis/satisfaction des clients)
- Améliorer la communication entre les développeurs (tests servent de doc)

Comment tester une application ?

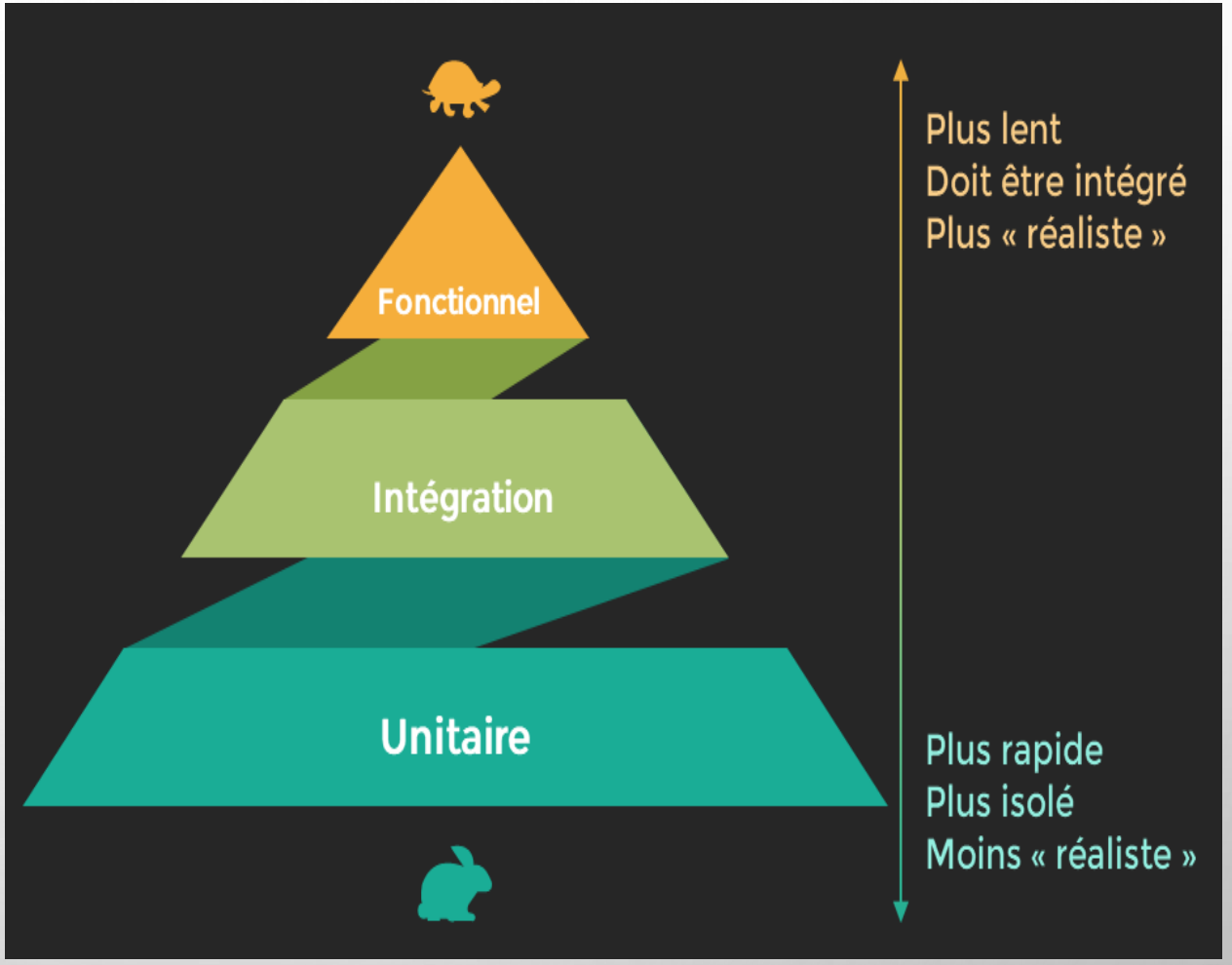
En réalisant nous-même des **tests manuels** sur les nouvelles fonctionnalités ou l'expérience utilisateur…
Le pb c'est qu'ils sont souvent incomplets voir insuffisant et ils ne tiennent pas toujours compte de l'évolution des applications.
En effet, sur des projets importants, nombreuses sont les interventions de devs, de versions en versions, est-ce qu'ils répètent tous les tests ?

En automatisant les tests, on garantit que tous les tests prévus sur chaque version d'une application seront effectués automatiquement, de-lors, on évite les régressions notamment.

En Conclusion, tester une application est un processus essentiel qui combine **l'intuition humaine des tests manuels** et **la puissance des outils d'automatisation**. Le choix de l'approche dépend des besoins, des contraintes, et de la maturité du projet. Une stratégie hybride maximise la qualité tout en optimisant le temps et les efforts.

Différents types de tests

Mike Cohn, l'un des fondateurs du mouvement Agile, présente ces types de tests sous forme de **pyramide des tests.**



Les tests fonctionnelles (end to end) visent à simuler le comportement d'un User final.

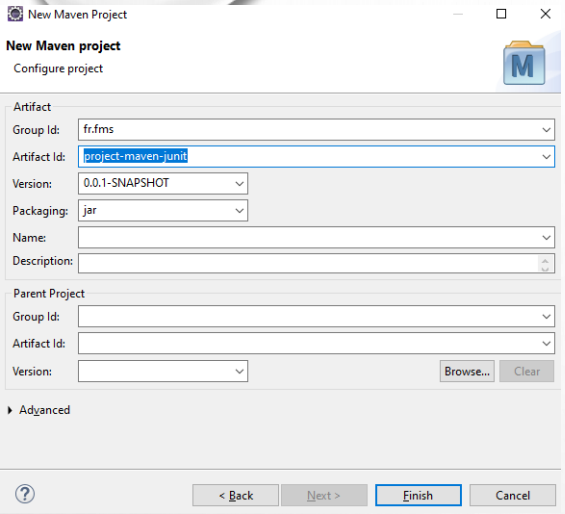
Les tests d'intégrations vérifient que les unités de code fonctionnent ensemble correctement comme prévu.
Par ex, tester l'inter action entre 2 couches

Les tests unitaires sont les plus nombreux et **les plus importants**. Ils garantissent que chaque unité de code ou fonctionnalités se comporte comme prévu.
Ils sont rapides, faciles à exécuter, stables quelle que soit les évolutions et rentables car une fois écrits, ils sont exécutés de nombreuses fois.

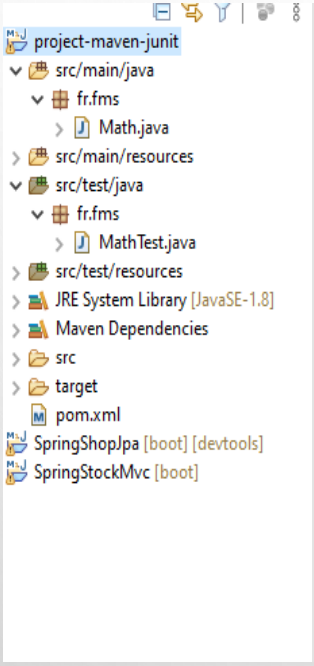
Plus un bug est détecté tôt, moins son coût de correction est élevé :

- Moins on réalise de tests unitaires, plus on va devoir faire de tests manuels pour le débusquer (mauvaise pratique)
- Plus notre couverture de tests unitaires est de qualité et importante, plus vite un bug est détecté (bonne pratique)

MAVEN ET JUNIT



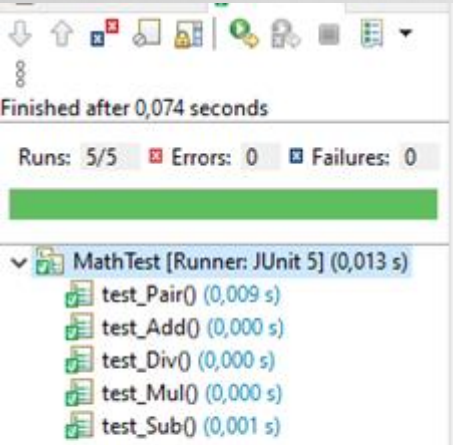
Dans cet exemple, nous avons crée un projet maven simple possédant donc un fichier pom.xml.
Il faudra donc ajouter les dépendances Junit manuellement dans le pom.xml.
Puis ajouter une classe à tester (**src/main/java/**fr.fms.Math) et une classe de test (**src/test/java/**fr.fms.MathTest) dans les répertoires dédiés




```java
package fr.fms;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;

public class MathTest {

  @Test
  void test_Add() {
    assertEquals(5, Math.add(3, 2));
  }
  @Test
  void test_Mul() {
    assertEquals(15, Math.mul(3, 5));
  }
  @Test
  void test_Sub() {
    assertEquals(2, Math.sub(5, 3));
  }
  @Test
  void test_Div() {
    assertEquals(5, Math.div(25, 5));
  }
  @Test
  void test_Pair() {
    assertTrue(Math.pair(4));
```

```java
package fr.fms;

public class Math {

      public static int add(int num1, int num2)
        return num1 + num2;
      }

      public static int mul(int num1, int num2)
        return num1 * num2;
      }

      public static int sub(int num1, int num2)
          return num1 - num2;
      }

      public static int div(int num, int by) {
        return num/by;
      }

      public static boolean pair(int num) {
          return num%2 == 0;
      }
}
```

Pour exécuter les tests sous Eclipse : Project/clic droit/run as/junit test

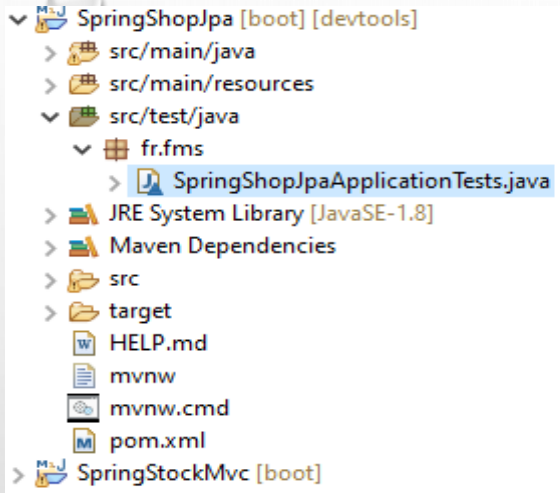


Sous IntelliJ : Project/clic droit/run all tests ou sélectionner le fichier de test/clic droit/run …

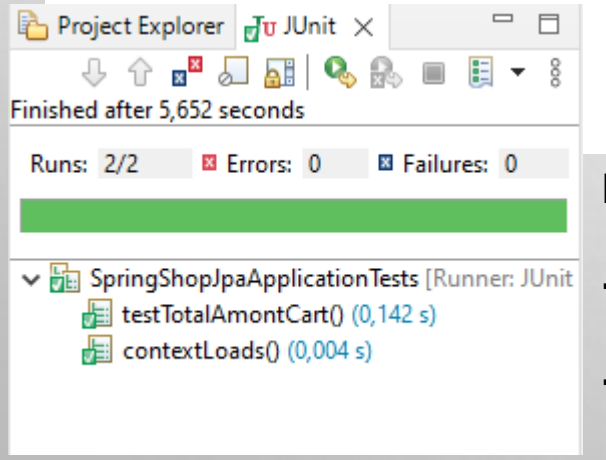
SPRING BOOT MAVEN ET JUNIT

On souhaite maintenant lancer des tests unitaires dans une application Spring Boot, nous dirons dans le contexte d'exécution de celle-ci. Nous allons trouver toujours dans le chemin **src/test/java/**fr.fms.NomAppli notre classe principale avec l'annotation @SpringBootTest. On peut injecter simplement notre couche métier et effectuer des tests basiques tels que : *vérifier si le montant total de notre panier est cohérent avec les prix des articles ajoutés qq soit les quantités.* (veiller à ce que la dépendance soit bien présente)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```




```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import fr.fms.business.IBusinessImpl;
import fr.fms.entities.Article;

@SpringBootTest
class SpringShopJpaApplicationTests {
    @Autowired
    IBusinessImpl business;

    @Test
    void contextLoads() {
        assertFalse(1==2);
    }

    @Test
    void testTotalAmontCart() {
        business.addArtToCart(new Article((long)1,"Samsung","Samsung S8",250,1,null));
        business.addArtToCart(new Article((long)2,"Samsung","Samsung S9",250,1,null));
        business.addArtToCart(new Article((long)3,"IPhone","IPhone 10",500,1,null));
        business.addArtToCart(new Article((long)1,"Samsung","Samsung S8",250,1,null));

        assertEquals(business.getTotalAmount(),1250);
    }
}
```



Pour 2 tests lancés avec succès :

- **contextLoads** vérifier que l'assertion est bien fausse

- **testTotalAmountCart** vérifie que le montant du panier est cohérent aux ajouts

Failures = nb de test en échec
Errors = nb d'erreur de compil

Project Explorer | JUnit

Finished after 9,071 seconds

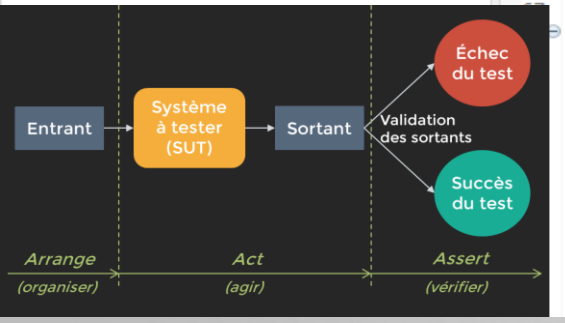
Runs: 7/7 Errors: 0 Failures: 0

- SpringShopJpaApplicationTests [Runner: JUnit
  - multiply_shouldReturnZero(int) (0,176 s)
    - 1 x 0 doit être égal à 0 (0,176 s)
    - 2 x 0 doit être égal à 0 (0,008 s)
    - 42 x 0 doit être égal à 0 (0,005 s)
    - 1011 x 0 doit être égal à 0 (0,003 s)
    - 5089 x 0 doit être égal à 0 (0,004 s)
  - orderShouldComputeLess1Second() (0,511
  - testTotalAmontCart() (0,006 s)

Failure Trace

Console | SpringShopJpaApplicationTests.java | SpringShopJpa/pom.xml | ArticleRepository.java

```java
package fr.fms;

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

@SpringBootTest
class SpringShopJpaApplicationTests {
    @Autowired
    IBusinessImpl business;

    private static Instant startedAt;

    @BeforeEach
    public void beforeEachTest() {
        System.out.println("avant chaque test");
    }

    @AfterEach
    public void afterEachTest() {
        System.out.println("après chaque test");
    }

    @BeforeAll
    public static void initStartingTime() {
        System.out.println("Appel avant tous les tests");
        startedAt = Instant.now();
    }

    @AfterAll
    public static void showTestDuration() {
        System.out.println("Appel après tous les tests");
        final Instant endedAt = Instant.now();
        final long duration = Duration.between(startedAt, endedAt).toMillis();
        System.out.println(MessageFormat.format("Durée des tests : {0} ms", duration));
    }

    @ParameterizedTest(name = "{0} x 0 doit être égal à 0")
    @ValueSource(ints = { 1, 2, 42, 1011, 5089 })
    public void multiply_shouldReturnZero(int arg) {
        assertEquals(0, arg*0);
    }

    @Timeout(1)
    @Test
    public void orderShouldComputeLess1Second() {
        business.order();
    }

    @Test
    void testTotalAmontCart() {
        // Arrange
        business.addArtToCart(new Article((long)1,"Samsung","Samsung S8",250,1,null));
        business.addArtToCart(new Article((long)2,"Samsung","Samsung S9",250,1,null));
        business.addArtToCart(new Article((long)3,"IPhone","IPhone 10",500,1,null));

        // Act
        double amount = business.getTotalAmount();

        // Assert
        assertEquals(amount,1000);
    }
```




```
Appel avant tous les tests
13:51:18.517 [main] DEBUG org.springframewo
13:51:18.522 [main] DEBUG org.springframewo

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.0)

2022-06-26 13:51:18.878  INFO 16604 --- [
2022-06-26 13:51:18.879  INFO 16604 --- [
2022-06-26 13:51:20.017  INFO 16604 --- [
2022-06-26 13:51:20.114  INFO 16604 --- [
2022-06-26 13:51:21.026  INFO 16604 --- [
2022-06-26 13:51:21.122  INFO 16604 --- [
2022-06-26 13:51:21.396  INFO 16604 --- [
2022-06-26 13:51:21.665  INFO 16604 --- [
2022-06-26 13:51:21.759  INFO 16604 --- [
2022-06-26 13:51:21.780  INFO 16604 --- [
2022-06-26 13:51:22.855  INFO 16604 --- [
2022-06-26 13:51:22.866  INFO 16604 --- [
2022-06-26 13:51:23.859  WARN 16604 --- [
2022-06-26 13:51:24.403  INFO 16604 --- [
2022-06-26 13:51:25.981  INFO 16604 --- [

Using generated security password: be568f54

2022-06-26 13:51:26.007  INFO 16604 --- [
avant chaque test
après chaque test
avant chaque test
après chaque test
avant chaque test
après chaque test
avant chaque test
après chaque test
avant chaque test
après chaque test
avant chaque test
après chaque test
avant chaque test
après chaque test
Appel après tous les tests
Durée des tests : 8 362 ms
```

TDD : Test Driven Développement
Ou
Développement piloté par les tests

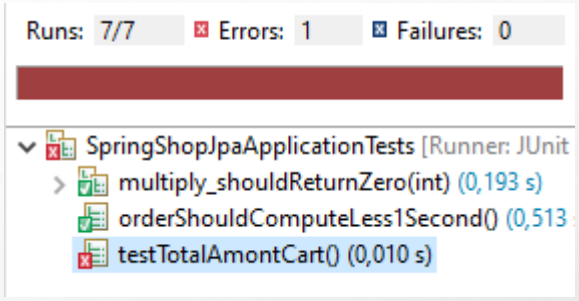
En règle générale, on va réaliser le test associé à une fonctionnalité, on parle de codage du test en fonction du besoin. Après quoi, on lance le test, il sera en échec dans un 1er temps. Ensuite, on va coder la fonctionnalité et relancer le test

```
68    @Test
69    void testTotalAmontCart() {
70        // Arrange
71        business.addArtToCart(new Article((long)1,"Samsung","Samsung S8",250,1,null));
72        business.addArtToCart(new Article((long)2,"Samsung","Samsung S9",250,1,null));
73        business.addArtToCart(new Article((long)3,"IPhone","IPhone 10",500,1,null));
74
75        // Act
76        double amount = business.getTotalAmount();
77
78        // Assert
79        assertEquals(amount,1000);
80    }
```

The method getTotalAmount() is undefined for the type IBusinessImpl

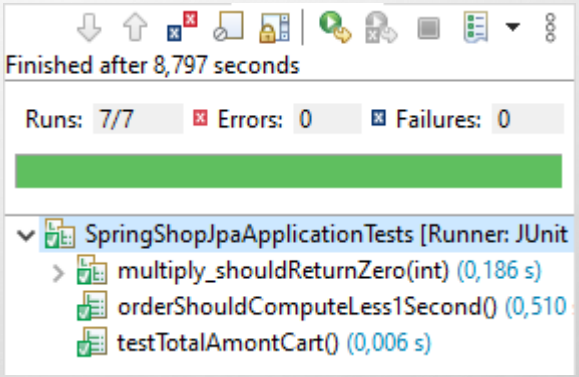
2 quick fixes available:

Create method 'getTotalAmount()' in type 'IBusinessImpl'



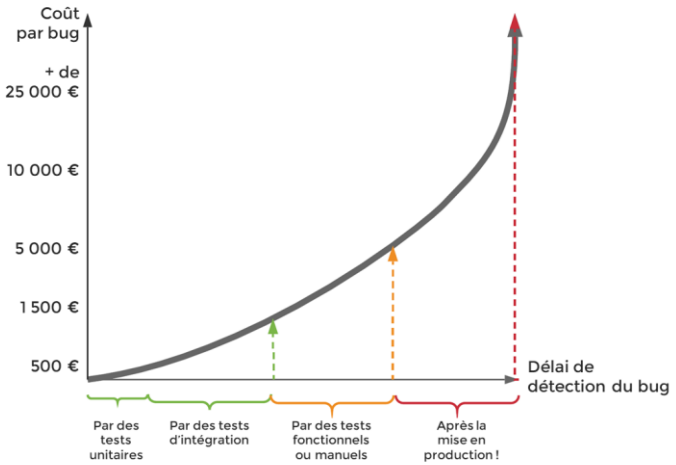
Puis on ajoute le code dans la méthode de la classe :

```
public double getTotalAmount() {
    double total = 0;
    for(Article article : cart.values()) {
        total += article.getPrice()*article.getQuantity();
    }
    return total;
}
```



Cette approche garantie qu'on reste centré sur les fonctionnalités
De plus, le code écrit après le test est plus simple à tester,
plus clair à comprendre et plus facile à maintenir : Robuste

ASSERTJ



La courbe montre que + un bug est découvert tardivement, plus il est couteux de le corriger aussi comment écrire des tests :

- Simples (à tester)
- Clairs (à comprendre)
- Faciles (à maintenir)

→ AssertJ apporte à Junit tout ce complément

Cas de test	**Assertions JUnit**	**Assertions AssertJ**
Un nom est compris entre 5 et 10 caractères.	assertTrue(name.length > 4 && name.length < 11);	assertThat(name) .hasSizeGreaterThan(4) .hasSizeLessThan(11);
Un nom est situé dans la première moitié de l'alphabet	assertTrue(name.compareTo("A") >= 0 && name.compareTo("M") <= 0);	assertThat(name).isBetween("A", "M");
Une date et heure locale se situent aujourd'hui ou dans le futur.	assertTrue(dateTime.toLocalDate().isAfter(LocalDate.now()) || dateTime.toLocalDate().isEqual(LocalDate.now()));	assertThat(dateTime.toLocalDate()) .isAfterOrEqualTo(LocalDate.now());

@DataJpaTest

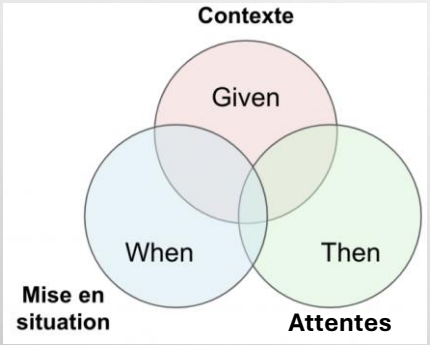
Si vous souhaitez maintenant tester vos entités Jpa dans une base de données virtuelle avec l'accès aux données persistantes sans modifications possibles :

```java
import org.junit.jupiter.api.Test;

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class SpringShopJpaTests {
    @Autowired
    ArticleRepository articleRepository;

    @Autowired
    CategoryRepository categoryRepository;

    @Test
    void test_add_article() {
        //GIVEN
        Category anonymous = categoryRepository.save(new Category(null,"anonymous",null));
        articleRepository.save(new Article(null,"incognito","incognito 007" , 375 , 1 , anonymous));

        //WHEN
        Article article = articleRepository.findByBrandContains("incognito").get(0);

        //THEN
        assertEquals("incognito 007", article.getDescription());
    }

    @Test
    void should_find_one_article() {
        Iterable<Article> articles = articleRepository.findAll();
        assertThat(articles).isNotEmpty();
    }
}
```

Permet de charger le contexte Spring avant l'exécution des tests. Si vous utilisez junit 5 (jupiter) cette annotation est inutile.

Fournit un contexte Spring léger pour tester JPA (entités, répo), configure une bdd type H2

Permet de tester avec une base de données réelle



Il y a un problème ici, lequel ?

Notre base n'a subi aucun changement

```
| 21 | Panasonic | HT           |  1500 |        1 |          6 |
| 22 | Philips   | L43          |   450 |        1 |          6 |
| 26 | samsung   | samsung S200 |  1500 |        1 |          1 |
| 29 | Iphone    | Iphone 10    |   500 |        1 |          1 |
+----+-----------+--------------+-------+----------+------------+
24 rows in set (0.000 sec)

MariaDB [Stock]>
```

Introduction à Mockito

Lorsqu'on teste un composant (par ex, un controller ou un service), il peut dépendre d'autres classes (par ex, un repository). Ces dépendances rendent les tests difficiles, car elles ajoutent des comportements externes non contrôlés.

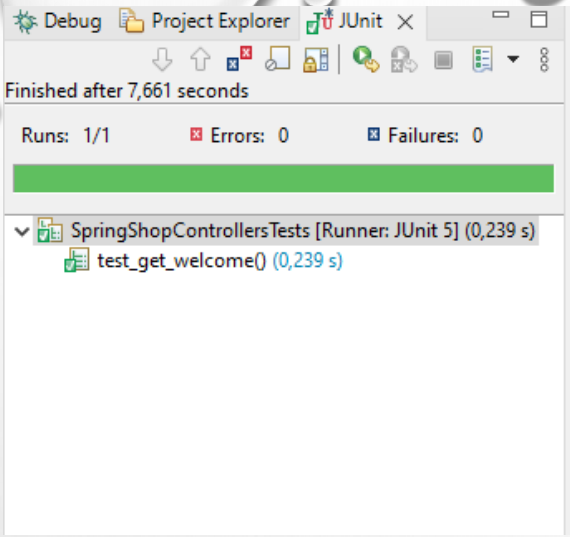
Mockito permet de "mock" ces dépendances, c'est-à-dire de simuler leur comportement, afin de tester uniquement le composant cible.

Un Mock est un objet simulé qui imite le comportement d'une vraie classe sans en exécuter le code réel.

Mocking : Créer un mock d'une classe ou d'une interface.
Stubbing : Configurer le comportement d'un mock pour des cas spécifiques.
Verification : Vérifier qu'un mock a été utilisé comme prévu.

En ajoutant la dépendance de test « spring-boot-starter-test » plusieurs bibliothèques sont incluses :
JUnit 5 (Jupiter) : Pour écrire et exécuter des tests.
Mockito : Pour le mocking et le stubbing.
AssertJ : Pour des assertions riches et fluides.
Hamcrest : Fournit des matchers pour les assertions.
JsonPath : Pour interagir avec les réponses JSON.
Spring Test : Outils spécifiques à Spring pour les tests d'intégration.

Tester un Controller
(@MockBean & @MockMvc)



SpringShopControllersTests.java

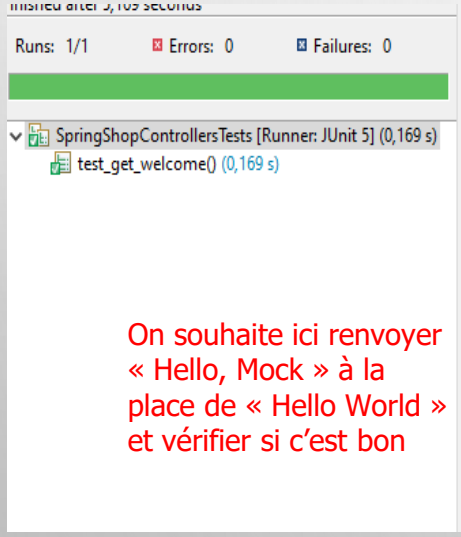
```
1  package fr.fms;
2
3  import static org.hamcrest.CoreMatchers.containsString;
22
23 @SpringBootTest
24 @AutoConfigureMockMvc
25 public class SpringShopControllersTests {
26
27     @Autowired
28     private MockMvc mvc;
29
30     @Test
31     public void test_get_welcome() throws Exception {
32         //GIVEN
33         mvc.perform(get("/"))
34             .andExpect(status().isOk())
35             .andExpect(content().string(containsString("Hello")));
36     }
37 }
```

Configure MockMvc pour effectuer des tests HTTP sans démarrer un serveur

est utilisé pour exécuter une requête GET et vérifier les réponses.

```
@RequestMapping("/")
public @ResponseBody String home() {
    return "Hello";
}
```

Cette méthode gère les méthodes envoyées à la racine de l'application (/), elle accepte toutes les méthodes HTTP. @ResponseBody indique que le retour de la méthode est directement écrit dans le corps de la réponse HTTP (et non une vue Html)




```
import static org.mockito.Mockito.when;
3  import static org.hamcrest.CoreMatchers.containsString;
24
25 @WebMvcTest(ArticleController.class)
26 public class SpringShopControllersTests {
27
28     @Autowired
29     private MockMvc mvc;
30
31     @MockBean
32     private IBusinessImpl businessImpl;
33
34     @Test
35     public void test_get_welcome() throws Exception {
36         when(businessImpl.great()).thenReturn("Hello, Mock");
37
38         this.mvc.perform(get("/greating"))
39             .andDo(print())
40             .andExpect(status().isOk())
41             .andExpect(content()
42             .string(containsString("Hello, Mock")));
43     }
44 }
```

Permet de tester un **composant** de la couche Web, aucun autre composant de Spring boot ne sera chargé

Permet de remplacer une dépendance réelle par un mock, ici IBusinessImpl

```
public String great() {
    return "Hello World";
}
```

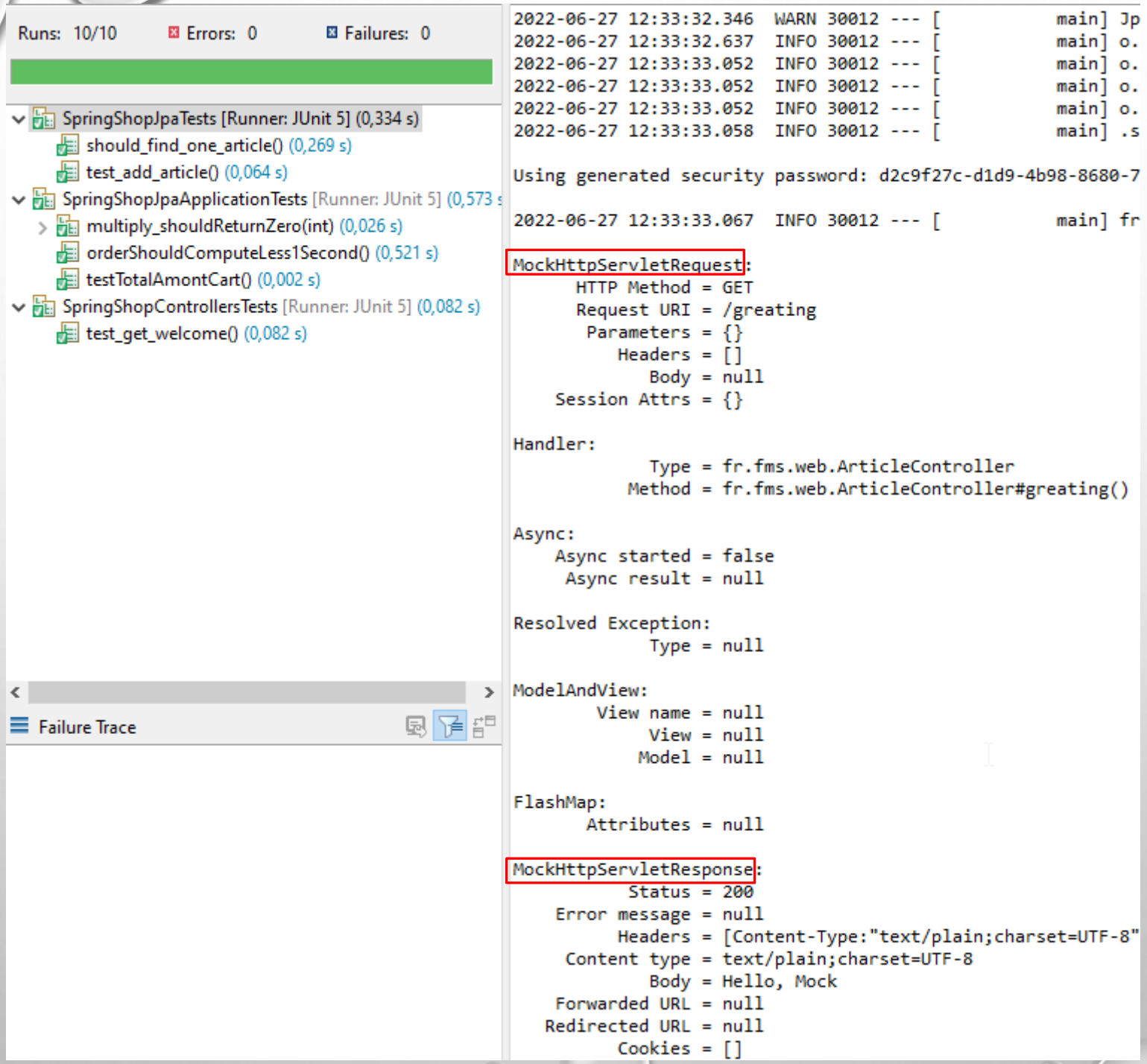
On souhaite ici renvoyer « Hello, Mock » à la place de « Hello World » et vérifier si c'est bon

```
@RequestMapping("/greating")
public @ResponseBody String greating() {
    return businessImpl.great();
}
```

Voir résultat slide suivante

Ex avec une api, à suivre

```
@Test
public void test_get_articles() throws Exception {
    //.................
    this.mvc.perform(get("/articles"))  //THEN
        .andExpect(status().isOk())
        .andExpect(jsonPath("$[0].brand", is("Samsung")));
        //$    : pointe sur la racine de la structure JSON
        //[0]  : pour le 1er élément
        //brand: brand pour l'attribut
        //is   : correspond au résultat attendu
}
```



Tester un Service
(@Mock & @InjectMocks)

```java
@SpringBootTest
@AutoConfigureMockMvc
public class IBusinessArticleTest {

    public static Article article;
    public static Category category;

    @Mock
    ArticleRepository articleRepository;

    @Mock
    CategoryRepository categoryRepository;

    @InjectMocks
    IBusinessArticleImpl businessArticle;

    @BeforeAll
    static void initializeData(){
        category = new Category( id: 1L, name: "Télévision", articleList: null);
        article = new Article( id: 1L, name: "Ecran plat",category);
    }
```

Nous avons un service qui utilise plusieurs repository. Comment s'assurer qu'un échec de nos tests vient forcément d'un problème avec notre service, et non pas avec l'un des repository?
On utilise Mockito pour créer un mock de nos repository, puis on donne dans nos tests le comportement à suivre lorsque ces mocks sont appelés. Ainsi on contrôle ce que nos repository nous renvoient, et donc on évite toute erreur pouvant provenir de ces derniers.

```java
@Test
void should_find_by_category(){
    //GIVEN
    List<Article> articleList = new ArrayList<>();
    articleList.add(article);
    when(categoryRepository.findById(category.getId())).thenReturn(Optional.ofNullable(category));
    when(articleRepository.findByCategory(category)).thenReturn(articleList);

    //WHEN
    final List<Article> expectedResults = businessArticle.findByCategory(category);

    //THEN
    verify(articleRepository).findByCategory(category);
    assertNotEquals( unexpected: 0,expectedResults.size());
}
```

On utilise when().thenReturn() pour dicter le comportement de l'élément mocké.
Ici, on teste la logique de notre service sans jamais avoir besoin d'accéder à notre base de données.

verify(*mock*).*nomDeMethode* vérifie que l'on appelle bien la méthode de notre mock.

La méthode du service qui est testée

```java
@Override
public List<Article> findByCategory(Category category){
    if (categoryRepository.findById(category.getId()).isPresent()){
        return articleRepository.findByCategory(category);
    } else {
        return new ArrayList<>();
    }
}
```

Mock Vs MockBean

@Mock & @InjectMocks Vs @MockBean & @MockMvc

Les 2 premiers ne nécessitent pas de contexte Spring alors que les 2 autres oui

Critère	`@Mock` & `@InjectMocks`	`@MockBean` & `@MockMvc`
Contexte requis	Aucun, indépendant de Spring.	Contexte Spring requis.
Performance	Rapide.	Plus lent, car démarre le contexte Spring.
Facilité de mocking	Manuel, nécessite de créer des mocks avec Mockito.	Automatique avec injection dans le contexte Spring.
Couverture	Tests unitaires purs.	Tests d'intégration ou tests de contrôleurs HTTP.

Utiliser les 1ers si vous souhaitez réaliser des tests unitaires rapides et isolés
Vs
Utiliser la seconde approche si vous souhaitez réaliser des tests de contrôleurs HTTP ou d'intégrations

3 niveaux de Tests à la loupe

Tests unitaires :

Objectif : Les tests unitaires visent à vérifier le bon fonctionnement des parties les plus petites d'une application, généralement des méthodes ou des fonctions individuelles.

Côté serveur (Spring Boot) : Les tests unitaires côté serveur peuvent inclure des classes de service, des composants DAO (Data Access Object), des contrôleurs, etc. Utilisation de JUnit et Mockito pour simuler les dépendances et isoler les tests.

Côté client (Angular) : Les tests unitaires côté client portent généralement sur des composants Angular, des services, des pipes, etc. Jasmine et Karma sont des outils couramment utilisés pour écrire des tests unitaires Angular.

Tests d'intégration :

Objectif : Les tests d'intégration vérifient que les différentes parties d'une application fonctionnent correctement ensemble. Cela peut inclure des tests de communication entre différentes classes ou composants.

Côté serveur (Spring Boot) : Les tests d'intégration peuvent couvrir des fonctionnalités qui nécessitent la collaboration de plusieurs composants, par exemple, les tests d'intégration de l'API REST avec la base de données.

Côté client (Angular) : Les tests d'intégration côté client peuvent vérifier que les composants interagissent correctement entre eux, par exemple, que les données sont correctement transmises du service à un composant.

Tests end-to-end (E2E) :

Objectif : Les tests end-to-end simulent le parcours complet d'un utilisateur à travers l'application, du côté client jusqu'au serveur, pour s'assurer que toutes les parties fonctionnent correctement ensemble.

Côté serveur (Spring Boot) : Les tests E2E côté serveur peuvent impliquer l'appel des API REST via des requêtes HTTP simulées et la vérification des réponses.

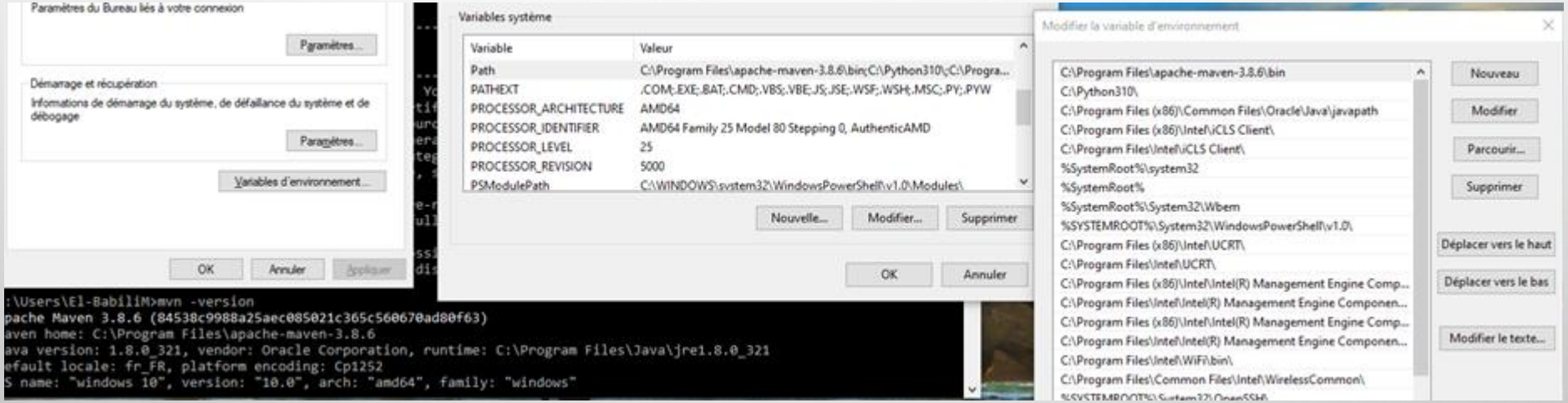
Côté client (Angular) : Les tests E2E côté client peuvent être écrits à l'aide d'outils comme Protractor ou Cypress. Ils simulent les actions de l'utilisateur (clics, saisies, etc.) et vérifient que l'application se comporte comme prévu.

Les autres tests : Tests fonctionnels, Tests de montée en charge, Tests de sécurité, Tests de compatibilité.

INSTALLER MAVEN

- Jusqu'ici, nous avons utilisé Maven pour gérer nos dépendances via le fichier Pom.xml
- Nous avons utilisé des commandes Maven via nos Ide notamment pour construire nos applications…
- Il est possible d'installer Maven manuellement afin d'interagir avec notre projet directement en ligne de commande sans l'intermédiaire d'un IDE.
- Delors, il convient de connaître les différentes commandes mais avant tout il faut installer Maven

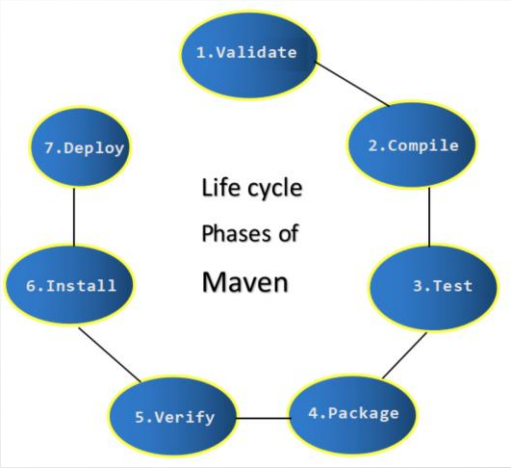
Télécharger sur le site le zip « apache-maven-3.8.6-bin.zip » le dézipper dans programmes
Puis ajouter à la variable d'environnement le chemin vers le rep bin
Tester en mode console si c'est ok



Après quoi, vous pouvez ouvrir un terminal et vous positionner à la racine d'un projet, puis lancer la commande : **mvn compile**
(Commande demandant à Maven de compiler l'ensemble du code source de votre projet)

Cycle de vie d'un projet Maven

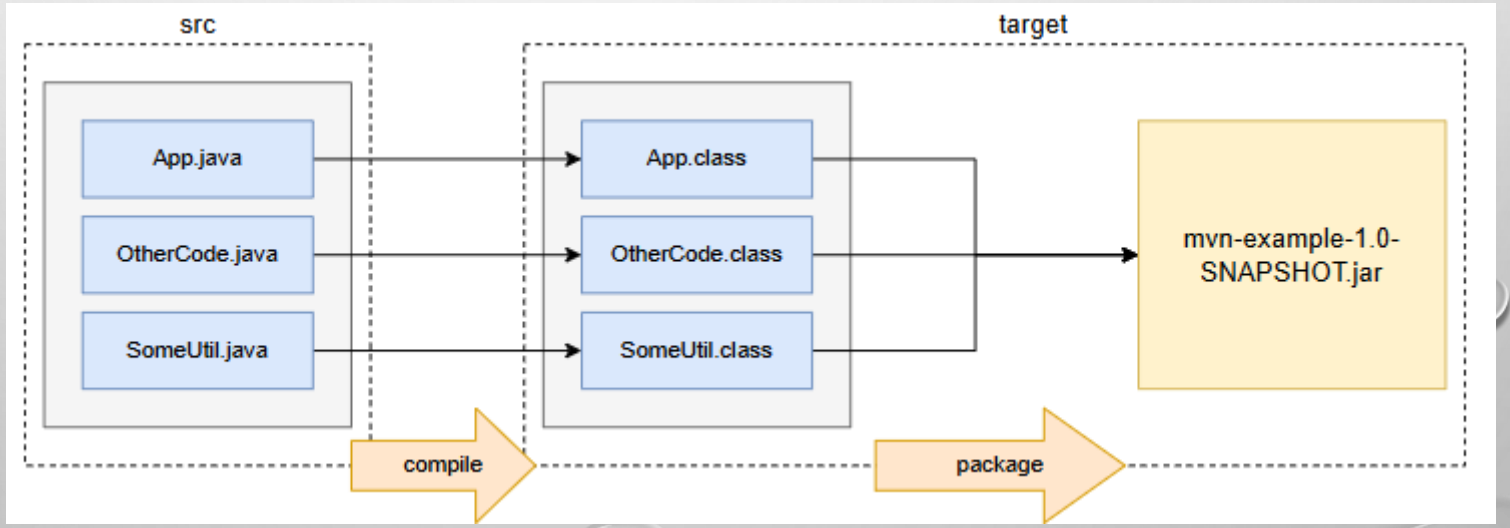
Le cycle de vie de Maven est constitué de différentes phases, chacune représentant une étape de construction d'un projet. Par exemple, dans le cycle de vie "default", on retrouve des phases comme la validation du projet, la compilation, les tests, l'emballage, l'installation et le déploiement. Les phases sont exécutées séquentiellement, et chaque phase peut être associée à un ou plusieurs objectifs de plugins, permettant ainsi d'effectuer des tâches spécifiques lors de la construction. Des commandes telles que mvn clean deploy permettent de nettoyer et déployer les artefacts.



Le cycle de vie de Maven comprend plusieurs phases :

1) **validate** : Vérification de la validité du projet.
2) **compile** : Compilation du code source.
3) **test** : Exécution des tests unitaires.
4) **package** : Création de l'artefact (JAR, WAR, etc.).
5) **install** : Installation de l'artefact dans le répertoire local.
6) **deploy** : Déploiement de l'artefact dans un repository distant.

Chaque phase peut être associée à des objectifs spécifiques, exécutés séquentiellement.



Commandes Maven

Voici les principales commandes de Maven pour gérer le cycle de vie du projet :

mvn validate : Vérifie la validité du projet.
mvn compile : Compile le code source.
mvn test : Exécute les tests unitaires.
mvn package : Crée l'artefact (JAR, WAR, etc.).
mvn install : Installe l'artefact dans le repository local.
mvn deploy : Déploie l'artefact dans un repository distant.
mvn clean : Supprime les fichiers générés lors des constructions précédentes.

Ces commandes permettent d'exécuter des phases spécifiques du cycle de vie.

Il est possible de combiner des commandes pour exécuter plusieurs actions :
mvn clean install : Cela va d'abord nettoyer le projet (supprimer les fichiers précédemment générés), puis installer l'artefact dans le repository local.
mvn clean package deploy : Cela nettoie, empaquette l'artefact et le déploie dans le repository distant.

mvn clean install
à la loupe

La commande **mvn clean install** est utilisée pour nettoyer, compiler, tester et empaqueter un projet Java avec Maven. Elle inclut plusieurs étapes importantes du cycle de vie Maven.

- Nettoyage avec clean

Avant de commencer la compilation, Maven supprime les fichiers générés précédemment dans le dossier target. Cela inclut tous les fichiers compilés et les artefacts de build précédents. Cela permet de s'assurer que le build suivant est propre, sans interférences d'une ancienne compilation.

- Compilation avec compile

Une fois le projet nettoyé, Maven passe à la **compilation** du code source. Il lit tous les fichiers Java dans ton projet et les compile pour générer des fichiers .class dans un dossier spécifique (généralement target/classes). Maven utilise le plugin maven-compiler-plugin pour cette tâche. La configuration par défaut compile tout le code source dans le répertoire src/main/java. Maven utilise un **annotation processor**(outil qui scrute le code source à la recherche d'annotations spécifiques) pour traiter les annotations (Lombok et autre). Le code généré est directement ajouté au bytecode (.class).

- Tests avec test

Une fois la compilation terminée, Maven lance tous les tests définis dans ton projet (généralement dans le répertoire src/test/java). Ces tests peuvent être des tests unitaires (par exemple, avec **JUnit**) ou des tests d'intégration. Vérifier que ton code fonctionne comme prévu. Si des tests échouent, Maven arrête le processus et n'installe pas le JAR.

- Packaging avec package

Après que les tests aient été exécutés avec succès, Maven passe à l'étape de **packaging**, où il crée un fichier JAR (Java ARchive) ou un WAR (Web ARchive) selon la configuration du projet dans le pom.xml.

L'objectif est de créer un fichier exécutable qui peut être partagé, déployé ou installé dans un repository local ou distant.

- Installation avec install

Une fois le JAR (ou WAR) généré, la commande install l'installe dans le **repository local de Maven** (généralement dans ~/.m2/repository), afin qu'il puisse être utilisé par d'autres projets. Un **repository Maven** est un emplacement où Maven stocke des artefacts (comme des JARs, WARs, etc.). Un **repository local** est utilisé pour stocker les artefacts générés sur ta machine. Un **repository distant** est souvent utilisé pour récupérer des dépendances externes.

- Un **JAR (Java ARchive)** est un fichier compressé (en format .zip) qui contient plusieurs fichiers nécessaires à l'exécution d'une application Java. Un JAR peut contenir : Les **fichiers .class**, Les **ressources** nécessaires à l'application (images, fichiers de config..), Un fichier **META-INF/MANIFEST.MF** qui contient des informations sur le contenu du JAR, comme le point d'entrée de l'application (Main-Class), et des informations sur la version du projet.

A RETENIR

Les tests unitaires se concentrent sur l'exécution d'une méthode spécifique pour vérifier son comportement et sa logique interne. En revanche, les tests d'intégration vérifient le bon fonctionnement de plusieurs composants ou services ensemble, en testant une fonctionnalité dans un contexte plus large, incluant les interactions entre différentes parties du système.

La génération de notre Jar est précédée par l'ensemble des tests :

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ SpringShopJpa ---
[INFO] Building jar: C:\Users\El-BabiliM\SpringWorkspace\SpringShopJpa\target\SpringShopJpa-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.6.0:repackage (repackage) @ SpringShopJpa ---
[INFO] Replacing main artifact with repackaged archive
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  11.607 s
[INFO] Finished at: 2022-06-27T14:35:26+02:00
[INFO] ------------------------------------------------------------------------
```

ETAPES SUIVANTES

- → Api/Jwt
- → Scrum
- → Code Clean
- → DevOps (SonarQube, Jenkins & Docker)