

Ateliers

« Mise en pratique des tests avec

JUnit »»

Pré-requis :

- JDK 17+
- IDE (IntelliJIDEA, Eclipse, STS, VSCode)

Support et énoncé des ateliers disponibles à : <https://github.com/dthibau/junit>
Solutions des ateliers disponibles à : <https://github.com/dthibau/junit-solutions>

Table des matières

Atelier 1 : Concepts JUnit.....	2
1.1 : Mise en place du projet.....	2
1.2 : Premières méthodes de test.....	2
1.3 : Approche TDD.....	2
1.4 : Exécution Maven.....	3
Atelier 2 : AssertJ.....	4
2.1: Ajout de dépendance.....	4
2.2: Matchers.....	4
Atelier 3 : Mockito.....	5
3.1: Ajout des dépendances.....	5
3.2: Configuration de Mock.....	5
3.3: Vérifications.....	5
Atelier 4 : Compléments, tests paramétrés.....	6
4.1: Tests paramétrés.....	6
4.2 ParameterResolver.....	6
Atelier 5 : Tests d'intégration avec SpringBoot.....	7
5.1: Mise en place.....	7
5.2 Tests auto-configurés.....	7
Atelier 6 : Spring Cloud Contract.....	8
6.1 Projet Producer.....	8
6.2 Projet consumer.....	8
Atelier 7 : Maven et CI.....	10
7.1 Distinction unitaires et intégration.....	10
7.2 Couverture des tests.....	10
Atelier Optionnel : DBUnit.....	11

Atelier 1 : Concepts JUnit

Objectifs

L'objectif de ce TP est de se familiariser avec les concepts de JUnit. En particulier, les notions de TestCase, de Fixture. Ensuite, l'approche XP est entrevue.

1.1 : Mise en place du projet

Créer un projet Maven et ajouter la dépendance suivante dans le **pom.xml**

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.x</version>
    <scope>test</scope>
</dependency>
```

Récupérer la classe **org.formation.Money**

1.2 : Premières méthodes de test

Nous disposons d'une classe Money qui comprend deux attributs : le montant (*amount*) et la devise (*currency*).

Cette classe contient déjà une méthode permettant d'additionner deux objets de type Money.

Nous proposons d'écrire les TestCase permettant de tester la méthode *add()* et la méthode *equals()*

1. Créer la classe MoneyTest.
2. Implémenter les méthodes vous paraissant adéquates pour tester les différents cas de test

Tester dans l'IDE.

Tester via Maven :

```
mvn test
```

1.3 : Approche TDD

Jusqu'à maintenant, nous avions utilisé des sommes d'argent de même type.

Pour additionner des sommes d'argent de monnaie différente, nous créons un nouvelle classe *MoneyBag* qui représente une collection de sommes d'argent de monnaies différentes.

Les classes *Money* et *MoneyBag* implémentent l'interface **IMoney** qui spécifie une seule méthode :

IMoney add(IMoney money)

Avant d'implémenter la méthode pour ces 2 classes, définir les différents cas de test et les implémenter dans une classe de test.

Réfléchir à la notion de MoneyBag vide

Procéder comme suit :

1. Définir les fixtures nécessaires et les initialiser
2. Écrire les méthodes de tests et s'assurer qu'elles échouent
3. Implémenter la méthode *add()* de la classe Money, ré-exécuter les tests
4. Implémenter la méthode *add()* de la classe MoneyBag et s'assurer que tous les tests passent

1.4 : Exécution Maven

Quelle plugin et quelle version par défaut correspond à la phase maven test ?

Exécuter via Maven et visualiser les résultats.

Créer une suite de test **MoneySuiteTest** :

- Qui affiche un libellé personnalisé
- Qui sélectionne tous les tests précédents

Vous allez avoir besoin de **org.junit.platform:junit-platform-suite** ; pour trouver le bon numéro de version, vous pouvez utiliser le BOM

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.junit</groupId>
            <artifactId>junit-bom</artifactId>
            <version>5.10.2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Réexécuter *mvn test* et visualiser les rapports générés

Atelier 2 : AssertJ

2.1: Ajout de dépendance

Ajouter la dépendance AssertJ dans le pom.xml

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.x</version>
    <scope>test</scope>
</dependency>
```

2.2: Matchers

Réécrire toutes les assertions avec des Matchers. Vous pouvez utiliser :

- Des contraintes sur le type
- Des contraintes sur des propriétés de JavaBean en utilisant la méthode *extracting*
- Faire échouer des tests pour visualiser le reporting

Atelier 3 : Mockito

3.1: Ajout des dépendances

Ajouter les dépendances mockito-core et les extensions JUnit5

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.x</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.x</version>
    <scope>test</scope>
</dependency>
```

3.2: Configuration de Mock

Récupérer l'interface **org.formation.ConvertService** fournie

En utilisant l'interface **ConvertService**, écrire les tests unitaires et l'implémentation de la méthode de MoneyBag

```
public Money convertInto(String destinationCurrency)
```

Cette méthode convertit toutes les monnaies d'un porte monnaie dans une devise donnée.

Effectuer les tests avec :

- Un MoneyBag contenant 2 devises différentes de la devise cible
- Un MoneyBag contenant 2 devises dont une est la devise cible
- Un MoneyBag vide

3.3: Vérifications

Utiliser la méthode **verify()** pour vérifier les appels à *ConvertService* (nombre et arguments)

Atelier 4 : Compléments, tests paramétrés

4.1: Tests paramétrés

Réécrire les tests de la méthode `convertInto` de MoneyBag sous forme de tests paramétrés

4.2 ParameterResolver

Reprendre la classe `org.formation.validator.CurrencyValidator`

Écrire une méthode de test prenant en paramètre une String

Injecter plusieurs valeurs invalides dans la méthode via l'annotation `@RepeatedTest` et l'extension `ParameterResolver`

Effectuer les bonnes assertions

Atelier 5 : Tests d'intégration avec SpringBoot

5.1: Mise en place

Importer le projet Maven fourni. Ce projet utilise la librairie Lombok.

Le projet est une application web développée avec SpringBoot, offrant 3 principaux point d'accès :

- http://localhost:8080 : Application web
- http://localhost:8080/swagger-ui.html : Documentation REST API
- http://localhost:8080/actuator : Urls de monitoring

L'application est sécurisée (Sécurité stateful pour l'application Web ou stateless/JWT pour l'API Rest). Les utilisateurs sont définis dans le fichiers *src/main/resources/users.csv*

L'application utilise SpringData pour accéder à une base de données relationnelles.

L'implémentation utilisée est une base embarquée h2. Dans ce cas, SpringBoot configure automatiquement Hibernate afin qu'il crée et initialise la base à chaque démarrage de l'application.

5.2 Tests auto-configurés

A chaque mise en place des tests, il est conseillé de se faire injecter le contexte applicatif Spring (ApplicationContext) afin de pouvoir inspecter les beans configurés.

@DataJpaTest

Ecrire une classe de test vérifiant le bon fonctionnement de la méthode

```
Optional<Fournisseur> findByReference(String reference);
```

@JsonTest

Ecrire une classe de test vérifiant le bon fonctionnement de la sérialisation/désérialisation de la classe Produit

@WebMVCTest

Utiliser **@WebMVCTest** pour tester une méthode de ProduitRestController en utilisant un mockMVC

Utiliser les annotations **@WithMockUser** pour contourner la sécurité

Écrire un test d'intégration complet

Atelier 6 : Spring Cloud Contract

Reprise de <https://www.baeldung.com/spring-cloud-contract>

Récupérer les 2 projets **producer** et **consumer**, vérifier qu'ils démarrent correctement et regarder les classes contrôleur.

6.1 Projet Producer

Dans le projet *producer*,

Ajouter les dépendances suivantes :

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <version>4.0.4</version>
    <scope>test</scope>
</dependency>
```

Et configurer le plugin :

```
<plugin>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-contract-maven-plugin</artifactId>
<version>4.0.4</version>
<extensions>true</extensions>
<configuration>
<baseClassForTests>
    org.formation.BaseTestClass
</baseClassForTests>
</configuration>
</plugin>
```

- Récupérer le contrat et la classe de base.
- Exécuter les tests
- Installer le projet producer

6.2 Projet consumer

Dans le projet consumer

Ajouter les dépendances suivantes :

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-wiremock</artifactId>
    <version>4.0.4</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-stub-runner</artifactId>
    <version>4.0.4</version>
    <scope>test</scope>
</dependency>
```

Et exécuter la classe de test fournie

Atelier 7 : Maven et CI

7.1 Distinction unitaires et intégration

Sur le projet précédent, mettre en place une configuration Maven afin que les tests de la couche controller soit effectué dans la phase d'intégration.

7.2 Couverture des tests

Mettre en place JaCoCo et visualiser les rapports

Atelier Optionnel : DBUnit

Créer un projet Maven avec les dépendances sur h2 et DBUnit :

```
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.7.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.200</version>
    <scope>test</scope>
</dependency>
```

- Visualisez le script de création de schéma fourni : schema.sql
- Mettre en place un jeu de données remplissant les tables fournisseur et produit
- Mettre en place une première méthode de test qui vérifie qu'à l'exécution la table fournisseur est bien identique à celle du dataset original.
- Écrire une méthode de test qui charge un DataSet attendu, insère un nouveau fournisseur en appelant la classe fournie FournisseurDao et vérifie que le dataset réel est bien identique au dataset attendu