LO54 SEMESTRE A19

JUnit Backend : Mockito

Auteurs :
Thomas GUBIEN
Pierre NEAU
Aurélien SAUNIER

Encadrants :
Olivier RICHARD
Karen GÁRATE ESCAMILLA

AUTOMNE 2019

Table des matières

Présentation du projet

Contexte

Le projet s'articule autour de la gestion d'offre de formation d'une école privée. Celle-ci devra être en mesure de gérer les différentes formations qui sont proposées sous son nom sous une plateforme accessible par internet.

Dans ce cadre, un utilisateur de la plateforme pourra s'inscrire à une formation l'intéressant en fournissant ses coordonnées personnelles (Prénom, Nom, Adresse, Téléphone, Email). Il aura pour ce faire la possibilité de filtrer ces dernières selon différents critères (Titre, date, lieu…).

Structure du projet

L'architecture de ce projet se reposera sur Hibernate et Maven. Les technologies MongoDB et Mockito seront utilisées.

Hibernate pour transposer les classes JAVA en base de données et vice-versa. Pour ce faire, l'utilisation d'annotation simplifie la tâche et permet de ne pas alourdir le projet avec des fichiers .xml supplémentaires.

MongoDB est utilisé dans le but de stocker les inscriptions des utilisateurs aux formations.

Mockito est utilisé pour la réalisation de tests unitaires backend. Nous allons voir cette technologie et l'implémentation que nous en avons fait plus en détail.

Mockito

L'Installation

Grâce à notre utilisation de Maven, l'installation de Mockito est relativement simple :

- Il suffit de rajouter la dépendance Maven correspondant à la version voulut de Mockito dans le fichier pom.xml à la racine du projet (versions trouvables ici : https://mvnrepository.com/artifact/org.mockito).

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.2.0</version>
    <scope>test</scope>
</dependency>
```

On remarquera ici l'utilisation de « <scope>test</scope> » qui permet de définir que cette dépendance n'est à utilisé que lors de la phase de test et ne doit pas faire partie du build final.

L'utilisation de Mockito

Pour pouvoir utiliser Mockito, il faut maintenant importer les bibliothèques de test que l'on va utiliser pour notre test comme dans l'exemple ci-dessous :

```
import org.mockito.Mock;
import static org.mockito.Mockito.*;
import org.mockito.MockitoAnnotations;
```

Mockito propose plusieurs outils pour tester nos différentes classes. On retrouve notamment les Mock, les Spy et les InjectMocks.

Voici plus en détails leurs spécificités :

- **Mock :** Permet la création d'un objet « mocké » (qui est en réalité une interface de l'objet). L'utilisation des méthodes de cet objet ne retournera que des valeurs nulles (selon le type du retour). Pour obtenir une autre valeur de retour, il faudra utiliser le mécanisme de « stubb » que nous détaillerons un peu plus tard.
- **Spy :** Contrairement au Mock, Spy permet d'instancier l'objet mocké. Cela signifie qu'en l'absence de stubbing, les méthodes réelles de l'objet seront appelées.

- **<u>InjectMocks :</u>** Permet de créer une instance de l'objet et lui injecte les Mock (ou Spy) qui lui serviront de dépendences.

Pour utiliser ce que nous venons de voir au-dessus, il y a deux méthodes :

- Par appel de la classe : User user = *mock(User.class) ;*
- Par annotation : @Mock (@Spy, @InjectMocks…)

Dans le cadre du projet nous avons utilisé les annotations pour un soucis de lisibilité et pour simplifier le code.

Toutefois l'utilisation d'annotation ne suffit pas en elle-même, il faut préciser à Mockito d'initialiser et d'utiliser les objets mockés. On retrouve là aussi deux méthodes :

- ```
  @RunWith(MockitoJunitRunner.class)
  public class MyTestClass {
  }
  ```
- ```
  @Before
  public void init() {
          MockitoAnnotations.initMocks(this);
  }
  ```

C'est la deuxième méthode que nous avons utilisé, principalement car nous avons une utilité supplémentaire à « init() ».

Voici un exemple pour illustrer :

```
public class MYSQLCourseDAOTest {

    @Mock
    MYSQLCourseDAO mysqlCourseDAOMocked = new MYSQLCourseDAO();

    Course c = new Course();

    @Before
    public void init() {
        MockitoAnnotations.initMocks(this);
        when(mysqlCourseDAOMocked.saveRecord(c)).thenReturn("CODE");
    }


    /**
     * Test of saveRecord method, of class MYSQLCourseDAO.
     */
    @Test
    public void testSaveRecord() {
       System.out.println("saveRecord");

       c.setCode("CODE");
       c.setTitle("Course");

      String s =  mysqlCourseDAOMocked.saveRecord(c);
      verify(mysqlCourseDAOMocked).saveRecord(c);
      assertEquals(s,c.getCode());
    }
```

Nous retrouvons bien ici l'annotation « @Mock » ainsi que l'utilisation de « MockitoAnnotations.initMocks(this); »

On remarque cependant la présence d'un « when(…).thenReturn(…) ; ». Ceci est le « stubb » dont nous avons parlé précédemment. Il permet de définir une valeur de retour pour une fonction appelée. Dans notre exemple ci-dessous la fonction saveRecord(c) de mysqlCourseDAOMocked retournera « CODE ».

En détaillant un peu plus cet exemple, on retrouve l'annotation @Test qui permet de définir la fonction annotée comme fonction de test. On retrouve aussi les fonctions verify() et assertEquals().

La première permet de vérifier dans notre cas qu'il y a bien un appel de la fonction saveRecord(c). La seconde permet de vérifier le code de retour avec l'information renseigné dans l'objet Course « c » (Cette valeur de retour est celle défini par « when(…).thenReturn(…) ; »).

L’utilisation des diverses fonctions asserts se fait par ailleurs grâce à la librairie JUnit. Ils permettent d’effectuer de nombreuses vérifications.

Notre expérience avec Mockito

Dans le groupe, nous connaissions déjà tous les tests unitaires. Nous n'avions cependant jamais utilisé de librairie tierce comme Mockito.

Si on parle purement de la technologie en elle-même, la compréhension de celle-ci n'a pas posé de grandes difficultés. Il faut toutefois préciser que nous n'avons ni vu ni utilisé l'entièreté de Mockito. Nous pouvons même dire que nous avons seulement utilisé qu'une petite partie de ce que qui est possible avec.

Dans la pratique toutefois, nous nous sommes heurtés à une grosse difficulté, que nous n'avons pas pu surmonter. En effet, pour tester efficacement repository, nous soutions pouvoir utiliser @InjectMocks pour tester pleinement l'objet. Or, dans ce cas-ci, nous nous retrouvions avec une erreur insolvable, dont nous ne comprenions pas l'origine.

Même après concertation avec monsieur Olivier RICHARD, nous n'avons pas pu résoudre cette erreur.

Nous en avons de ce fait été réduits à la simple utilisation de @Mock et de stubb ce qui a grandement limité la quantité et précision des tests réalisables.

Malgré ce problème, ce fut une bonne expérience car cela nous a permis d'acquérir de nouvelles connaissances.