Cours 5 : Tests

Mardi 1 Décembre 2020

Antoine Flotte (aflotte@excilys.com)

Excilys
Développeurs de passion

SOMMAIRE

Quel Type ?

Test intégration :

On teste que les différents modules fonctionnent bien entre eux quand ils s'appellent les uns les autres.

Par exemple :

A appelle B avec un fichier C en paramètre. Il s'agit de vérifier que C est dans le même format dans A et B.

Test unitaire :

On teste une classe de façon unitaire, tout les appels à un élément externe est simulé

Outils : JUnit, Mockito, potentiellement PowerMockito

Test de non régression :

On vérifie que le comportement n’évolue pas, permet d’éviter des effets de bord.

Exemple :

Si on code quelque chose d’immuable comme la gravité.

Quel Type ? de charge

Test de charge :

On simule la présence d'utilisateur avec des appels très nombreux, et on vérifie que le site ne crash pas, et que les temps d'exécution restent acceptables

Outils : Gatling

Test end to end :

On vérifie le comportement du projet suite à des appels le parcourant dans son entièreté.

Par Exemple : on fait des appels REST sur les get/post/… du back

Outils : Cucumber, Postman

Généralités

Coverage

Coverage = nb ligne passé par les tests / nb de ligne total exprimé en %

En général on essaie au minimum d’avoir 80% de coverage

Attention, 80% de coverage ne signifie pas que vous avez testé 80% des cas possibles par vos fonctions.

Il suffit d’avoir une ligne dans un else et des initialisations et vous aurez un coverage elevé sans tester les else (donc 50% des cas)

Lancer les tests

mvn test

Quand vous faites les mvn clean install les tests se lancent.

Ce n’est pas le cas avec mvn spring-boot:run

Pour ne pas les lancer, rajouter dans votre commande :

-DskipTests

Quoi tester ?

On ne teste pas les modèles, Pojo, DTO. Il n’y a pas d’intelligence dedans

On ne teste pas les Mapper (fonction static, en vrai on peut les tester avec PowerMockito)

Mock

On utilise des Mocks. Il s’agit d’objet qui vont remplacer vos appels aux classes externes. Il existe aussi des Spy. Par default les Mocks ont des fonctions vide alors que les Spy on le même comportement.

Importance

Code sale et peu documenté = difficile de rajouter des tests dans le futur

Code avec peu de tests = les gens évitent de toucher cette partie du code

Code peu touché = code destiné à être refais complétement par les équipes futurs

Les tests peuvent aussi servir de cahier des charges de vos fonctions. C’est donc aussi une façon indirecte de documenter vos fonctions !

TDD = Test Driven Development

Tests Java

Dépendances

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

Et on fait les tests dans le package test.java.com

On crée les fonctions de tests avec l'annotation @Test

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.15.0</version>
</dependency>
```

Dépendances

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.15.0</version>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-inline</artifactId>
    <version>2.15.0</version>
</dependency>
```

On utilise Mockito :

Sur la classe de test on met l'annotation :

@RunWith(MockitoJUnitRunner.class)

Sur l'instance de la classe que l'on teste :

@InjectMocks

Sur les classes appelés par cette instance :

@Mock

Au lieu d'utiliser RunWith on peut aussi dans la classe de test utiliser :

```
@Before
    public void init() {
        MockitoAnnotations.initMocks(this);
    }
```

Exemples de Mock

```java
package com.service;
import com.ensta.myfilmlist.dao.FilmDAO;
import com.ensta.myfilmlist.service.FilmService;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class FilmServiceTest {
    @InjectMocks
    FilmService filmService;

    @Mock
    FilmDAO filmDAO;
}
```

Mock

On ne mock pas les fonctions statiques (on peut avec PowerMockito) ni les fonctions privées (que l'on ne teste pas non plus)

On peut maintenant utiliser :

```
when(filmDAO.findAll()).thenReturn(list);
```

Où list aura été crée auparavant.

Avec SpringBoot on @Autowired au lieu de @InjectMocks, au lieu d'utiliser @Mock on utilise @MockBean

Ensuite sur la classe Test on rajoute au début :

```
@RunWith(SpringRunner.class)
```

```
@SpringBootTest
```

À la place de

```
@RunWith(MockitoJUnitRunner.class)
```

Tests Java

Exemple

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class FilmServiceTest {
    @Autowired
    FilmService filmService;
    @MockBean
    FilmDAO filmDAO;


    }
```

Tests Java

Exemple

```
@Test
    public void findAllTest()throws DaoException, ServiceException    {
        List<FilmPojo> listFilm = new ArrayList<>();
        FilmPojo film1 = new FilmPojo("1", 1);
        List<FilmDTO> listFilmResult = new ArrayList<>();
        FilmDTO filmResult1 = new FilmDTO("1", 1);
        listFilmResult.add(filmResult1);
        listFilm.add(film1);
        when(filmDAO.findAll()).thenReturn(listFilm);
        List<FilmDTO> resultList = filmService.findAll();
        assertEquals(1, resultList.size());
        assertEquals(filmResult1, resultList.get(0)));
    }
```

Ensuite vous appelez la méthode que vous testez avec ses arguments.

Vous pourrez ensuite tester :

assertEquals(expected,result); sur la size du résultat ou en parcourant la liste pour vérifier que c'est bien celle attendu

Vous pouvez aussi tester les DAO en faisant des tests de non régression. On fixe la base de donnée et on vérifie qu'on a toujours le même resultat.

Vous pourrez avoir besoin de :

@Before de JUnit pour exécuter une fonction d'initialisation de la base de donnée.

Controller

Pour vos controllers (ou web services), les tests unitaires ne sont pas suffisant (et pas forcément utile pour le projet du cours).

Par contre, vous pouvez mettre en place des tests end to end.

Vous appelez juste la fonction à tester, sans rien mocker derrière. Toutes les couches seront alors parcouru et vous saurez que les cas d’usages sont respecté vu de l'extérieur.