Test Unitaires

Niveau Repository

En pratique les Tests unitaires au niveau Repository (DAO) utilisent une surcouche de JUnit.
Il y a de nombreuses façons de faire.
Voici la plus courante :

Il s'agit d'employer les annotations suivantes :

- @RunWith(SpringRunner.**class**) : Indique qu'on utilise la surclasse de Spring
- @DataJpaTest : indique qu'on travaille sur les composants JPA et qu'on utilise la base en mémoire H2.
- @SpringBootTest(classes=SmallApp.**class**) : Indique qu'on utilise le système Spring dans son ensemble. Dans ce cas, il faut créer une application minimale au même niveau que les classes de test. Exemple :

```
@SpringBootApplication
public class SmallApp {
	public static void main(String[] args) {
		SpringApplication.run(SmallApp.class, args);
	}
}
```

Ça donne donc pour la classe de test :

```
@RunWith(SpringRunner.class)
@DataJpaTest
@SpringBootTest(classes=SmallApp.class)
public class SpecieTest {
```

On remarque que la classe de test comporte toujours « Test » dans son nom, sans quoi Maven ne peut pas s'y retrouver lorsqu'on lance les Tests Unitaires via Maven.

De plus :

- Les fonctions de test de la classe ne sont pas obligées de comporter « test » dans leur nom. Il faut juste les annoter avec *@Test*
- *setUp()* et *tearDown()* n'existent plus. Il faut juste créer des fonctions void public annotées respectivement avec *@Before* et *@After*
- Avec H2, inutile de créer un fichier *application.properties*. Toutefois, il arrive qu'on ait des erreurs bizarres comme quoi les tables ne sont pas créées dans H2. Dans ce cas, créer un *application.properties* dans *src/test/resources*, avec dedans :
    - `spring.jpa.generate-ddl=true`
    - `spring.jpa.hibernate.ddl-auto=create`

Pour charger la base H2, on le fait comme d'habitude dans le *setUp()*, ici dans la fonction annotées avec *@Before*. Pour charger la base, on pourrait utiliser la fonction *save()* du repository, mais comme on est censé tester le repository, c'est un peu gênant. On va donc utiliser le *TestEntityManager* qu'on récupère via un autowire. Comme ça :

```
@Autowired
private TestEntityManager entityManager;
```

Donc dans la fonction *@Before*, on crée des Objets dans la base via *entityManager.persist(<T>)*. Et le tour est joué.
A noter que si l'on charge la base via la fonction *@Before*, les id des différents champs seront

incrémentés à chaque lancement de la fonction *@Before* (donc avant chaque fonction de test). On peut forcer la recréation de la base H2 à chaque appel de la fonction *@Before* (et donc on repart avec les mêmes id) en utilisant l'annotation *@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)* au niveau de la classe de test.

A noter aussi qu'on peut employer l'annotation *@BeforeClass* qui, elle, est appelée une seule fois lors du chargement de la classe de test. Suivant le type de before, les tests unitaires seront différents, bien y réfléchir, donc.

Après, il suffit de créer les fonctions de test standard (list, get, update, delete, create) et de les tester via *Run as>JUnit*.
Puis de vérifier que les tests se lancent aussi correctement via Maven.

A noter que généralement on ne teste pas les fonctions générées automatiquement via le CrudRepository, mais seulement celles qu'on rajoute en plus (celles codées en JQL, typiquement).

A noter aussi que, comme on utilise H2, il faut la déclarer en dépendance dans le pom.xml en scope test :

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

Niveau Service

Là, c'est plus compliqué. On pourrait se dire, pas de problème, on teste le service en appelant le (ou les) repository associés et tout va bien. Sauf que …

Sauf que, on peut avoir à tester les services sans avoir à utiliser les repository pour diverses raisons. En particulier, on peut avoir à tester les services sans que les repository soient déjà écrits.

Comment fait-on ?

En pratique, on utilise ce qu'on appelle des mocks, c.a.d des trucs qui font semblant d'être autre chose, ici les repository.

Intuitivement, c'est assez simple à comprendre : imaginons qu'on ait un repository avec une fonction list() qui renvoie le contenu d'une table. Si on n'a pas le repository, comment va-t-on faire ? Eh bien, on va coder une classe bidon qui renvoie ce qu'est censé renvoyé la fonction list() soit une List<?> (créée en mémoire). C'est ça un mock. Un truc qui simule autre chose.

Evidemment, on ne va pas, pour chaque service à tester, créer un mock particulier, mais plutôt utiliser des mocks génériques. Et pour cela, on utilise un produit qui s'appelle Mockito. Mockito permet d'utiliser des mocks génériques et aussi d'utiliser JUnit 4 comme dans les tests de repository.

```
On indique qu'on utilise Mockito
@RunWith(MockitoJUnitRunner.class)
public class ServiceTest {
        Ici on indique qu'on a un mock qui simule le repository des Specie
        @Mock
        SpecieRepository repoMock;
```

```
        Ici, on indique que le mock est injecté/utilisé par le service à tester (ici SpecieService)
        @InjectMocks
        SpecieService service;
```

L'idée générale est qu'on va utiliser le mock comme si c'était un repository.

Supposons qu'on veuille tester la fonction *list()* du service qui fait appel à *findAll()* du repository. En vrai, on va appeler la fonction *findAll()* du mock. Laquelle ne fait rien pour le moment. On va donc indiquer ce que renvoie le *findAll()* du mock.

```
@Test
public void listTest() {
        when(repoMock.findAll()).thenReturn(list);
        assertEquals(2, service.list().size());
}
```

La première ligne indique que quand on va appeler la fonction *findAll()* du mock (qui simule le repository, donc), on va renvoyer une List<> que l'on a préalablement chargée avec 2 Specie (par exemple dans le *@Before*)

Ensuite on teste que la fonction *list()* du service renvoie bien 2 éléments.

Tout cela, c'est bien sympa, mais que fait-on dans le cas où la fonction du repository est void (ne renvoie rien) ?

Dans ce cas, on utilise une Answer, c'est-à-dire qu'on va faire une opération particulière.

```
public void create() {
                Là, on crée une Answer qui ajoute un objet Specie dans la List<?> utilisée dans
l'exemple précédent
                doAnswer(new Answer<Void>(){
                        @Override
                        public Void answer(InvocationOnMock invocation) throws Throwable {
                                Object[] os = invocation.getArguments();
                                if (os != null && os.length != 0 && os[0] != null) {
                                        Specie sp = (Specie)os[0];
                                        sp.setId(12L);
                                        list.add(sp);
                                }
                                return null;
                        }
                Là, on dit que quand on appelle la fonction save() du mock (du repository) avec
n'importe quelle valeur en paramètre, on fera l'opération définie plus haut dans le Answer
                }).when(repoMock).save(Mockito.any(Specie.class)); →
                Specie sp = new Specie();
                sp.setCommonName("AA");
                sp.setLatinName("BB");
                service.create(sp);
                assertEquals(3, list.size());
                assertEquals(12L, list.get(2).getId().longValue());
        }
```

L'idée ici, c'est qu'on teste la fonction *create()* du service qui appelle la fonction *save()* du mock (du repository dans l'idée). Il faut donc dire ce que cette fonction *save()* du mock va faire. On crée donc une Answer qui ajoute toujours la même valeur dans la List<?> vue précédemment.

Après le *create()* du service, on a donc 3 éléments dans la List<?>, et on teste qu'on a bien 3 éléments.

Je conçois qu'ici les exemples sont tellement triviaux qu'on ne voit pas très bien l'intérêt du mock (et même l'intérêt de tester les services tout court). Mais ne perdons pas de vue que dans le vrai monde,

les services sont beaucoup plus complexes et peuvent faire appel à plusieurs repository (et même à d'autres choses).

Mockito permet de faire plein d'autres choses encore plus intéressantes. Je vous renvoie à ce petit tutoriel : https://javacodehouse.com/blog/mockito-tutorial/