Introduction aux tests avec Spock et Groovy

1. Introduction

Dans cette partie, nous allons jeter un œil à Spock , un framework de test Groovy . Principalement, Spock vise à être une alternative plus puissante à la pile JUnit traditionnelle, en tirant parti des fonctionnalités de Groovy.

Groovy est un langage basé sur JVM qui s'intègre parfaitement à Java. En plus de l'interopérabilité, il offre des concepts de langage supplémentaires tels que la dynamique, les types optionnels et la méta-programmation.

En utilisant Groovy, Spock introduit de nouvelles façons expressives de tester nos applications Java, ce qui n'est tout simplement pas possible dans le code Java ordinaire. Nous explorerons certains des concepts de haut niveau de Spock au cours de cet article, avec quelques exemples pratiques étape par étape.

2. Dépendance Maven

Avant de commencer, ajoutons nos dépendances Maven :

```
<dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-core</artifactId>
    <version>1.0-groovy-2.4</version>
```

```
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-all</artifactId>
    <version>2.4.7</version>
    <scope>test</scope>
</dependency>
```

Nous avons ajouté Spock et Groovy comme nous le ferions pour n'importe quelle bibliothèque standard. Cependant, comme Groovy est un nouveau langage JVM, nous devons inclure le plugin *gmavenplus* afin de pouvoir le compiler et l'exécuter :

```
<plugin>
    <groupId>org.codehaus.gmavenplus</groupId>
    <artifactId>gmavenplus-plugin</artifactId>
    <version>1.5</version>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>testCompile</goal>
            </goals>
        </execution>
     </executions>
</plugin>
```

Nous sommes maintenant prêts à écrire notre premier test Spock, qui sera écrit en code Groovy. Notez que nous utilisons Groovy et Spock uniquement à des fins de test et c'est pourquoi ces dépendances sont à portée de test.

3. Structure d'un test de Spock

3.1. Spécifications et fonctionnalités

Comme nous écrivons nos tests dans Groovy, nous devons les ajouter au répertoire *src/test/groovy* , au lieu de *src/test/java.* Créons notre premier test dans ce répertoire, en le nommant *Specification.groovy :*

```
class FirstSpecification extends Specification {

}
```

Notez que nous étendons l' interface de *spécification .* Chaque classe Spock doit étendre ceci afin de rendre le framework disponible. C'est ce qui nous permet d'implémenter notre première *fonctionnalité :*

```
def "one plus one should equal two"() {
  expect:
  1 + 1 == 2
}
```

Avant d'expliquer le code, il convient également de noter que dans Spock, ce que nous appelons une *fonctionnalité* est en quelque sorte synonyme de ce que nous considérons comme un *test* dans JUnit. Ainsi, **chaque fois que nous faisons référence à une *fonctionnalité,* nous faisons en fait référence à un *test.***

Maintenant, analysons notre *fonctionnalité* . Ce faisant, nous devrions immédiatement être en mesure de voir certaines différences entre lui et Java.

La première différence est que le nom de la méthode de fonctionnalité est écrit sous la forme d'une chaîne ordinaire. Dans JUnit, nous aurions eu un nom de méthode qui utilise camelcase ou des traits de soulignement pour séparer les mots, ce qui n'aurait pas été aussi expressif ou lisible par l'homme.

Ensuite, notre code de test vit dans un bloc *expect* . Nous aborderons les blocs plus en détail sous peu, mais ils constituent essentiellement une manière logique de diviser les différentes étapes de nos tests.

Enfin, on se rend compte qu'il n'y a pas d'affirmations. C'est parce que l'assertion est implicite, passant lorsque notre déclaration est égale à *true* et échouant lorsqu'elle est égale à *false* . Encore une fois, nous couvrirons les affirmations plus en détail sous peu.

3.2. Blocs

Parfois, lors de l'écriture d'un test JUnit, nous pouvons remarquer qu'il n'y a pas de moyen expressif de le diviser en parties. Par exemple, si nous suivions le développement piloté par le comportement, nous pourrions finir par désigner les parties *données quand puis* en utilisant des commentaires :

```
@Test
public void givenTwoAndTwo_whenAdding_thenResultIsFour() {
   // Given
   int first = 2;
   int second = 4;

   // When
   int result = 2 + 2;

   // Then
   assertTrue(result == 4)
}
```

Spock résout ce problème avec des blocs. **Les blocs sont une manière native de Spock de décomposer les phases de notre test à l'aide d'étiquettes.** Ils nous donnent des étiquettes pour *donné quand alors* et plus :

1. *Configuration* (Aliased by Given) - Ici, nous effectuons toute configuration nécessaire avant l'exécution d'un test. Il s'agit d'un bloc implicite, avec du code qui n'est dans aucun bloc et qui en fait partie

2. *Quand* - C'est là que nous fournissons un *stimulus* à ce qui est testé. En d'autres termes, où nous invoquons notre méthode sous test
3. *Alors* - C'est là que les affirmations appartiennent. Dans Spock, celles-ci sont évaluées comme des assertions booléennes simples, qui seront couvertes plus tard
4. *Attendre* - C'est une façon d'exécuter notre *stimulus* et notre *affirmation* dans le même bloc. Selon ce que nous trouvons plus expressif, nous pouvons ou non choisir d'utiliser ce bloc
5. *Nettoyage* - Ici, nous supprimons toutes les ressources de dépendance de test qui seraient autrement laissées pour compte. Par exemple, nous pouvons vouloir supprimer tous les fichiers du système de fichiers ou supprimer les données de test écrites dans une base de données

Essayons à nouveau d'implémenter notre test, cette fois en utilisant pleinement les blocs :

```
def "two plus two should equal four"() {
    given:
        int left = 2
        int right = 2

    when:
        int result = left + right

    then:
        result == 4
}
```

Comme nous pouvons le voir, les blocs aident notre test à devenir plus lisible.

3.3. Tirer parti des fonctionnalités Groovy pour les assertions

Dans les blocs *then* et *expect* , les assertions sont implicites .

La plupart du temps, chaque instruction est évaluée puis échoue si elle n'est pas *true* . Lorsqu'il est associé à diverses fonctionnalités de Groovy, il

élimine efficacement le besoin d'une bibliothèque d'assertions. Essayons une assertion de *liste* pour démontrer ceci :

```
def "Should be able to remove from list"() {
    given:
        def list = [1, 2, 3, 4]

    when:
        list.remove(0)

    then:
        list == [2, 3, 4]
}
```

Bien que nous ne parlions que brièvement de Groovy dans cet article, cela vaut la peine d'expliquer ce qui se passe ici.

Premièrement, Groovy nous donne des moyens plus simples de créer des listes. Nous pouvons simplement déclarer nos éléments entre crochets, et en interne une *liste* sera instanciée.

Deuxièmement, comme Groovy est dynamique, nous pouvons utiliser *def* , ce qui signifie simplement que nous ne déclarons pas de type pour nos variables.

Enfin, dans le cadre de la simplification de notre test, la fonctionnalité la plus utile démontrée est la surcharge des opérateurs. Cela signifie qu'en interne, plutôt que de faire une comparaison de référence comme en Java, la méthode *equals()* sera invoquée pour comparer les deux listes.

Cela vaut également la peine de démontrer ce qui se passe lorsque notre test échoue. Faisons-le casser, puis voyons ce qui est sorti sur la console :

```
Condition not satisfied:

list == [1, 3, 4]
|    |
|    false
[2, 3, 4]
```

```
 <Click to see difference>

at FirstSpecification.Should be able to remove from list(FirstSpecification.groovy:30)
```

Alors que tout ce qui se passe est d'appeler *equals()* sur deux listes, Spock est assez intelligent pour effectuer une décomposition de l'assertion défaillante, nous donnant des informations utiles pour le débogage.

3.4. Faire valoir des exceptions

Spock nous fournit également un moyen expressif de vérifier les exceptions. Dans JUnit, certaines de nos options peuvent utiliser un bloc *try-catch* , déclarer *attendu* en haut de notre test ou utiliser une bibliothèque tierce. Les assertions natives de Spock permettent de gérer les exceptions prêtes à l'emploi :

```
def "Should get an index out of bounds when removing a non-existent item"() {
    given:
        def list = [1, 2, 3, 4]

    when:
        list.remove(20)

    then:
        thrown(IndexOutOfBoundsException)
        list.size() == 4
}
```

Ici, nous n'avons pas eu à introduire une bibliothèque supplémentaire. Un autre avantage est que la méthode *throw()* affirmera le type de l'exception, mais n'arrêtera pas l'exécution du test.

4. Tests basés sur les données

4.1. Qu'est-ce qu'un test basé sur les données ?

Essentiellement, **les tests basés sur les données consistent à tester plusieurs fois le même comportement avec différents paramètres et assertions** . Un exemple classique serait de tester une opération mathématique telle que la quadrature d'un nombre. Selon les différentes permutations d'opérandes, le résultat sera différent. En Java, le terme que nous connaissons peut-être mieux est le test paramétré.

4.2. Implémentation d'un test paramétré en Java

Dans certains contextes, il vaut la peine d'implémenter un test paramétré à l'aide de JUnit :

```
@RunWith(Parameterized.class)
public class FibonacciTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
          { 1, 1 }, { 2, 4 }, { 3, 9 }
        });
    }

    private int input;

    private int expected;

    public FibonacciTest (int input, int expected) {
        this.input = input;
        this.expected = expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Math.pow(3, 2));
    }
}
```

Comme nous pouvons le voir, il y a beaucoup de verbosité et le code n'est pas très lisible. Nous avons dû créer un tableau d'objets à deux dimensions qui vit en dehors du test, et même un objet wrapper pour injecter les différentes valeurs de test.

4.3. Utiliser des tables de données dans Spock

Une victoire facile pour Spock par rapport à JUnit est la façon dont il implémente proprement des tests paramétrés. Encore une fois, dans Spock, cela s'appelle **Data Driven Testing.** Maintenant, implémentons à nouveau le même test, mais cette fois nous utiliserons Spock avec **Data Tables** , qui fournit un moyen beaucoup plus pratique d'effectuer un test paramétré :

```
def "numbers to the power of two"(int a, int b, int c) {
  expect:
      Math.pow(a, b) == c

  where:
      a | b | c
      1 | 2 | 1
      2 | 2 | 4
      3 | 2 | 9
  }
```

Comme nous pouvons le voir, nous avons juste une table de données simple et expressive contenant tous nos paramètres.

De plus, il appartient à ce qu'il devrait faire, parallèlement au test, et il n'y a pas de passe-partout. Le test est expressif, avec un nom lisible par l'homme, et un bloc *expect* et *where* pur pour décomposer les sections logiques.

4.4. Lorsqu'une table de données échoue

Cela vaut également la peine de voir ce qui se passe lorsque notre test échoue :

```
Condition not satisfied:

Math.pow(a, b) == c
     |   |  |  |  |
     4.0 2  2  |  1
               false

Expected :1

Actual   :4.0
```

Encore une fois, Spock nous donne un message d'erreur très informatif. Nous pouvons voir exactement quelle ligne de notre Datatable a causé un échec et pourquoi.

5. Mock

5.1. Qu'est-ce que le mock ?

Le mock est un moyen de modifier le comportement d'une classe avec laquelle notre service testé collabore. C'est un moyen utile de pouvoir tester la logique métier indépendamment de ses dépendances.

Un exemple classique de ceci serait de remplacer une classe qui fait un appel réseau par quelque chose qui fait simplement semblant de le faire. Pour une explication plus approfondie, cela vaut la peine de lire cet article .

5.2. les mock de Spock

Spock a son propre cadre de simulation, utilisant des concepts intéressants apportés à la JVM par Groovy. Tout d'abord, instancions un *Mock :*

```
PaymentGateway paymentGateway = Mock()
```

Dans ce cas, le type de notre maquette est déduit par le type de variable. Comme Groovy est un langage dynamique, nous pouvons également fournir un argument de type, nous permettant de ne pas avoir à affecter notre mock à un type particulier :

```
def paymentGateway = Mock(PaymentGateway)
```

Maintenant, chaque fois que nous appelons une méthode sur notre mock *PaymentGateway ,* une réponse par défaut sera donnée, sans qu'une instance réelle soit invoquée :

```
when:
    def result = paymentGateway.makePayment(12.99)

then:
    result == false
```

Le terme pour cela est l*e mock indulgente* . Cela signifie que les méthodes factices qui n'ont pas été définies renverront des valeurs par défaut raisonnables, au lieu de lever une exception. C'est par conception dans Spock, afin de rendre les simulations et donc les tests moins cassants.

5.3. Appels de méthode de substitution sur les *simulations*

Nous pouvons également configurer des méthodes appelées sur notre mock pour répondre d'une certaine manière à différents arguments. Essayons de faire en sorte que notre simulation *PaymentGateway* renvoie *true* lorsque nous effectuons un paiement de *20 :*

```
given:
    paymentGateway.makePayment(20) >> true

when:
    def result = paymentGateway.makePayment(20)

then:
    result == true
```

Ce qui est intéressant ici, c'est comment Spock utilise la surcharge d'opérateurs de Groovy pour stuber les appels de méthode. Avec Java, nous devons appeler de vraies méthodes, ce qui signifie sans doute que le code résultant est plus verbeux et potentiellement moins expressif.

Maintenant, essayons quelques autres types de stubs.

Si nous arrêtions de nous soucier de notre argument de méthode et que nous voulions toujours retourner *true,* nous pourrions simplement utiliser un trait de soulignement :

```
paymentGateway.makePayment(_) >> true
```

Si nous voulions alterner entre différentes réponses, nous pourrions fournir une liste, pour laquelle chaque élément sera retourné dans l'ordre :

```
paymentGateway.makePayment(_) >>> [true, true, false, true]
```

Il y a plus de possibilités, et celles-ci pourraient être couvertes dans un futur article plus avancé sur le mock.

5.4. Vérification

Une autre chose que nous pourrions vouloir faire avec les simulations est d'affirmer que diverses méthodes ont été appelées avec les paramètres attendus. En d'autres termes, nous devons vérifier les interactions avec nos simulacres.

Un cas d'utilisation typique pour la vérification serait si une méthode sur notre maquette avait un type de retour *void .* Dans ce cas, en l'absence de résultat sur lequel nous pouvons opérer, nous n'avons aucun comportement déduit à tester via la méthode testée. Généralement, si quelque chose était retourné, alors la méthode testée pourrait fonctionner dessus, et c'est le résultat de cette opération qui serait ce que nous affirmons.

Essayons de vérifier qu'une méthode avec un type de retour void est appelée :

```
def "Should verify notify was called"() {
    given:
        def notifier = Mock(Notifier)

    when:
        notifier.notify('foo')

    then:
        1 * notifier.notify('foo')
}
```

Spock exploite à nouveau la surcharge de l'opérateur Groovy. En multipliant notre appel de méthode mocks par un, nous disons combien de fois nous nous attendons à ce qu'il ait été appelé.

Si notre méthode n'avait pas été appelée du tout ou si elle n'avait pas été appelée autant de fois que nous l'avons spécifié, alors notre test n'aurait pas réussi à nous donner un message d'erreur Spock informatif. Prouvons-le en s'attendant à ce qu'il ait été appelé deux fois :

```
2 * notifier.notify('foo')
```

Ensuite, voyons à quoi ressemble le message d'erreur. Nous ferons cela comme d'habitude; c'est assez instructif :

```
Too few invocations for:

2 * notifier.notify('foo')   (1 invocation)
```

Tout comme le stub, nous pouvons également effectuer une correspondance de vérification plus lâche. Si nous ne nous soucions pas du paramètre de notre méthode, nous pourrions utiliser un trait de soulignement :

```
2 * notifier.notify(_)
```

Ou si nous voulions nous assurer qu'il n'a pas été appelé avec un argument particulier, nous pourrions utiliser l'opérateur not :

```
2 * notifier.notify(!'foo')
```

Encore une fois, il y a plus de possibilités, qui pourraient être couvertes dans un futur article plus avancé.

6. Conclusion

Dans cet article, nous avons donné un aperçu rapide des tests avec Spock.

Nous avons démontré comment, en tirant parti de Groovy, nous pouvons rendre nos tests plus expressifs que la pile JUnit typique. Nous avons expliqué la structure des *spécifications* et des *fonctionnalités* .

Et nous avons montré à quel point il est facile d'effectuer des tests basés sur les données, et aussi à quel point les mocks et les assertions sont faciles via la fonctionnalité native de Spock.

La mise en œuvre de ces exemples peut être trouvée sur GitHub . Il s'agit d'un projet basé sur Maven, il devrait donc être facile à exécuter tel quel.