

UNIVERSITÉ PARIS 13

INSTITUT GALILÉE

MASTER 2 : EXPLORATION INFORMATIQUE DES DONNÉES ET
DÉCISIONNEL

Scala tutoriels scalaTest and uTest

Rédigé par:

LEZARK Halima (uTest)
SEDDIKI Hanae (ScalaTest)

Supervisé par:

Mr. BOUDES Pierre

January 23, 2019



Contents

1	Préparation de l'environnement de travail	2
1.1	Configuration du JDK	2
1.2	SBT	2
1.3	Création du projet	2
2	UTest : un Framework pour les tests de scala	3
2.1	Présentation des uTest :	3
2.2	C'est quoi un TestSuite ?	3
2.3	Pour Commencer	4
2.3.1	Définir et exécuter un TestSuite	4
2.3.2	Tests d'imbrication (Nesting tests)	5
2.3.3	Tests de Partage de la configuration (Sharing Setup tests)	6
2.3.4	Les utilitaires de test	8
3	ScalaTest	9
3.1	Le premier pas avec ScalaTest	9
3.2	Sélection des styles de test pour votre projet	11
3.2.1	Style 1 : FunSuite	11
3.2.2	Style 2 : FlatSpec	13
3.2.3	Style 3 : FunSpec	14
3.2.4	Style 4 : WordSpec	14
3.2.5	Style 5 : FreeSpec	15
3.2.6	Style 6 : PropSpec	15

1 Préparation de l'environnement de travail

Avant de commencer le développement du projet, nous avons besoin de préparer un environnement de travail.

Dans ce tutoriel, nous allons travailler sur linux et avec l'IDE IntelliJ IDEA.

Pour se faire, nous commençons par l'installation du JDK, un environnement d'exécution dans lequel les programmes Scala sont exécutés.

On utilise la commande : `sudo apt-get install openjdk-8-jdk` pour installer jdk.

Ensuite, on installe IntelliJ Ultimate. Lorsque vous démarrez IntelliJ, un écran de bienvenue s'affiche. La première étape avant de créer ou d'ouvrir un projet Scala consiste à installer le plugin Scala. Pour cela, allez-vous-en bas à droite de l'écran de bienvenue et choisissez Configure → Plugins → Browse JetBrains Plugins. Si cet écran de bienvenue ne s'affiche pas, allez à Préférences (Settings) → Plugins.

Par conséquent, **redémarrez IntelliJ** avant de suivre les autres étapes.

1.1 Configuration du JDK

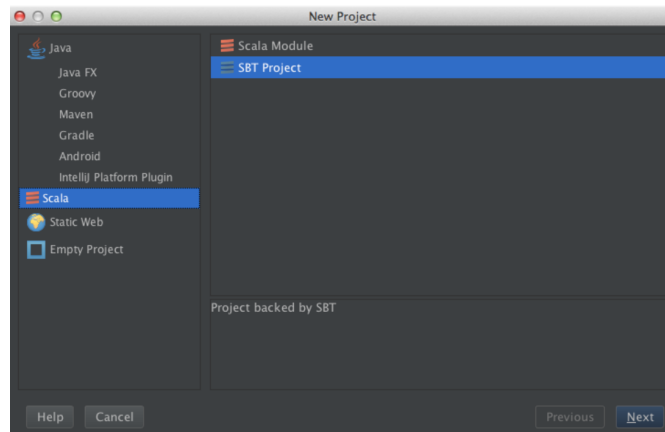
Sur l'écran de bienvenue, sur Configure → Project defaults → Project structure et ajouter JDK qui sera automatiquement ajouté, sinon, trouvez le chemin où il est stocké et sélectionnez-le dans le sélecteur de fichiers.

1.2 SBT

SBT est un outil de construction open source pour les projets Scala et Java, similaire à Maven et Ant de Java. Ses principales caractéristiques sont : Prise en charge native de la compilation de code Scala et de l'intégration à de nombreux frameworks de test Scala Compilation, test et déploiement continus.

1.3 Création du projet

La création du projet consiste à utiliser l'Assistant de projet. Pour l'utiliser, cliquez sur Créer un nouveau projet sur l'écran de bienvenue, puis sélectionnez **Scala** et enfin **Projet SBT**.



Cliquez sur Suivant pour spécifier le nom et l'emplacement du projet. Une fois ces informations saisies, IntelliJ IDEA créera un projet vide contenant un fichier build.sbt.

2 UTest : un Framework pour les tests de scala

2.1 Présentation des uTest :

uTest est un Framework de test pour le langage de programmation Scala. uTest est simple et convivial, il permet de se concentrer sur l'essentiel: les tests et le code.

Dans cette partie, on va explorer ce qui rend uTest intéressant et pourquoi on doit envisager de l'utiliser pour créer la suite de tests dans les projets Scala.

uTest n'est pas "essentiel" dans le sens où vous devez l'utiliser: d'autres Framework de test comme ScalaTest ou Specs2 sont toujours une option. Au lieu de cela, uTest est essentiel car il ne contient que les fonctionnalités qui sont absolument nécessaires pour tester votre code.

uTest prend en charge les tâches suivantes:

- Écrire des bouts de code à exécuter ("tests"), les étiqueter et les organiser soigneusement
- Partage de l'initialisation et d'autres codes entre vos différents tests
- Affirmer des choses pour s'assurer qu'elles sont vraies
- Sélectionner les tests que vous voulez exécuter et voir s'ils explosent

2.2 C'est quoi un TestSuit ?

Un testSuit est un conteneur qui contient un ensemble de tests qui aide les testeurs à exécuter et à signaler l'état d'exécution des tests. Cela peut prendre n'importe lequel des trois états, à savoir Actif, In Progress et complété.

Un scénario de test peut être ajouté à plusieurs suites de tests et plans de tests. Après avoir créé un plan de test, des suites de tests sont créées et peuvent comporter un nombre illimité de tests.

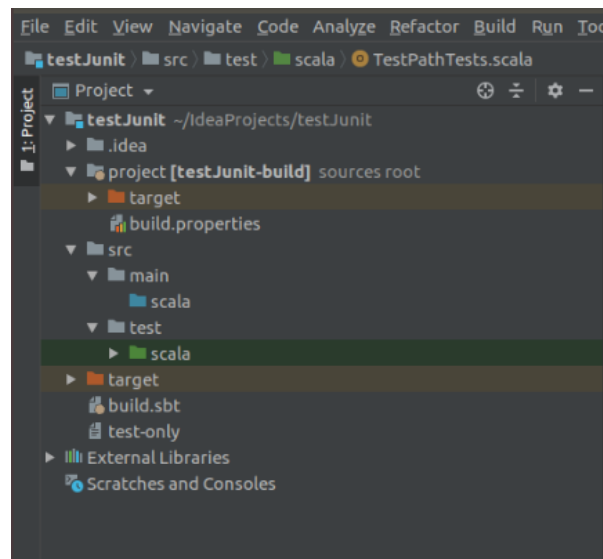
2.3 Pour Commencer

La plupart des gens qui utilisent uTest exécuteront des tests via SBT.

On doit faire la modification nécessaire au fichier build.sbt et on peut immédiatement commencer à définir et exécuter des tests par programmation.

Dès la création du projet scala, le fichier build.sbt se génère automatiquement, ainsi qu'un dossier src contenant deux dossiers : main et test.

Dans le dossier main, on crée des classes scala où on met le code et dans le dossier test on met les objets scala contenant les tests.



2.3.1 Définir et exécuter un TestSuite

Dans le dossier src / test / scala / , on met :

```
1 package test.utest.examples
2
3 import utest._
4
5 object HelloTests extends TestSuite{
6   val tests = Tests{
7     'test1 - {
8       throw new Exception("test1")
9     }
10    'test2 - {
11      1
12    }
13    'test3 - {
14      val a = List[Byte](xs = 1, 2)
15      a(10)
16    }
17  }
18 }
19
20
```

```
sbt testJunit/test
```

On lance le test sur l'objet `HelloTests`, on obtient :



Notez que les tests de la suite peuvent être imbriqués les uns dans les autres, mais uniquement directement. Par exemple. Vous ne pouvez pas définir de tests dans les instructions if ou for-loops.

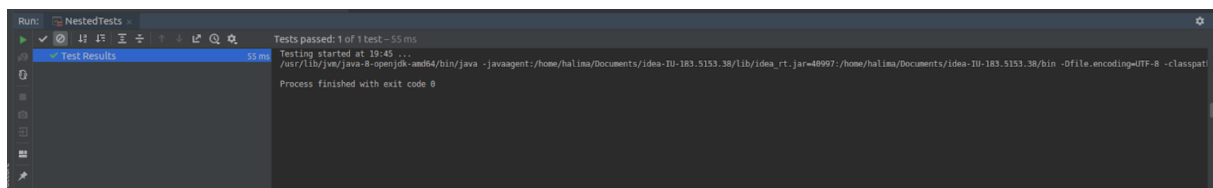
uTest s'appuie sur la structure de test pour être connue statiquement au moment de la compilation.

```

1 package test.uteest.examples
2
3
4 import utest._
5
6 object NestedTests extends TestSuite{
7   val tests = Tests{
8     val x = 1
9     'outer1 - {
10       val y = x + 1
11
12       'inner1 - {
13         assert(x == 1, y == 2)
14         (x, y)
15       }
16       'inner2 - {
17         val z = y + 1
18         assert(z == 3)
19       }
20     }
21     'outer2 - {
22       'inner3 - {
23         assert(x > 1)
24       }
25     }
26   }
27 }
28

```

Après exécution, on obtient :



- 4 tests exécutés => 4 Réussis

Comme vous pouvez le constater, les trois éléments différents sont imbriqués les uns dans les autres.

2.3.3 Tests de Partage de la configuration (Sharing Setup tests)

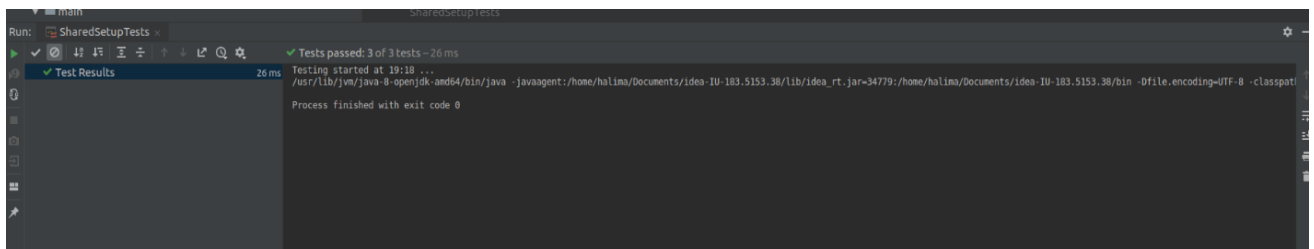
Vous pouvez utiliser les tests imbriqués pour regrouper les tests associés et les faire partager.

```

1 package test.uteest.examples
2
3
4 import utest._
5
6 object SharedSetupTests extends TestSuite{
7   val tests = this{
8     var x = 0
9
10    A{
11      x += 1
12      'X{
13        x += 2
14        assert(x == 3)
15        x
16      }
17    }
18    x += 3
19    assert(x == 4)
20    x
21  }
22  B{
23    x += 4
24    'Z{
25      x += 5
26      assert(x == 9)
27      x
28    }
29  }
30 }

```

- Cela donnera :



Ici, nous partageons l'initialisation de la variable `x` entre tous les différents sous-tests dans le même dossier.

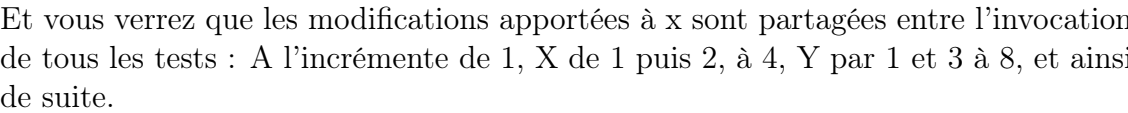
Bien qu'ils soient partagés lexicalement, ces helpers sont recréés pour chaque test qui est exécuté, donc si elles contiennent un état mutable (par exemple, des collections mutables ou vars) vous n'avez pas à vous soucier des mutations de plusieurs tests qui interfèrent entre eux.

Cela permet de partager de manière concise un code de configuration commun entre tests liés dans le même groupe, tout en évitant les interférences entre les tests en raison de la mutation des appareils partagés

Si vous voulez que les rencontres soient vraiment partagé entre différents tests, définissez-le en dehors du bloc `this { }` :

```
1 package test.utest.examples
2
3 import utest._
4
5 object SharedFixturesTests extends TestSuite{
6   var x = 0
7   val tests = this{
8     'A{
9       x += 1
10      'X{
11        x += 2
12        assert(x == 4)
13        x
14      }
15      'Y{
16        x += 3
17        assert(x == 8)
18        x
19      }
20    }
21    'B{
22      x += 4
23      'Z{
24        x += 5
25        assert(x == 21)
26        x
27      }
28    }
29  }
30 }
```

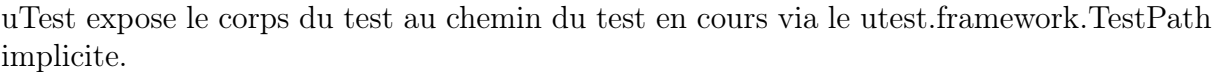
- Après exécution :



Les utilitaires de test

JUnit fournit une gamme d'utilitaires de test qui ne sont pas strictement nécessaires, mais visent à rendre l'écriture des tests beaucoup plus pratique.

TestPath



3 ScalaTest

3.1 Le premier pas avec ScalaTest

ScalaTest est l'outil de test le plus flexible et le plus populaire de l'écosystème Scala. Avec ScalaTest, vous pouvez tester le code Scala, Scala.js (JavaScript) et Java.

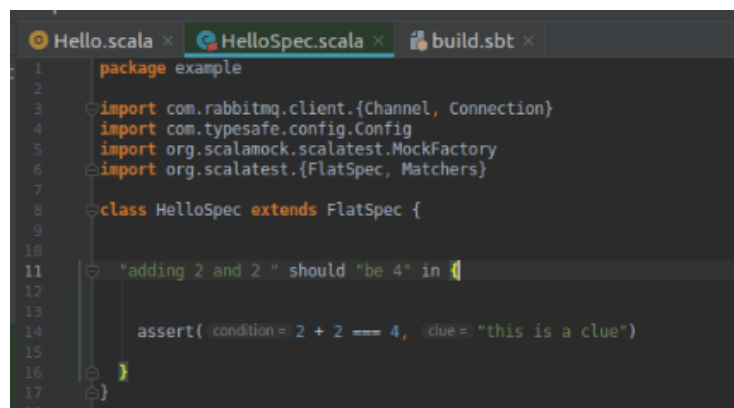
Pour pouvoir visualiser rapidement certains des tests pouvant être écrits avec ScalaTest, nous pouvons tirer parti du modèle test-patterns-scala de l'activateur Typesafe.

Pour ce faire, nous allons créer un objet scala « Hello », dans le chemin `src/main/scala/example` :



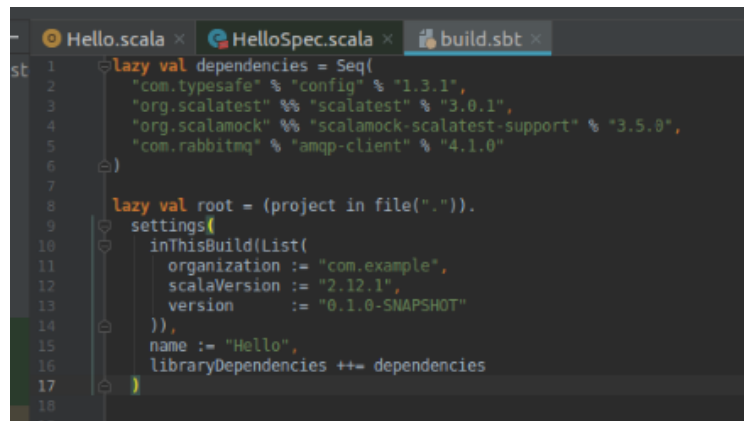
```
1 package example
2
3 import com.rabbitmq.client.{Channel, Connection}
4 import com.typesafe.config.Config
5
6 object Hello {
7
8   def main(args: Array[String]): Unit = {
9
10     println("Hello world")
11   }
12
13   def addInt(a: Int, b: Int): Int = {
14
15     var sum = 0
16     sum = 0
17     return sum
18   }
19
20 }
21
22
23
```

Dans le chemin `src/test/scala/example`, on crée une classe « HelloSpec » :



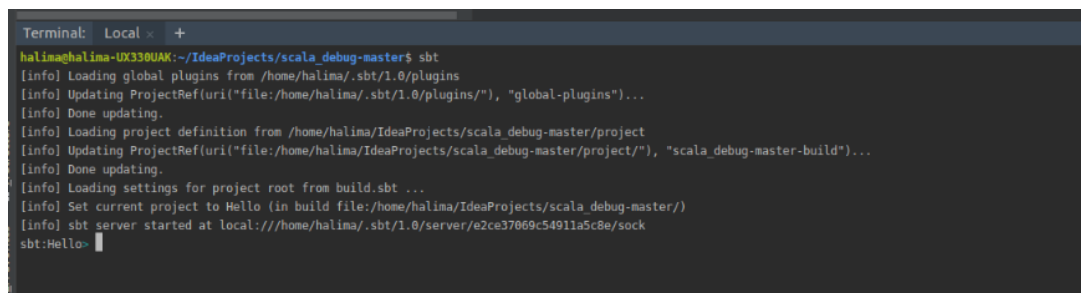
```
1 package example
2
3 import com.rabbitmq.client.{Channel, Connection}
4 import com.typesafe.config.Config
5 import org.scalamock.scalatest.MockFactory
6 import org.scalatest.{FlatSpec, Matchers}
7
8 class HelloSpec extends FlatSpec {
9
10
11   "adding 2 and 2" should "be 4" in {
12
13     assert( condition = 2 + 2 == 4, clue = "this is a clue")
14
15   }
16
17 }
18
```

Le plus important est de faire la configuration du fichier `build.sbt` :



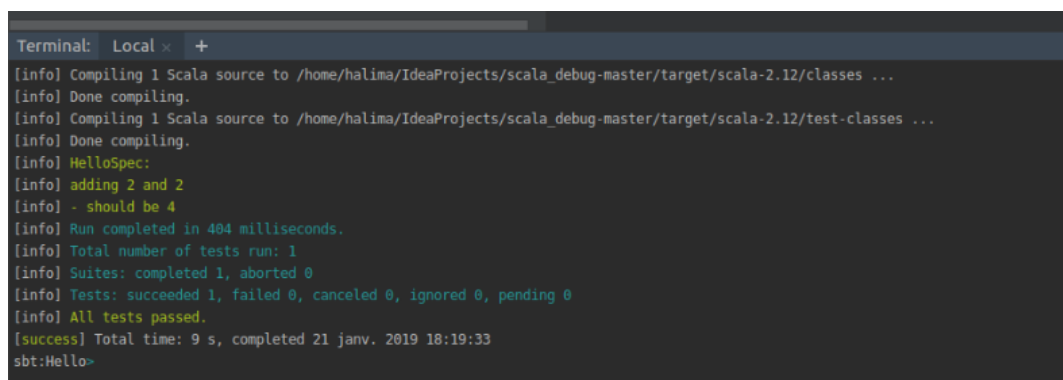
```
1 lazy val dependencies = Seq(  
2   "com.typesafe" % "config" % "1.3.1",  
3   "org.scalatest" %% "scalatest" % "3.0.1",  
4   "org.scalamock" %% "scalamock-scalatest-support" % "3.5.0",  
5   "com.rabbitmq" % "amqp-client" % "4.1.0"  
6 )  
7  
8 lazy val root = (project in file(".")).  
9   settings(  
10    inThisBuild(List(  
11      organization := "com.example",  
12      scalaVersion := "2.12.1",  
13      version      := "0.1.0-SNAPSHOT"  
14    )),  
15    name := "Hello",  
16    libraryDependencies ++= dependencies  
17  )  
18
```

On lance le sbt :



```
Terminal: Local x +  
halima@halima-UX330UAK:~/IdeaProjects/scala_debug-master$ sbt  
[info] Loading global plugins from /home/halima/.sbt/1.0/plugins  
[info] Updating ProjectRef(uri("file:/home/halima/.sbt/1.0/plugins/"), "global-plugins")...  
[info] Done updating.  
[info] Loading project definition from /home/halima/IdeaProjects/scala_debug-master/project  
[info] Updating ProjectRef(uri("file:/home/halima/IdeaProjects/scala_debug-master/project/"), "scala_debug-master-build")...  
[info] Done updating.  
[info] Loading settings for project root from build.sbt ...  
[info] Set current project to Hello (in build file:/home/halima/IdeaProjects/scala_debug-master/)  
[info] sbt server started at local:///home/halima/.sbt/1.0/server/e2ce37069c54911a5c8e/sock  
sbt:Hello>
```

Puis, on lance le scalaTest sur la classe de test :



```
Terminal: Local x +  
[info] Compiling 1 Scala source to /home/halima/IdeaProjects/scala_debug-master/target/scala-2.12/classes ...  
[info] Done compiling.  
[info] Compiling 1 Scala source to /home/halima/IdeaProjects/scala_debug-master/target/scala-2.12/test-classes ...  
[info] Done compiling.  
[info] HelloSpec:  
[info] adding 2 and 2  
[info] - should be 4  
[info] Run completed in 404 milliseconds.  
[info] Total number of tests run: 1  
[info] Suites: completed 1, aborted 0  
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0  
[info] All tests passed.  
[success] Total time: 9 s, completed 21 janv. 2019 18:19:33  
sbt:Hello>
```

Le test donne un résultat positif.

Si on change un chiffre au niveau du code, nous allons avoir un résultat de test négatif :

```

package example

import com.rabbitmq.client.{Channel, Connection}
import com.typesafe.config.Config
import org.scalamock.scalatest.MockFactory
import org.scalatest.{FlatSpec, Matchers}

class HelloSpec extends FlatSpec {

  "adding 2 and 2 " should "be 4" in {

    assert( condition = 2 + 2 == 5, clue = "this is a clue")

  }
}

```

```

Terminal: Local x +
[info] HelloSpec:
[info] adding 2 and 2
[info] - should be 4 *** FAILED ***
[info]   4 did not equal 5 this is a clue (HelloSpec.scala:14)
[info] Run completed in 256 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed tests:
[error]   example.HelloSpec
[error] (Test / test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 3 s, completed 21 janv. 2019 18:20:38
sbt:Hello>

```

3.2 Sélection des styles de test pour votre projet

ScalaTest prend en charge différents styles de test, chacun étant conçu pour répondre à un ensemble particulier de besoins. Pour vous aider à trouver les styles les mieux adaptés à votre projet, cette partie décrira les cas d'utilisation prévus pour chaque option.

Nous allons utiliser un ensemble de styles de test pour chaque projet, Cela permet aux styles de test de s'adapter à l'équipe tout en maintenant l'uniformité dans la base de code du projet.

Si vous préférez faire vos propres choix, nous vous donnerons un aperçu rapide des avantages et des inconvénients de chaque trait de style.

3.2.1 Style 1 : FunSuite

Test01

Dans ce cas, la classe Test01 entend de org.scalatest.fun suite, dedans nous avons mis deux tests qui sont basics.

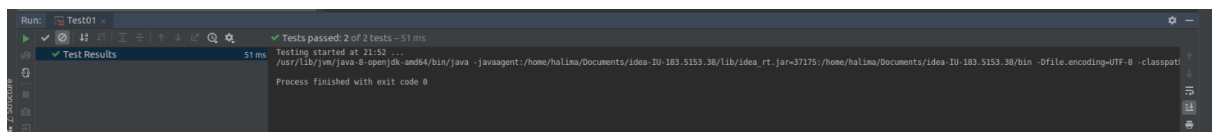
```

Test01.scala x Test02.scala x SetSuite.scala x SetSpec.scala x
1 package example
2
3 package scalatest
4 import org.scalatest.FunSuite
5
6 class Test01 extends FunSuite {
7   test( testName = "Very Basic") {
8     assert( condition = 1 == 1)
9   }
10  test( testName = "Another Very Basic") {
11    assert( condition = "Hello World" == "Hello World")
12  }
13 }
14

```

‘test’ est une méthode défini dans FunSuite.

Après exécution, les deux tests sont bien passés.



Test02

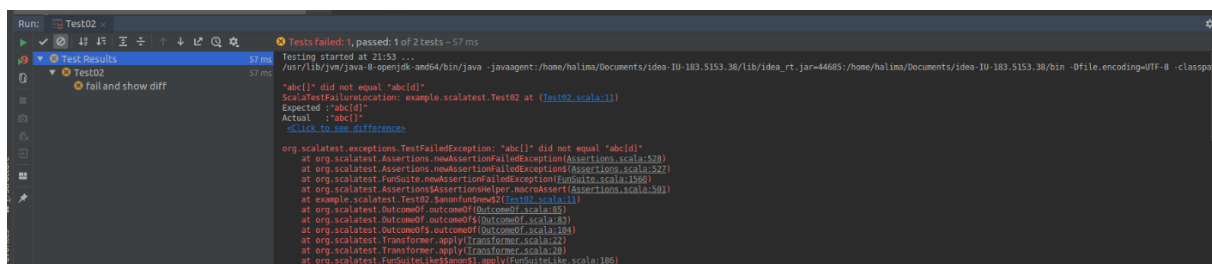
Pareil pour la classe test02, mais maintenant nous avons introduit une erreur dans l'un des deux tests.

```

Test01.scala x Test02.scala x SetSuite.scala x SetSpec.scala x
1 package example
2
3 package scalatest
4 import org.scalatest.FunSuite
5
6 class Test02 extends FunSuite {
7   test( testName = "pass") {
8     assert( condition = "abc" == "abc")
9   }
10  test( testName = "fail and show diff") {
11    assert( condition = "abc" == "abcd") // provide reporting info
12  }
13 }
14

```

Après exécution, le premier test a réussi et le deuxième test n'a pas passé.

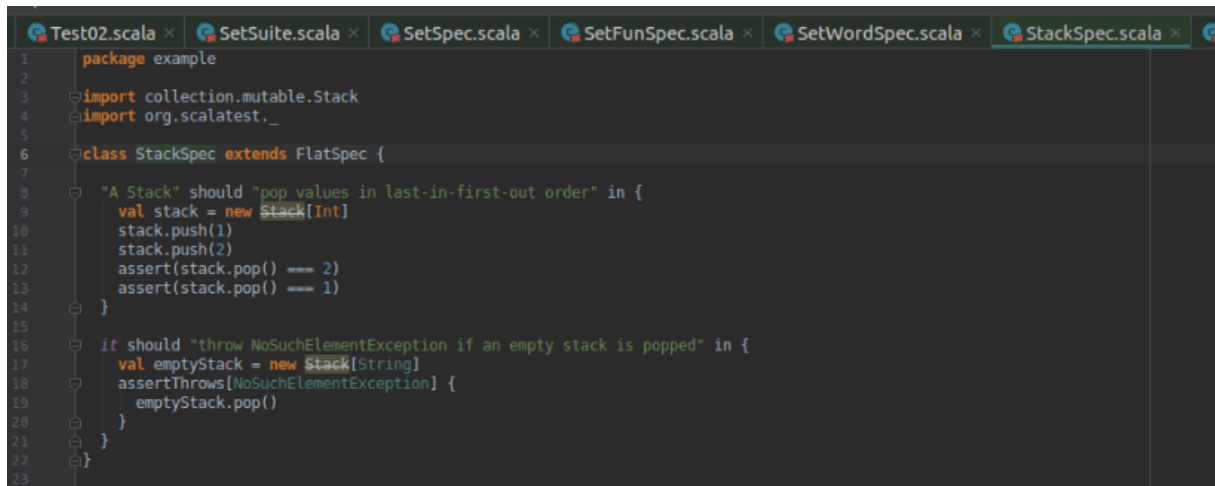


Il existe d'autres styles qu'on peut appliquer sur les classes de test à part le FunSuite.

Vers la suite je vais les citer, et je vais donner également un exemple d'application pour chacun.

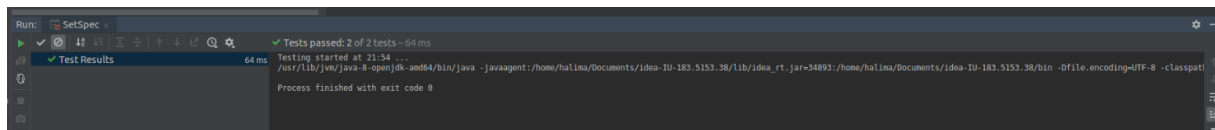
3.2.2 Style 2 : FlatSpec

FlatSpec est une bonne première étape pour passer de xUnit à BDD. Sa structure est plate comme xUnit, très simple et familière, mais les noms de test doivent être écrits dans un style de spécification : "X should Y", "A must B", etc.



```
1 package example
2
3 import collection.mutable.Stack
4 import org.scalatest._
5
6 class StackSpec extends FlatSpec {
7
8   "A Stack" should "pop values in last-in-first-out order" in {
9     val stack = new Stack[Int]
10    stack.push(1)
11    stack.push(2)
12    assert(stack.pop() === 2)
13    assert(stack.pop() === 1)
14  }
15
16   it should "throw NoSuchElementException if an empty stack is popped" in {
17     val emptyStack = new Stack[String]
18     assertThrows[NoSuchElementException] {
19       emptyStack.pop()
20     }
21   }
22 }
23
```

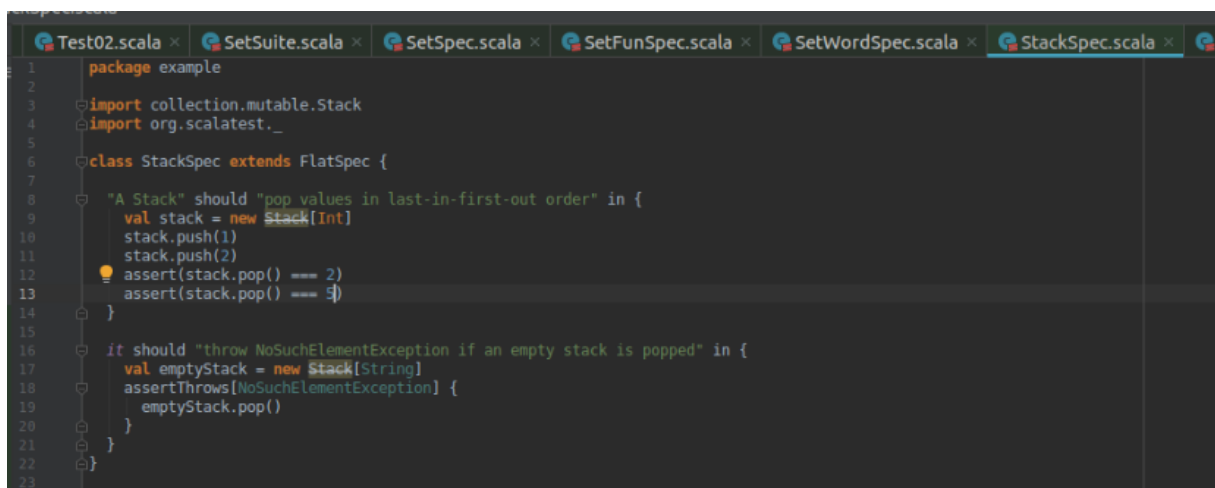
Après exécution, les deux tests sont bien passés.



```
Run: SetSpec
Test Results
64 ms
Tests passed: 2 of 2 tests - 64 ms
Testing started at 21:54 ...
Process finished with exit code 0
```

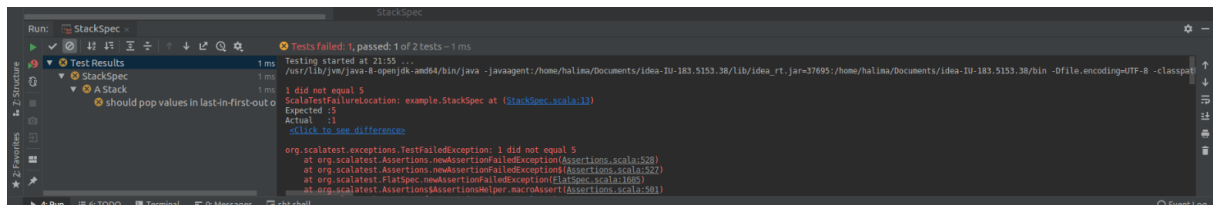
Ensuite, on essaye de modifier le code pour le tester, du coup je remplace :

‘ assert(stack.pop() === 1)’ par ‘ assert(stack.pop() === 5)’



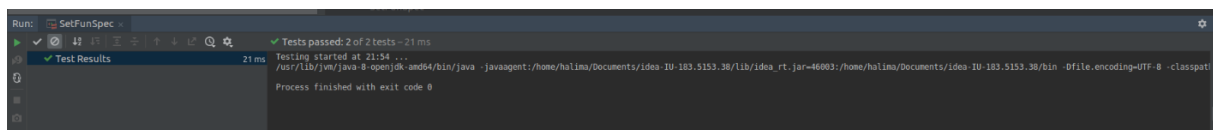
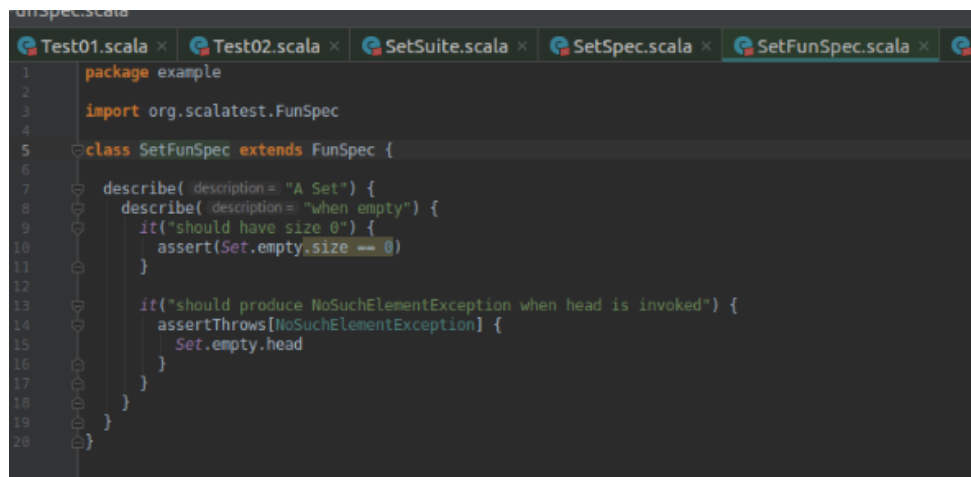
```
1 package example
2
3 import collection.mutable.Stack
4 import org.scalatest._
5
6 class StackSpec extends FlatSpec {
7
8   "A Stack" should "pop values in last-in-first-out order" in {
9     val stack = new Stack[Int]
10    stack.push(1)
11    stack.push(2)
12    assert(stack.pop() === 2)
13    assert(stack.pop() === 5)
14  }
15
16   it should "throw NoSuchElementException if an empty stack is popped" in {
17     val emptyStack = new Stack[String]
18     assertThrows[NoSuchElementException] {
19       emptyStack.pop()
20     }
21   }
22 }
23
```

Après exécution, un test est passé sur deux.



3.2.3 Style 3 : FunSpec

Trait qui facilite un style de développement « basé sur le comportement », dans lequel les tests sont combinés avec du texte qui spécifie le comportement que les tests vérifient.

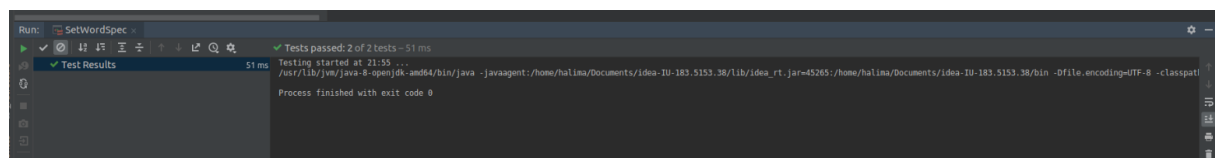


3.2.4 Style 4 : WordSpec

WordSpec constitue souvent le moyen le plus naturel de transférer des tests specsN vers ScalaTest. WordSpec est très prescriptif dans la façon dont le texte doit être écrit.

Ce Trait WordSpec est ainsi nommé parce que votre texte de spécification est structuré en plaçant des mots après des chaînes.

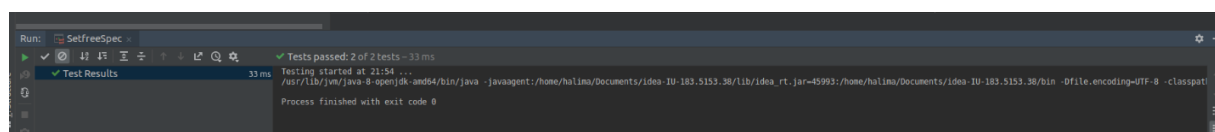
```
SetWordSpec.scala
Test01.scala x Test02.scala x SetSuite.scala x SetSpec.scala x SetFunSpec.scala x SetWordSpec.scala x
1 package example
2
3
4 import org.scalatest.WordSpec
5
6 class SetWordSpec extends WordSpec {
7
8   "A Set" when {
9     "empty" should {
10       "have size 0" in {
11         assert(Set.empty.size == 0)
12       }
13
14       "produce NoSuchElementException when head is invoked" in {
15         assertThrows[NoSuchElementException] {
16           Set.empty.head
17         }
18       }
19     }
20   }
21 }
```



3.2.5 Style 5 : FreeSpec

Ce Trait facilite un style de développement « basé sur le comportement », dans lequel les tests sont imbriqués dans des clauses de texte dénotées avec l'opérateur de tiret (-).

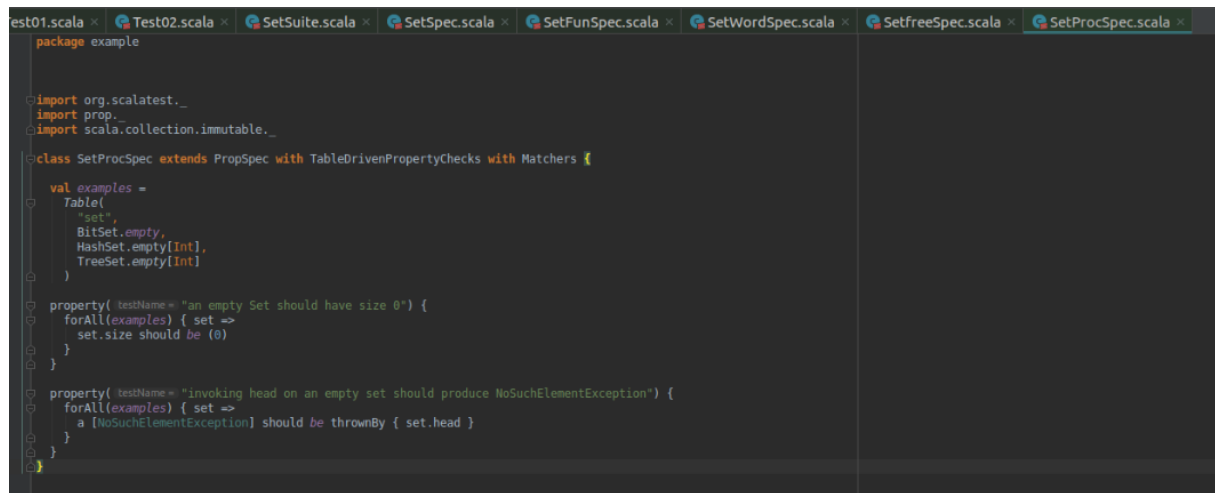
```
Test01.scala x Test02.scala x SetSuite.scala x SetSpec.scala x SetFunSpec.scala x SetWordSpec.scala x SetFreeSpec.scala x
1 package example
2
3
4 import org.scalatest.FreeSpec
5
6 class SetFreeSpec extends FreeSpec {
7
8   "A Set" - {
9     "when empty" - {
10       "should have size 0" in {
11         assert(Set.empty.size == 0)
12       }
13
14       "should produce NoSuchElementException when head is invoked" in {
15         assertThrows[NoSuchElementException] {
16           Set.empty.head
17         }
18       }
19     }
20   }
21 }
```



3.2.6 Style 6 : PropSpec

Une suite de tests basés sur les propriétés.

Ce trait facilite un style de test dans lequel chaque test est composé d'une vérification de propriété. Les tests sont enregistrés via une méthode "property", et reçoivent un nom et un corps.



```
package example

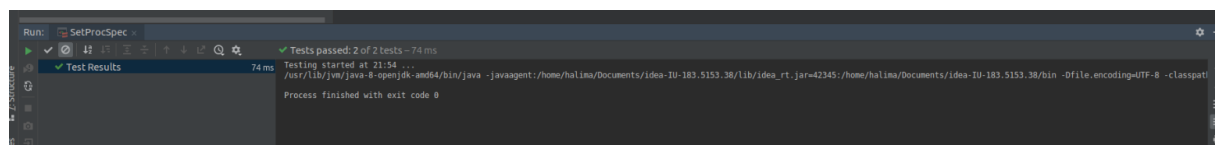
import org.scalatest._
import prop._
import scala.collection.immutable._

class SetProcSpec extends PropSpec with TableDrivenPropertyChecks with Matchers {

  val examples =
    Table(
      "set",
      BitSet.empty,
      HashSet.empty[Int],
      TreeSet.empty[Int]
    )

  property(testName = "an empty Set should have size 0") {
    forAll(examples) { set =>
      set.size should be (0)
    }
  }

  property(testName = "invoking head on an empty set should produce NoSuchElementException") {
    forAll(examples) { set =>
      a [NoSuchElementException] should be thrownBy { set.head }
    }
  }
}
```



```
Run: SetProcSpec
Tests passed: 2 of 2 tests - 74 ms
Testing started at 21:54 ...
Process finished with exit code 0
```

[Lien vers le Github du projet](#)