SBT

L'objectif de ce premier lab est de manipuler le shell SBT afin de se familiariser avec les différentes tasks utiles de l'outil

Lancer le RPEL

Ouvrir une console et lancer la commande :

```
sbt
```

si vous obtenez le prompt suivant :

```
>
```

C'est que vous êtes prêt à en découdre avec SBT.

1. prise de contact avec SBT

Dans cette première partie, il s'agit juste d'exécuter des commandes pour se familiariser avec l'environnement.

1.1. Help

Utiliser la commande help pour obtenir l'ensemble des commandes de base de SBT. Très utile lorsque l'on découvre SBT.

Aussi, on peut utiliser cette commande en l'applicant sur une task directement (comme un man)

```
>help show
```

1.2 Tasks et settings

Exécuter les tâches tasks et settings

La première affiche l'ensemble des tâches disponibles par défaut dans SBT. La second affiche l'ensemble des settings disponible par défaut dans SBT.

Si le projet sous-jacent ne respecte pas les conventions, ces informations sont pratiques.

1.3 show

Cette commande permet d'afficher le résultat d'une task ou la valeur d'un

setting Lancer les commandes suivantes :

```
>show baseDirectory
>show scalaVersion
>show name
>show offline
```

Les commandes ci-dessus, loin d'être exhaustive, montrent comment obtenir des informations sur le projet et mettent en évidence l'aspect interactif de SBT.

1.4 inspect

Cette commande permet d'obtenir l'arbre de dépendance d'une task ou d'un setting pour que celle-ci soit exécutée.

```
>inspect tree clean
```

Cette commande affichera l'arbre d'exécution de la commande.

```
>inspect uses clean
>inspect uses compile
```

Avec cette commande, on peut découvrir les différents workflows défini sur le projet.

1.5 !

Cette commande permet de voir l'historique des commandes déjà passée et les ré exécuter automatiquement

En tapant

```
>!
```

On accède au détail des commandes possibles, vous pouvez les tester :

- !! Execute the last command again
- !: Show all previous commands
- !:n Show the last n commands
- !n Execute the command with index n, as shown by the !: command
- !-n Execute the nth command before this one
- !string Execute the most recent command starting with 'string'
- !?string Execute the most recent command containing 'string'

2. Exécuter du code avec SBT

Ici nous allons voir la configuration nécessaire pour exécuter du code selon la typologie de projet :

2.1 Classe main

Dans un premier temps, nous allons voir que SBT n'a besoin d'aucune configuration pour exécuter une classe Scala (ou java).

Dans un fichier nommé MainClass.scala, copier le code suivant :

```
object main extends App {
    println("Hello world")
}
```

remarque : Même fonctionnement avec une classe écrite en Java

Ensuite positionner votre console dans le répertoire et lancer le shell SBT. Ensuite lancer la commande :

```
run
```

Vous devriez voir le texte : Hello word s'afficher. On constate que sbt ne nécessite aucune configuration pour fonctionner.

--> Ici, nous avons pu exécuter du code sans fournir un descripteur à SBT (CoC)

3. Ecrire sa première tâche

L'idée ici est décrire une tâche permettant de créer la structure standard d'un projet maven (structure sur laquelle SBT se base par défaut).

Pour cela, nous allons devoir coder en Scala, youpi ! :)

Dans un répertoire de votre choix, créer un fichier **build.sbt**

Lorsque l'on créé une tâche, la première chose à faire est de définir la clef avec laquelle on va l'invoquer. Voici la ligne de code permettant de faire ça :

```
val init = taskKey[Unit]("init task")
```

Vous pouvez copier/coller la ligne ci-dessus dans le fichier créé précédemment.

Maintenant, que nous avons créé la tâche init, il va falloir lui donner un comportement. Nous allons utiliser plusieurs éléments pour écrire la tâche :

- La classe utilitaire **IO** qui permet d'interagir avec le système de fichier.
- le setting : BaseDirectory qui va nous donner l'endroit à partir duquel on

va créer l'arborescence de répertoire.

Le code ci-dessous permet d'affecter le comportement à la tâche init.

```
init := {
    val root = baseDirectory.value


IO.createDirectories(List("src","src/main","src/main/scala","src/tes
t/
scala","src/test/resources","src/main/resources").map(f => root /
f))
    val s = streams.value
    s.log.info("init task ended")

}
```

Que fait le code ?

Dans un premier temps, il récupère le répertoire racine du projet. Ensuite, à l'aide de la classe utilitaire IO, on va créer un ensemble de directories que l'on passe en paramètre de la méthode createDirectories. et sur chaque élément de la liste on rajoute la racine. Les deux dernières lignes récupère une référence sur le logger et affiche un message

Copier/coller le code dans le fichier build.sbt. Positionnez vous dans la console et le lancer le shell sbt. A l'invit' de commande taper

```
>init
```

Vous devriez voir le message : *init task ended* s'afficher. L'arborescence des répertoires devrait être créée.

Conlusion

Nous savons maintenant écrire des tâches.. Oui, car finalement écrire une tâche ne demande pas une connaissance poussée. En effet il suffit de connaitre seulement 3 choses :

- = -> Initialisation de la clef
- := -> affectation du corps de la tâche.
- value -> permet de récupérer le résultat de la tâche.

4.Exécuter des tâches en parallèle

Maintenant, nous allons voir comment introduire une dépendance entre

plusieurs tâches et ainsi créer une workflow d'exécution.

Dans une premier temps, Nous allons écrire deux tâches :

```
val oneLongTask = taskKey[Int]("One Long Task")
val otherLongTask = taskKey[Int]("Other Long Task")
```

Deux tâches qui auront un traitement artificiel de 10s :

```
oneLongTask := {
  val s = streams.value
  s.log.info("start traitement 1")
  Thread.sleep(5000)
  s.log.info("end traitement 1")
  100
}

otherLongTask := {
  val s = streams.value
  s.log.info("start traitement 2 ")
  Thread.sleep(5000)
  s.log.info("end traitement 2")
  200
}
```

Ensuite, nous allons créer une tâche qui dépendra du résultat de l'exécution des deux tâches précédente :

```
val mainTask = taskKey[Unit]("main task")

mainTask := {
  println(onelongTask.value + otherLongTask.value)
}
```

Exécutons la tâche *mainTask*. Nous constatons que le temps d'exécution globale est de 5 secondes, ce qui m'est en évidence que les deux premières tâches sont exécutées en parallèle.

Maintenant, on va rajouter une dépendance entre la tâche *oneLongTask* et *otherLongTask*

```
oneLongTask := {
  val s = streams.value
  s.log.info("start traitement 1")
  Thread.sleep(5000)
  s.log.info("end traitement 1")
  100 + otherLongTask.value.
```

```
}
```

Avant de re-exécuter le code, essayons d'anticiper le résultat et la manière d'y arriver. Dans cette nouvelle configuration, les tâches oneLongTask et otherLongTask seront exécutées séquentiellement (du à la dépendance entre elle).

Aussi on peut remarquer que la tâche otherLongTask est appelée deux fois dans le workflow. Que constatez vous après exécution?

5. projet multi-module

Ici, nous allons voir quelques aspects d'un projet multi-module. Avant tout, il faut récupérer l'archive (hands-on/sbtdemoProject.zip) depuis le projet sur github.

Ce projet comporte une implémentation simple avec des tests. Ce projet nous permettra de manipuler plusieurs commandes autour des tests.

Liste des commandes à utiliser :

- test = Permet de lancer tous les tests
- testQuick = Permet de lancer uniquement les tests dépendants du code modifier
- ~testQuick = permet de lancer à chaque modification du code source (avec les propriétés de la commande testQuick)
- [nom du module]/test = permet de lancer uniquement les tests d'un module.

Librairies de test

Ce lab à pour objectif de vous faire découvrir les librairies de tests issu de l'écosystème de scala. Pour ce faire, on va écrire des tests unitaires avec chacune des librairies.

Nous utiliserons la classe suivante pour faire nos tests :

```
import scala.util.Try

/**
  * Created by fsznajderman
  */
object Calculator {

  def +(a: Int,b:Int): Int ={
    a+b
```

```
  }

  def -(a: Int,b:Int): Int ={
    a-b
  }


  def *(a: Int,b:Int): Int ={
    a-b
  }

  def /(a:Int, b:Int): Try[Float] ={
    Try(a/b)
  }

}
```

Créer un descripteur de déploiement (build.sbt) et copier/coller les lignes ci-dessous Créer la structure de répertoire du projet à l'aide de la tâche init créée précédemment.

Dans les sources scala, ajouter la Calculator.scala Dans le répertoire de test scala, vous allez créer différentes classes de tests selon les différentes librairies.

ScalaTest

Documentation de référence : http://www.scalatest.org/user_guide

FunSuite

Utiliser le template suivant pour la création de de la classe :

```
class test1 extends FunSuite {
}
```

Attention de faire les bon imports. Ecrire les tests permettant de tester la classe calculator.

FlatSpec

Utiliser le template suivant pour la création de de la classe :

```
class test2 extends FlatSpec {
}
```

Attention de faire les bon imports. Ecrire les tests permettant de tester la

classe calculator.

Spec2

Utiliser le template suivant pour la création de de la classe :

```
class test3 extends Specification {
}
```

Attention de faire les bon imports. Ecrire les tests permettant de tester la classe calculator.

ScalaCheck

Documentation de référence :
https://github.com/rickynils/scalacheck/wiki/User-Guide

Utiliser le template suivant pour la création de de la classe :

```
object test4 extends Properties("Calculator") {
}
```

Attention de faire les bon imports. Ecrire les tests permettant de tester la classe calculator.