# Scala sur Intellij (TP guidé)

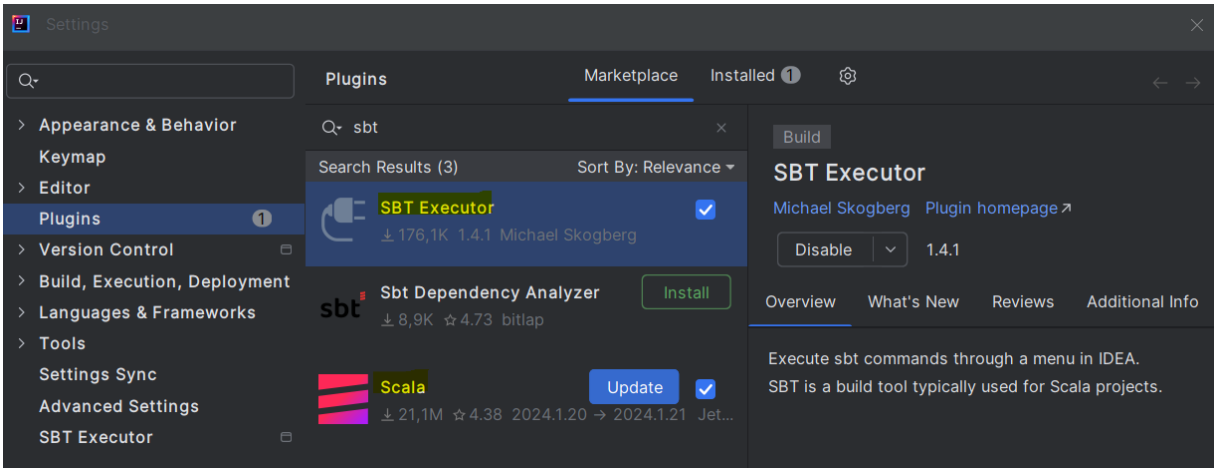
## Installation de Intellij Community Edition

https://www.jetbrains.com/fr-fr/idea/download/?section=windows


Ouvrez l'application. Allez dans Menu « Fichier » et cliquez sur « Paramètres ».

Allez dans la rubrique « Plugins » et tapez dans la barre de recherche « sbt ».

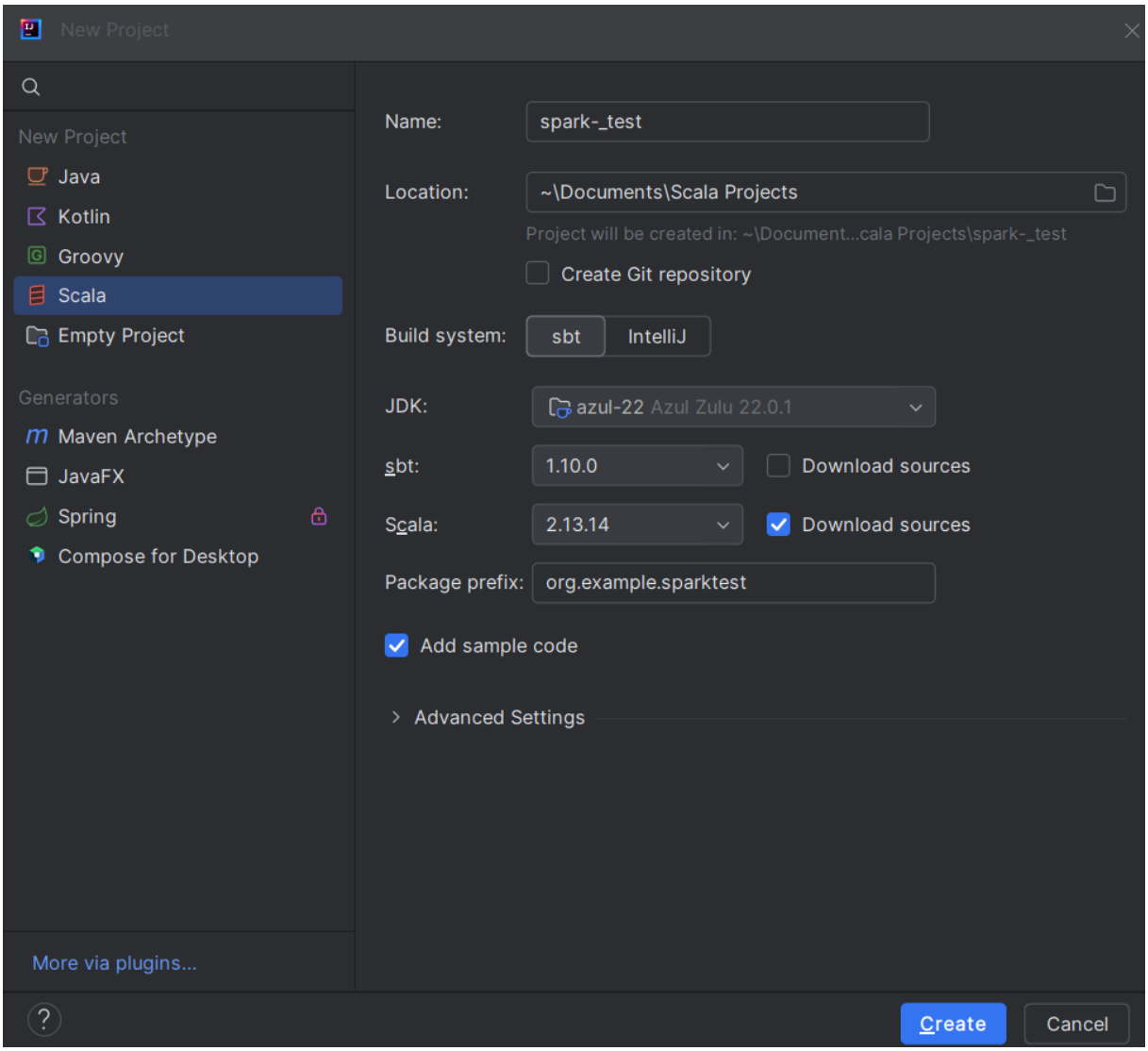
Vous téléchargerez le plugin « Scala » et « SBT Executor ».



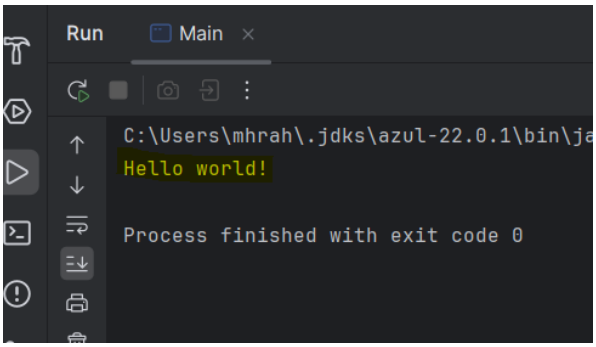

## Création d'un nouveau projet

Dans le menu cliquez sur Fichier → Nouveau → Projet

Mettez un nom à votre projet et vérifiez que votre système soit sur « sbt » et que les versions de JDK (e.g. Azul), sbt (dernière version) et scala (2.13.X) soient correctes. Cf. ci-dessous les paramètres à cocher et le package à écrire. Une fois terminée vous créerez le projet (sur la présente fenêtre ou sur une nouvelle).




Vérifiez que la ligne de code de base « Hello World » marche correctement.


Pratiquez avec d'autres lignes de code tout en restant sur les accolades { } de main().

Aparte : Si vous voulez utiliser scala sur un terminal, il est recommandé d'utiliser un terminal Ubuntu WSL (**Windows Subsystem for Linux**). Voici un tuto simple pour l'installer :

https://goodtech.info/comment-installer-le-terminal-ubuntu-sur-windows-10/

**Installation de Spark dans Intellij**

Ouvrez le fichier build.sbt

Dans le site MVN Repository, vous copierez dans ce fichier « build » les lignes de commande de Spark Project Core et Spark Project SQL en prenant la dernière version et correspondant à Scala 2.13. Dans notre cas ça sera la version de Spark sera "3.5.1"

```
libraryDependencies += "org.apache.spark" %% "spark-core" % "3.5.1"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "3.5.1" %
"provided"
```

**Importation d'un fichier .csv**

Dans votre projet Spark-Scala, créez un dossier « data ».

Déplacer le fichier téléchargé « AAPL.csv » dans ce dossier « data ».

Ecrivez ce code et exécutez-le :

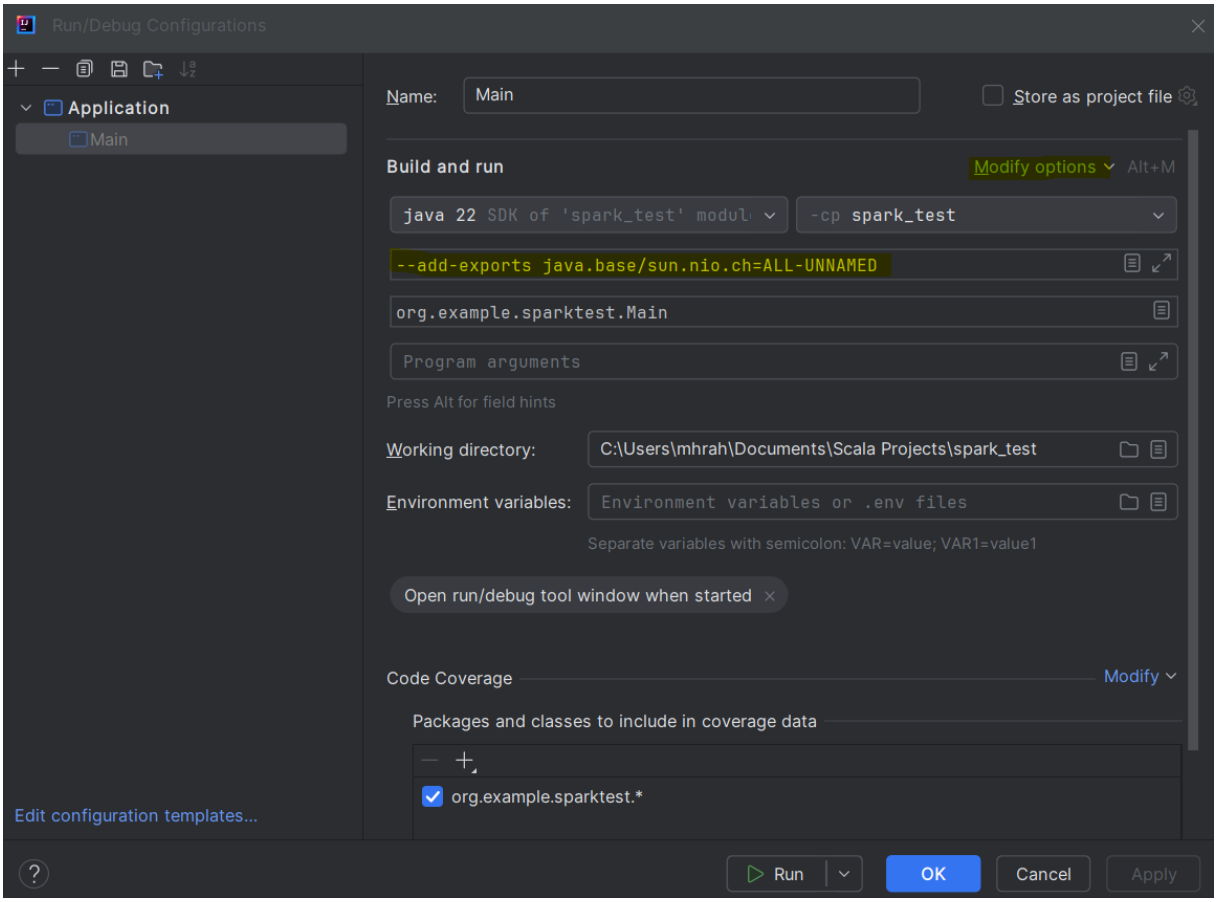
```
import org.apache.spark.sql.SparkSession

object Main {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .appName("spark_test")
      .master("local[*]")
      .getOrCreate()

    val df = spark.read
      .option("header", value = true)
      .csv("data/AAPL.csv")

    df.show()

  }
}
```

Que se passe-t-il ?

**Résolution du problème**

Allez dans Menu « Exécuter » → « Configurations», dans « Modifier les options » cocher « ajouter les options VM » et tapez :

```
--add-exports java.base/sun.nio.ch=ALL-UNNAMED
```




Exécutez à nouveau le code. Vous devrez voir afficher le dataframe :

```
24/05/28 18:03:17 INFO DAGScheduler: Job 1 finished: show at Main.scala:17, took 0,133002 s
24/05/28 18:03:17 INFO CodeGenerator: Code generated in 25.5563 ms
+----------+------------------+------------------+------------------+------------------+------------------+---------+
|      Date|              Open|              High|               Low|             Close|         Adj Close|   Volume|
+----------+------------------+------------------+------------------+------------------+------------------+---------+
|1980-12-12|0.5133928656578064|          0.515625|0.5133928656578064|0.5133928656578064|0.4067815542221069|117258400|
|1980-12-15|0.4888392984867096|0.4888392984867096|0.4866071343421936|0.4866071343421936| 0.385558158159256| 43971200|
|1980-12-16|          0.453125|          0.453125|0.4508928656578064|0.4508928656578064| 0.3572602868080139| 26432000|
|1980-12-17|0.4620535671710968|0.4642857015132904|0.4620535671710968|0.4620535671710968| 0.3661033511161804| 21610400|
|1980-12-18|0.4754464328289032|0.4776785671710968|0.4754464328289032|0.4754464328289032|0.37671515345573425| 18362400|
|1980-12-19|0.5044642686843872|0.5066964030265808|0.5044642686843872|0.5044642686843872| 0.3997070789337158| 12157600|
|1980-12-22|0.5290178656578064|           0.53125|0.5290178656578064|0.5290178656578064| 0.4191618859767914|  9340800|
|1980-12-23|0.5513392686843872|0.5535714030265808|0.5513392686843872|0.5513392686843872| 0.4368479549884796| 11737600|
|1980-12-24|0.5803571343421936|0.5825892686843872|0.5803571343421936|0.5803571343421936| 0.4598398804664612| 12000800|
|1980-12-26|0.6339285969734192|0.6361607313156128|0.6339285969734192|0.6339285969734192| 0.5022867918014526| 13893600|
|1980-12-29|0.6428571343421936|0.6450892686843872|0.6428571343421936|0.6428571343421936|  0.509361207485199| 23290400|
|1980-12-30|0.6294642686843872|0.6294642686843872|0.6272321343421936|0.6272321343421936| 0.4969809353351593| 17220000|
|1980-12-31|0.6116071343421936|0.6116071343421936|          0.609375|          0.609375|0.48283201456069946|  8937600|
|1981-01-02|0.6160714030265808|0.6205357313156128|0.6160714030265808|0.6160714030265808| 0.4881376624107361|  5415200|
|1981-01-05|0.6049107313156128|0.6049107313156128|0.6026785969734192|0.6026785969734192| 0.4775262176990509|  8932000|
|1981-01-06|          0.578125|          0.578125|0.5758928656578064|0.5758928656578064| 0.4563027024269104| 11289600|
|1981-01-07|0.5535714030265808|0.5535714030265808|0.5513392686843872|0.5513392686843872| 0.4368479549884796| 13921600|
|1981-01-08|0.5424107313156128|0.5424107313156128|0.5401785969734192|0.5401785969734192|0.42800483107566833|  9956800|
|1981-01-09|0.5691964030265808|0.5714285969734192|0.5691964030265808|0.5691964030265808| 0.4509968459606170 7|  5376000|
|1981-01-12|0.5691964030265808|0.5691964030265808|0.5647321343421936|0.5647321343421936| 0.4474595732803345|  5924800|
+----------+------------------+------------------+------------------+------------------+------------------+---------+
only showing top 20 rows
```

Ajouter cette ligne de code :

```
df.printSchema()
```

Exécutez et voyez ce que cela affiche.

Ensuite ajouter cette ligne dans votre df :

```
option("inferSchema", value = true)
```

Qu'affiche-t-il ?

**Referencing columns**

Ajoutez cet import

```
import org.apache.spark.sql.functions.col
```

Puis ces lignes de code dans le main()

```
df.select("Date", "Open", "Close").show()
val column = df("Date")
col("Date")
import spark.implicits._
$"Date"

df.select(col("Date"), $"Open", df("Close")).show()
```

Que voyez-vous ?

**Fonctions Colonne 1**

```
import org.apache.spark.sql.types.StringType
```

```
val column = df("Open")
val newColumn = column + (2.0)
val columnString = column.cast(StringType)

df.select(column, newColumn, columnString).show()
```

Que voyez-vous et quelle est la différence entre ces 3 colonnes ?

Filtrage des colonnes

```
df.select(column, newColumn, columnString)
  .filter(newColumn > 2.0)
  .filter(newColumn > column)
  .show()
```

Même question.

Pour comparer 2 colonnes :

```
df.filter(newColumn === column)
```

On peut les comparer en ajoutant la fonction « as » pour différencier les différentes colonnes :

```
val column = df("Open")
val newColumn = (column + 2.0).as("OpenIncreasedBy2")
val columnString = column.cast(StringType).as("OpenAsString")
```

**Fonctions Colonne 2**

Concaténer

```
import org.apache.spark.sql.functions.{concat, col, lit}

val litColumn = lit(2.0)
val newColumnString = concat(columnString, lit("Hello World"))

df.select(column, newColumn, columnString, newColumnString)
  .show()
```

Qu'y a-t-il comme résultat ?

On va tronquer :

```
import org.apache.spark.sql.functions.{col, concat, lit, trunc}

df.select(column, newColumn, columnString, newColumnString)
  .show(truncate = false)
```

**SQL Expressions**

```
val timestampFromExpression = expr("cast(current_timestamp() as string) as
timestampExpression")
val timestampFromFunctions
=current_timestamp().cast(StringType).as("timestampFunctions")

df.select(timestampFromExpression, timestampFromFunctions).show()
```

Attention à l'orthographe des mots (`current_timestamp`)

```
df.selectExpr("cast(Date as string)", "Open + 1.0",
"current_timestamp()").show()
```

Exécuter des requêtes SQL via Spark

```
df.createTempView("df")
spark.sql("select * from df").show()
```

Remarque : Non recommandé, faut utiliser l'API de Scala qui est beaucoup plus puissant.

**Assignment (Affectation)**

Changer le nom des colonnes :

Méthode 1

```
df.withColumnRenamed("Open", "open")
  .withColumnRenamed("Close", "close")
```

Méthode 2

```
val renameColumns = List(
  col("Date").as("date"),
  col("Open").as("open"),
  col("Close").as("close"),
  col("High").as("high"),
  col("Low").as("low"),
```

```
  col("Adj Close").as("adjClose"),
  col("Volume").as("volume")

)

df.select(renameColumns : _*).show()
```

Méthode 3

```
df.select(df.columns.map(c => col(c).as(c.toLowerCase())): _*).show()
```

Créer une nouvelle colonne en faisant la différence entre ‘open’ et ‘close’:

```
val stockData = df.select(renameColumns: _*)
  .withColumn("diff", col("close") - col("open"))

stockData.show()
```

Filtrer le Data Frame

```
val stockData = df.select(renameColumns: _*)
  .withColumn("diff", col("close") - col("open"))
  .filter(col("close") > col("open") * 1.1)
```

**GroupBy, Sort & Aggregations**

Groupby

```
import org.apache.spark.sql.functions.{col, current_timestamp, expr, year,
max, avg}
```

```
import spark.implicits._

stockData
  .groupBy(year($"date"))
  .agg(max($"close"), avg($"close"))
  .show()
```

Essayons de mettre un nom adéquat.

```
stockData
  .groupBy(year($"date").as("year"))
  .agg(max($"close").as("maxClose"), avg($"close").as("avgClose"))
  .show()
```

Faisons un tri de ‘maxClose’ avec ‘sort’

```
stockData
  .groupBy(year($"date").as("year"))
  .agg(max($"close").as("maxClose"), avg($"close").as("avgClose"))
```

```
  .sort($"maxClose".desc)
  .show()
```

Autre manière :

```
stockData
  .groupBy(year($"date").as("year"))
  .max("close","high")
  .show()
```

**Fonctions Fenêtre (Window functions)**

```
val window =
Window.partitionBy(year($"date").as("year")).orderBy($"close".desc)
stockData
  .withColumn("rank", row_number().over(window))
  .filter($"rank" === 1)
  .sort($"close".desc)
  .show()
```

**Partitions, AST, Plan logique et Optimisations**

**Cas pratique**

```
val window =
Window.partitionBy(year($"date").as("year")).orderBy($"close".desc)
stockData
  .withColumn("rank", row_number().over(window))
  .filter($"rank" === 1)
  .sort($"close".desc)
  .explain(extended = true)
```

Résultat à obtenir :

```
== Analyzed Logical Plan ==
date: date, open: double, high: double, low: double, close: double, adjClose: double, volume: int, rank: int
Sort [close#72 DESC NULLS LAST], true
+- Filter (rank#167 = 1)
   +- Project [date#68, open#69, high#70, low#71, close#72, adjClose#73, volume#74, rank#167]
      +- Project [date#68, open#69, high#70, low#71, close#72, adjClose#73, volume#74, rank#167, rank#167]
         +- Window [row_number() windowspecdefinition(year(date#68), close#72 DESC NULLS LAST, specifiedwindowfr
            +- Project [date#68, open#69, high#70, low#71, close#72, adjClose#73, volume#74]
               +- Project [Date#17 AS date#68, Open#18 AS open#69, High#19 AS high#70, Low#20 AS low#71, Close#2
                  +- Relation [Date#17,Open#18,High#19,Low#20,Close#21,Adj Close#22,Volume#23] csv
```

Quels sont ces différents types de plan ?

**### Évaluation paresseuse (Lazy evaluation)**

Il y a des transformations et des actions.

**Transformation**:

val stockData = df.select(renameColumns: _*) .withColumn("diff", col("close") - col("open"))

Ici, stockData est une transformation. Rien n'est exécuté encore.

**Action**:

val displayData = stockData.show()

L'action .show() déclenche l'exécution. Le filtrage est effectu.

### Tests unitaires

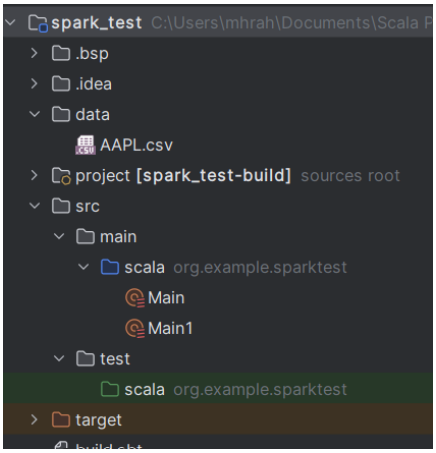
Ajouter une dépendance (scalatest) :

https://www.scalatest.org/install

Dans le fichier build.sbt :

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.18" % "test"
```

`% "test"` indique qu'on ne peut pas accéder à cette librairie directement dans notre main.




Créer une classe Scala dans le sous-dossier 'scala' de 'test'. Appelez-le 'FirstTest'

Prenez un style sur le site : https://www.scalatest.org/user_guide/selecting_a_style

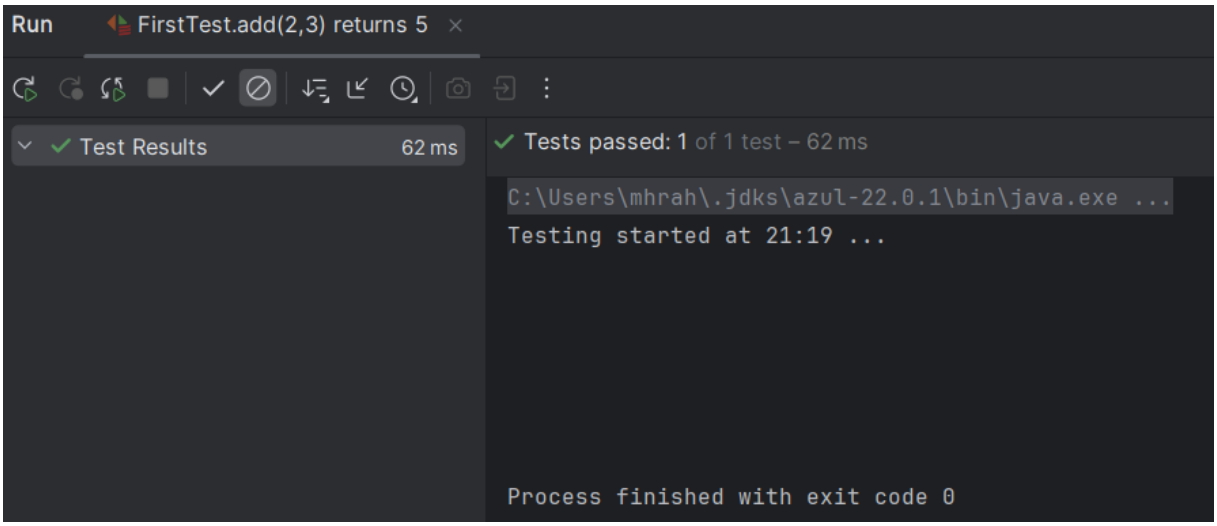
Dans le fichier Main.scala avant l'avant dernière accolade :

```
def add(x: Int, y:Int): Int = x + y
```

Testez ce code :

```
package org.example.sparktest

import org.scalatest.funsuite.AnyFunSuite

class FirstTest extends AnyFunSuite {
  test("add(2,3) returns 5"){
    val result = Main.add(2,3)
```

```
    assert(result === 5)
  }

}
```




Refaites un autre test avec cette ligne

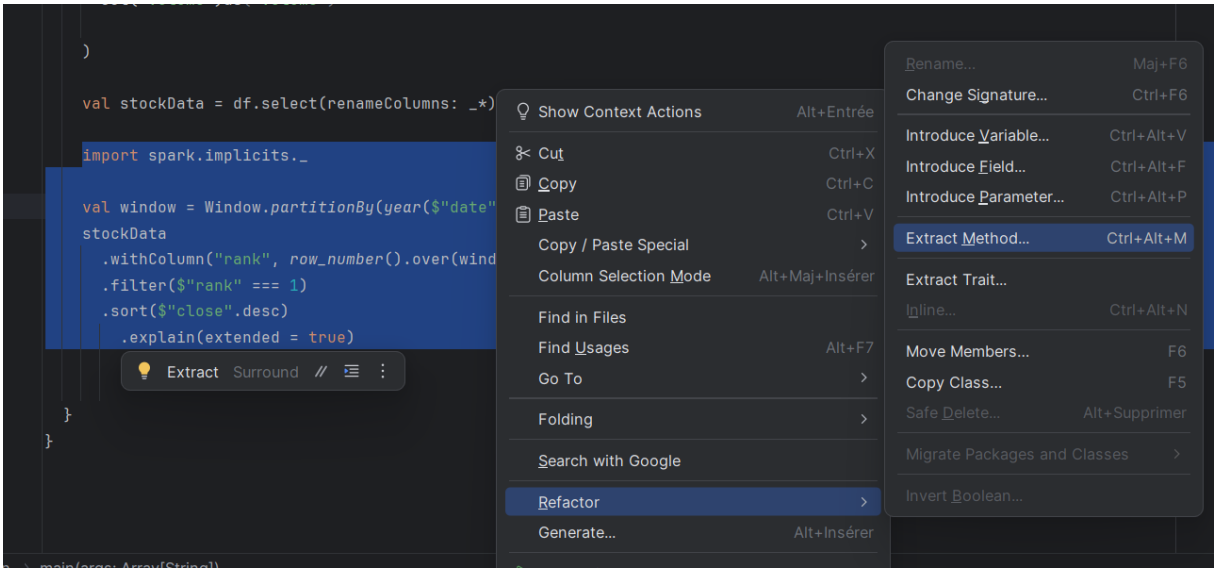
```
def add(x: Int, y:Int): Int = x + y -1
```

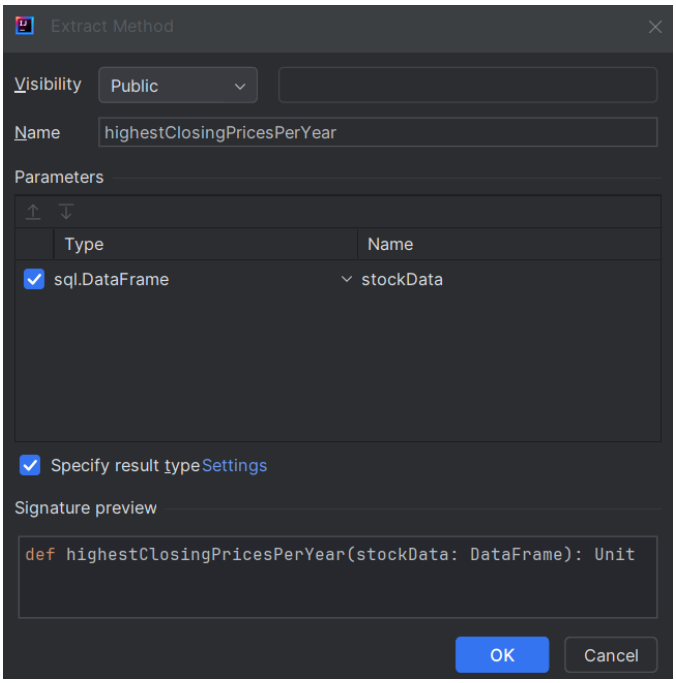
Que se passe-t-il ?

Qu'est-ce que cela implique en **principe** ?

**Autre Test**

**Extraire une méthode du code sélectionné dans le public Main**






Ecrivez ce code :

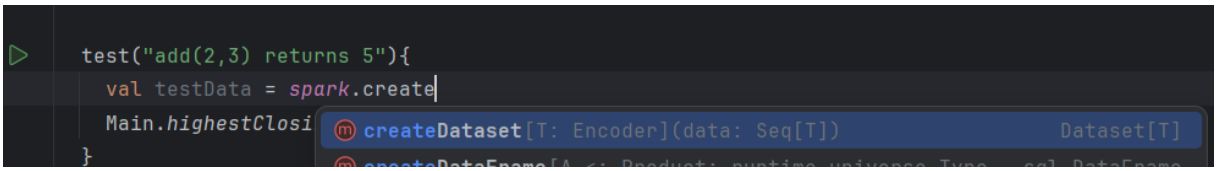
```
  val highestClosingPrices = highestClosingPricesPerYear(stockData)

}

def highestClosingPricesPerYear(df: DataFrame): DataFrame = {
  import df.sparkSession.implicits._

  val window = Window.partitionBy(year($"date").as("year")).orderBy($"close".desc)
  df
    .withColumn("rank", row_number().over(window))
    .filter($"rank" === 1)
    .sort($"close".desc)
```

Ecrivez ceci dans votre FirstTest :

```
class FirstTest extends AnyFunSuite {
  test("add(2,3) returns 5"){
    Main.highestClosingPricesPerYear()
  }
```

**Ecrire un test case pour le DataFrame**

On reprend le même code et dans le fichier test on créé le SparkSession.




Créez son schema :

```
private val schema = StructType(Seq(
  StructField("date", DateType, nullable=  true),
  StructField("open", DoubleType, nullable=  true),
  StructField("close", DoubleType, nullable=  true)
)
```

Créez son test qui va créer un df et faire le test et stocker le résultat dans result:

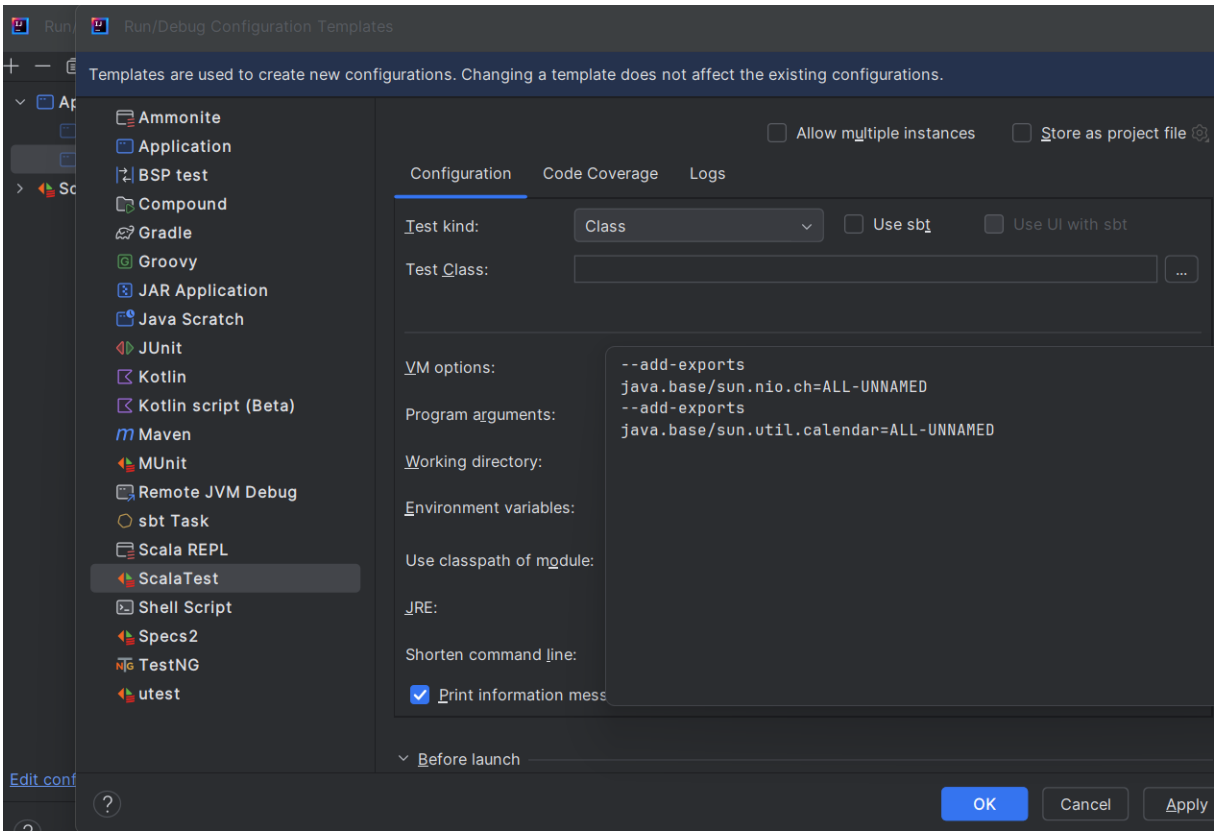
```
test("add(2,3) returns 5"){
  val testRows = Seq(
    Row(Date.valueOf("2022-01-12"), 1.0, 2.0), // open price - close price
    Row(Date.valueOf("2023-03-01"), 1.0, 2.0),
    Row(Date.valueOf("2023-01-12"), 1.0, 3.0)
  )

  implicit val encoder: Encoder[Row] = Encoders.row()
  val testDf = spark.createDataset(testRows)
  val result = Main.highestClosingPricesPerYear(testDf)
}
```

On veut vérifier si 'test' est correct. On va collecter les résultats en une liste de lignes. Il faut donc rajouter des lignes :

```
val expected = Seq(
  Row(Date.valueOf("2022-03-12"), 1.0, 2.0),
  Row(Date.valueOf("2023-01-12"), 1.0, 3.0)
)
implicit val encoder: Encoder[Row] = Encoders.row()
val testDf = spark.createDataset(testRows)
val actualRows = Main.highestClosingPricesPerYear(testDf)
  .collect()

actualRows should contain theSameElementsAs expected
//allElementsOf(expected)
```

**ATTENTION à bien écrire les dates correctement.**

Changer les configurations de scalatest (directement sans passer par les templates)




--add-exports

java.base/sun.nio.ch=ALL-UNNAMED

--add-exports

java.base/sun.util.calendar=ALL-UNNAMED

Exécutez et voyez si ça marche. Y a-t-il un problème ?

Dans Main :

```
def highestClosingPricesPerYear(df: DataFrame): DataFrame = {
  import df.sparkSession.implicits._

  val window =
Window.partitionBy(year($"date").as("year")).orderBy($"close".desc)
  df
    .withColumn("rank", row_number().over(window))
    .filter($"rank" === 1)
    .drop($"rank")
    .sort($"close".desc)
```

Le test est-il passé ?

**Data Visualisation**

Ajouter dans build.sbt la librairie breeze

```
"org.scalanlp" %% "breeze" % "2.1.0",
"org.scalanlp" %% "breeze-viz" % "2.1.0",
```

Dans Main importer la librairie puis tapez l'exemple de code ci-dessous :

```
import breeze.plot._
```

```
object Visualisation {
  def plotData(df: DataFrame): Unit = {
    val data = df.collect().map(row => (row.getAs[java.sql.Date]("date"),
row.getAs[Double]("close")))

    val f = Figure()
    val p = f.subplot(0)
    val x = data.map(_._1.getTime.toDouble)
    val y = data.map(_._2)

    p += plot(x, y)
    p.xlabel = "Date"
    p.ylabel = "Close Price"
    f.saveas("plot.png")
  }
}
val stockData = df.select(renameColumns: _*)

val highestClosingPrices = highestClosingPricesPerYear(stockData)
highestClosingPrices.show()

Visualisation.plotData(highestClosingPrices)
```

**MapReduce**

```
// Transformation Map
val mappedData = stockData.withColumn("year",
year(col("date"))).select("year", "volume")

// Transformation Reduce
val reducedData =
mappedData.groupBy("year").agg(sum("volume").as("total_volume"))

reducedData.show()
```

Qu'est-ce qu'on voit et qu'a fait le code ?