Writing and Testing Standalone Scala Spark Programs

BIG DATA 210

Autumn 2018

Introduction

In class we have generally used Spark via the spark-shell command line or a Jupyter or Zeppelin notebook. We can also write standalone Spark programs, test them locally against a Spark instance running on our development machine, and then deploy them to be run on a remote Spark cluster in production.

Writing standalone programs can make developing and debugging more complex Spark tasks, or tasks that you'd like to automate and run repeatedly, much easier than command shell or notebook-based approaches. Learning to build such programs is a worthwhile investment for the long term even though we won't rely heavily on doing so in this class.

In this tutorial we show how to build standalone Spark programs in Scala, covering in particular:

- The prerequisites you need to follow this tutorial
- The *sbt* build tool used by many Scala developers
- The *IntelliJ IDEA* integrated development environment (IDE)
- Using sbt to create a skeletal Spark Scala program from a template
- How to run the program from inside sbt and from inside IDEA
- The anatomy of the template project
- How to package the project for deployment to a remote Spark cluster
- How to submit the program for running in a remote Spark cluster

Set Up and Prerequisites

To follow these instructions, you'll need to install a Java Development Kit (JDK), the sbt build tool and the IntelliJ IDEA interactive development environment. The following sections tell you where to find each of these things.

Installing a Java Development Kit

You can download and install the JDK for your platform from:

https://www.oracle.com/technetwork/java/javase/downloads/index.html

You should use at least Java 8.

TASK: *Install a JDK on your development machine.*

Installing the sbt Build Tool

You can obtain the sbt build tool from a variety of places, including the sbt website at https://www.scala-sbt.org

The sbt Getting Started Guide (https://www.scala-sbt.org/1.x/docs/Getting-Started.html) is worth bookmarking and skimming.

For Windows users we suggest installing via the MSI installer package available from: https://piccolo.link/sbt-1.2.6.msi

Mac users have a few installation options including Macports and Homebrew, as well as ZIP and TGZ distributions. You can find them here: https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Mac.html If you already use Macports or Homebrew, I suggest using them to install sbt.

TASK: *Install sbt on your development machine using the appropriate method documented in the section above.*

Installing IntelliJ IDEA

You can obtain IntelliJ IDEA from its publisher, JetBrains, at: https://www.jetbrains.com/idea/features/editions_comparison_matrix.html

The *Community Edition* should be sufficient for our purposes and is free and open source. The *Ultimate Edition* has a lot of extra stuff that make it good for serious day to day development use and worth bugging your employer to expense.

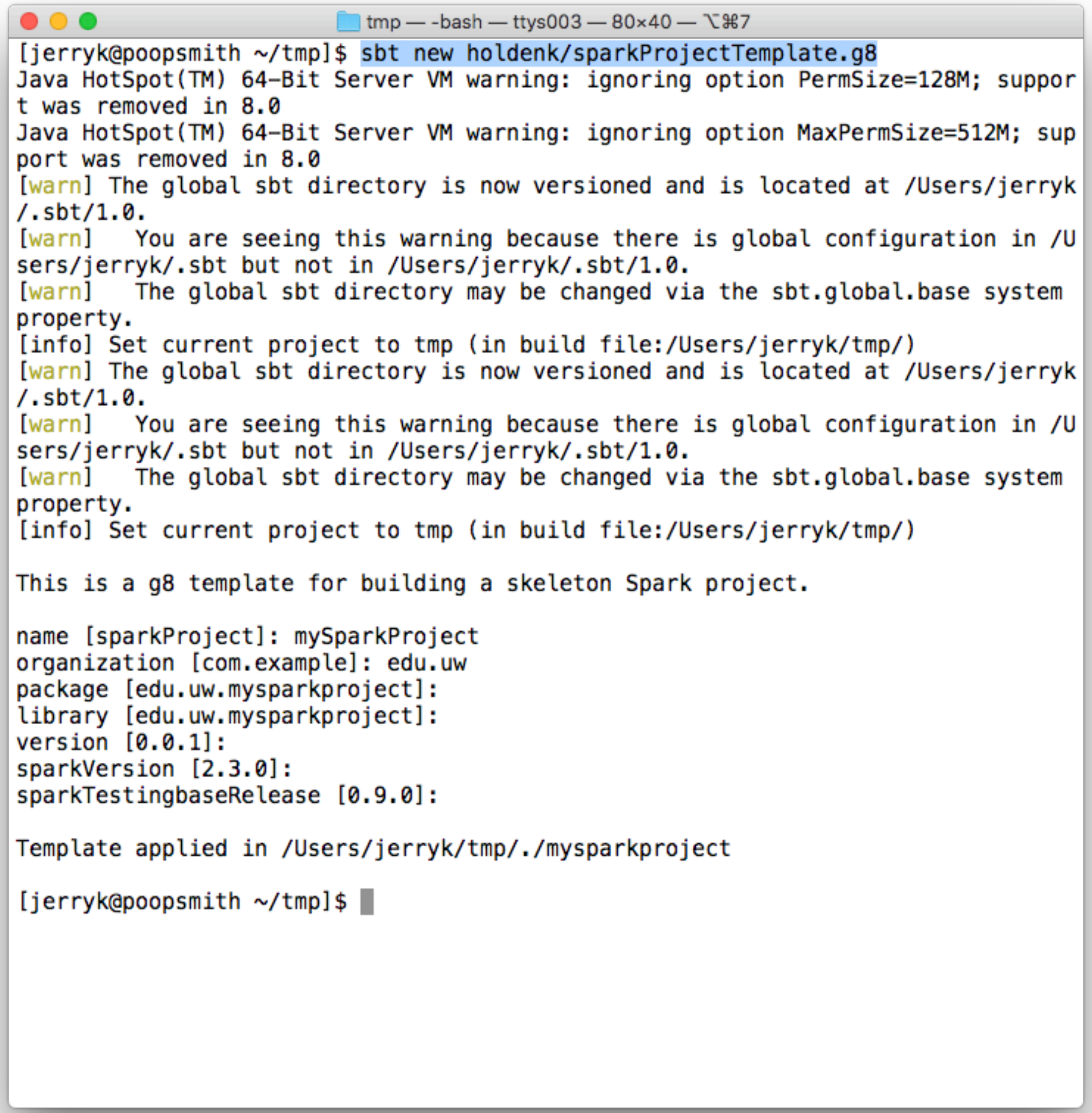
TASK: *Install IntelliJ IDEA on your development machine.*

Creating a Template Scala Spark Project

Now that you've installed a JDK, sbt and IDEA, you're ready to create a starter project.

One of the features sbt provides is the ability to generate stub starter projects of various types by consulting a remote repository of project types. You can generate such a template project using the 'sbt new' command and specifying the name of the desired template. We'll use the *holdenk/sparkProjectTemplate.g8* template, which was created by a developer at Databricks, the commercial vendor behind the Spark open source project.

TASK: *From a command prompt issue the 'sbt new holdenk/sparkProjectTemplate.g8' command as shown in the screenshot. Accept the defaults for the version prompts. If you change the project name or package, remember what you chose when working through the rest of this tutorial.*



```
[jerryk@poopsmith ~/tmp]$ sbt new holdenk/sparkProjectTemplate.g8
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128M; suppor
t was removed in 8.0
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512M; sup
port was removed in 8.0
[warn] The global sbt directory is now versioned and is located at /Users/jerryk
/.sbt/1.0.
[warn]   You are seeing this warning because there is global configuration in /U
sers/jerryk/.sbt but not in /Users/jerryk/.sbt/1.0.
[warn]   The global sbt directory may be changed via the sbt.global.base system
property.
[info] Set current project to tmp (in build file:/Users/jerryk/tmp/)
[warn] The global sbt directory is now versioned and is located at /Users/jerryk
/.sbt/1.0.
[warn]   You are seeing this warning because there is global configuration in /U
sers/jerryk/.sbt but not in /Users/jerryk/.sbt/1.0.
[warn]   The global sbt directory may be changed via the sbt.global.base system
property.
[info] Set current project to tmp (in build file:/Users/jerryk/tmp/)

This is a g8 template for building a skeleton Spark project.

name [sparkProject]: mySparkProject
organization [com.example]: edu.uw
package [edu.uw.mysparkproject]:
library [edu.uw.mysparkproject]:
version [0.0.1]:
sparkVersion [2.3.0]:
sparkTestingbaseRelease [0.9.0]:

Template applied in /Users/jerryk/tmp/./mysparkproject

[jerryk@poopsmith ~/tmp]$
```

Figure 1: Generating a Template Project with sbt

Now cd into the directory that sbt created for your project. Observe the following artifacts generated from the template.

- *build.sbt*: This is an sbt *build file*, which tells sbt about your project's structure and dependencies, as well as how to build it. This file is written in a domain specific language implemented on top of Scala. Its contents are valid Scala code, making use of functions and overloaded operators provided by sbt itself. For what we're doing now you won't have to understand much of the file's contents but it's worth looking at the *libraryDependencies* section which tells sbt which libraries your project relies on. As you write more elaborate programs you might find yourself adding new dependencies.

- *project directory*: This directory contains some additional settings sbt uses. You probably won't have to edit it very often.
- *src*: This directory contains the source code for your program and its unit tests in the *main* and *test* subdirectories respectively.

Importing your Project into IntelliJ IDEA

Now we'll import our sbt-generated project into IntelliJ. Upon starting IntelliJ, you should see a dialog like the following.

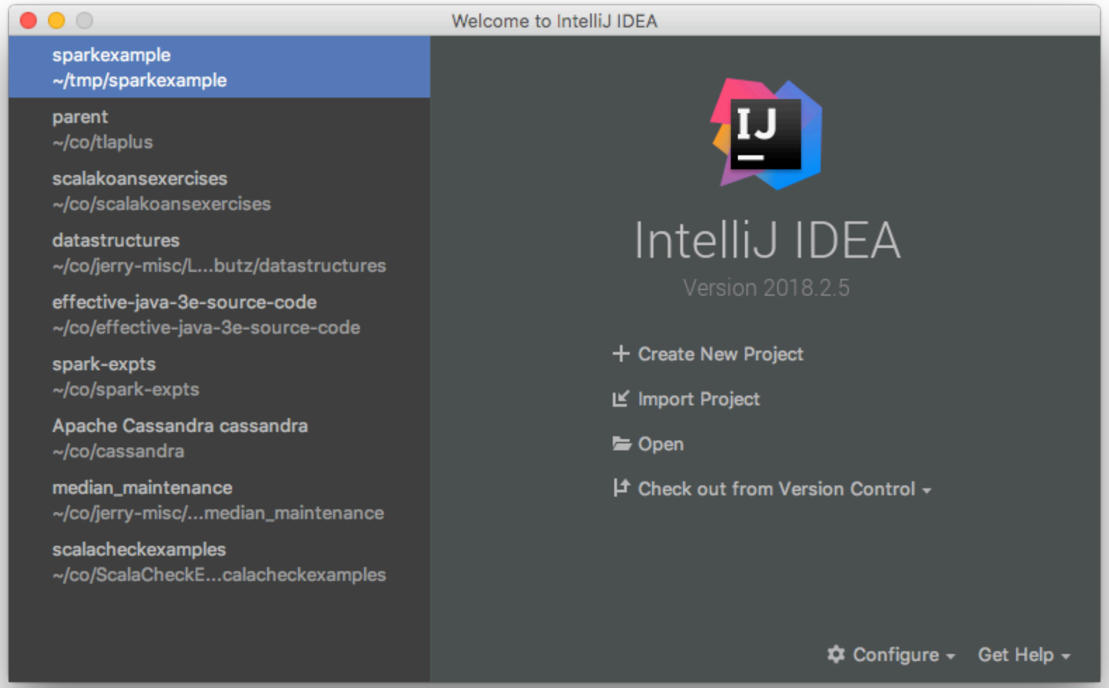
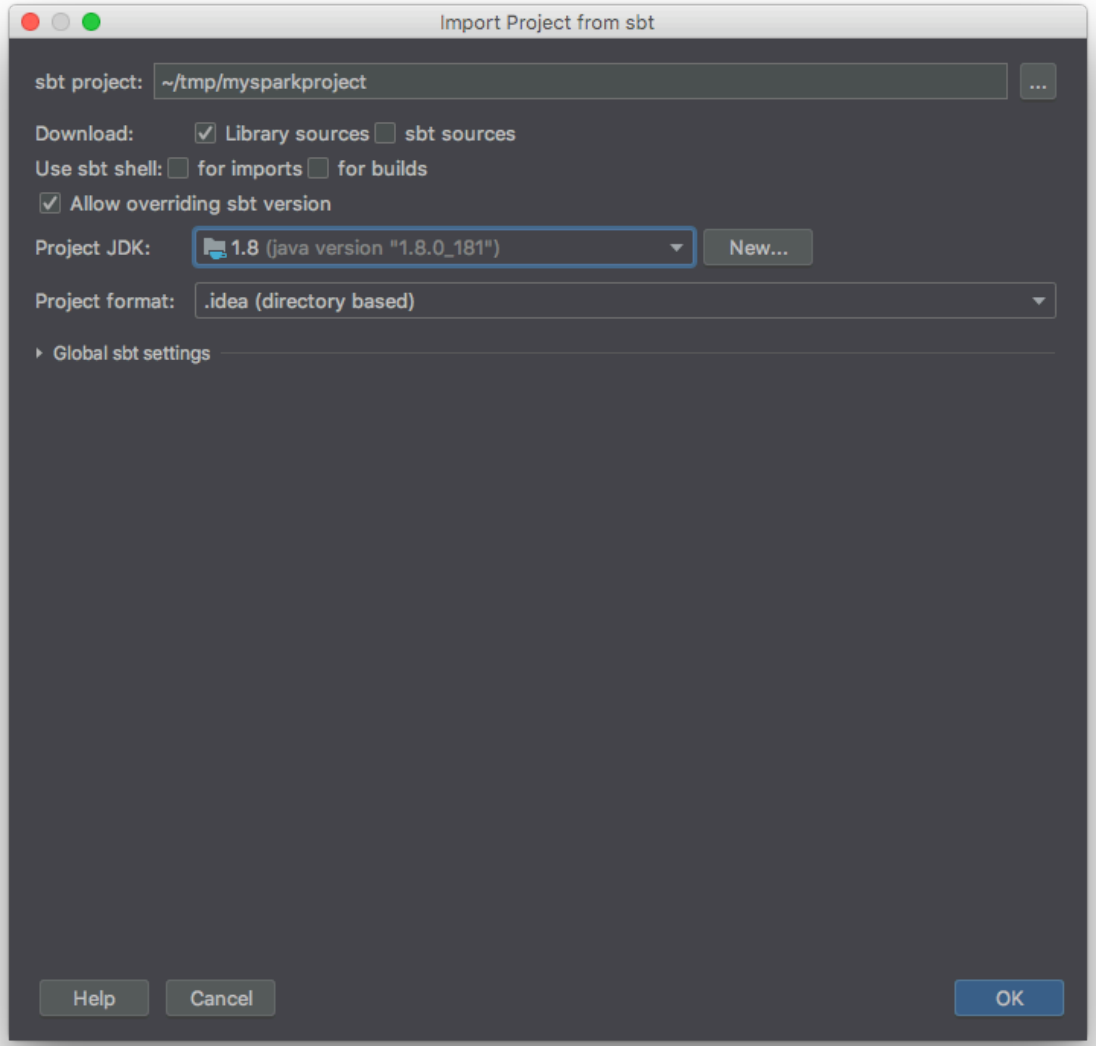


Figure 2: Importing an sbt Project into IntelliJ

TASK: *Start IntelliJ, select 'Import Project', navigate to where your sbt-generated template lives, and double click the build.sbt file. In the 'Import Project from sbt' dialog that appears, make sure the project JDK is set to the one you downloaded and click OK.*



sbt project: ~/tmp/mysparkproject

Download: [✓ Library sources] sbt sources

Use sbt shell: for imports for builds

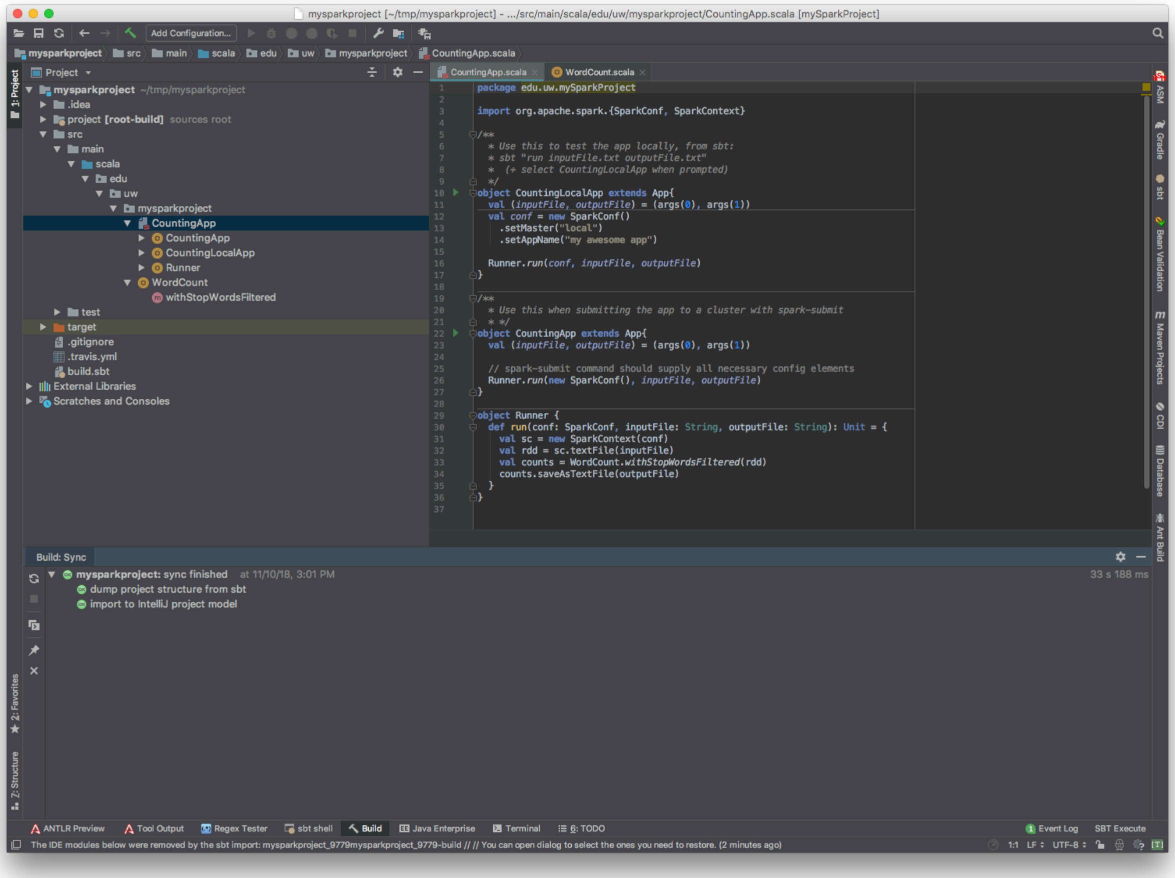
[✓ Allow overriding sbt version]

Project JDK: 1.8 (java version "1.8.0_181")

Project format: .idea (directory based)

Figure 3: Import project from sbt dialog

After you've done the above IntelliJ will run for a while with *"sbt: dump project structure from sbt"* displayed in the status bar at the bottom of the window. After it finishes you'll see the project structure tree on the left side of the IntelliJ window is filled in and you can navigate through the project's folders to see your sources under *src/main/scala*. Click your way to *CountingApp.*



```scala
package edu.uw.mySparkProject

import org.apache.spark.{SparkConf, SparkContext}

/**
  * Use this to test the app locally, from sbt:
  * sbt "run inputFile.txt outputFile.txt"
  *  (+ select CountingLocalApp when prompted)
  */
object CountingLocalApp extends App{
  val (inputFile, outputFile) = (args(0), args(1))
  val conf = new SparkConf()
    .setMaster("local")
    .setAppName("my awesome app")

  Runner.run(conf, inputFile, outputFile)
}

/**
  * Use this when submitting the app to a cluster with spark-submit
  * */
object CountingApp extends App{
  val (inputFile, outputFile) = (args(0), args(1))

  // spark-submit command should supply all necessary config elements
  Runner.run(new SparkConf(), inputFile, outputFile)
}

object Runner {
  def run(conf: SparkConf, inputFile: String, outputFile: String): Unit = {
    val sc = new SparkContext(conf)
    val rdd = sc.textFile(inputFile)
    val counts = WordCount.withStopWordsFiltered(rdd)
    counts.saveAsTextFile(outputFile)
  }
}
```

```
mysparkproject: sync finished   at 11/10/18, 3:01 PM        33 s 188 ms
    dump project structure from sbt
    import to IntelliJ project model
```

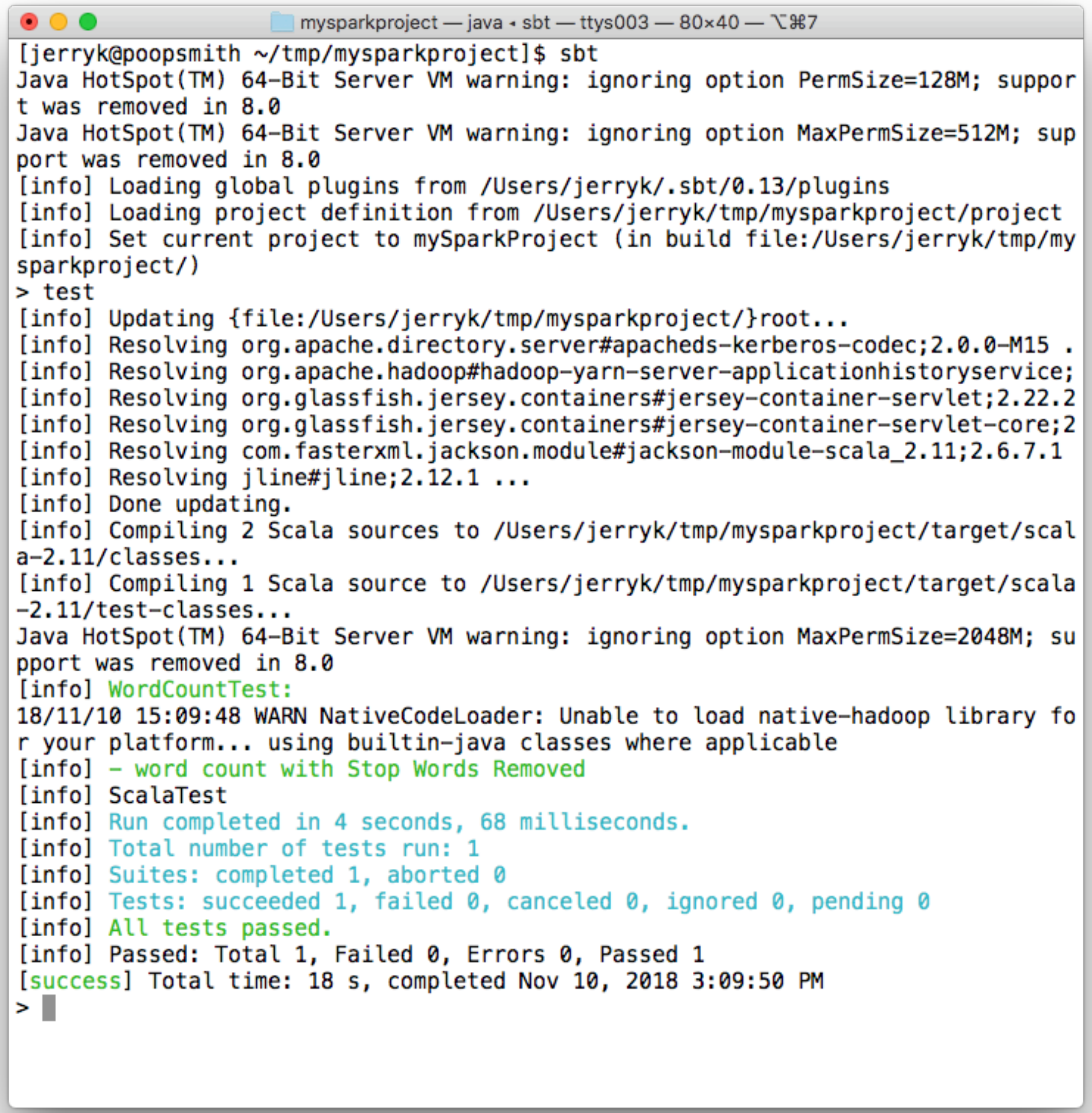
```
The IDE modules below were removed by the sbt import: mysparkproject_9779mysparkproject_9779-build // // You can open dialog to select the ones you need to restore. (2 minutes ago)
```

Figure 4: CountingApp source code in IntelliJ

Running the Starter Program and Tests from sbt

Now, we'll see how to run the project from sbt.

In your template project directory, run sbt. You'll be greeted by an sbt prompt. Typing *help* will show the commands the sbt shell accepts. To get started, run the template project's unit tests by typing *test* at the sbt prompt. You should see sbt compile the sources and print out the test results. They should pass as shown in the screenshot.



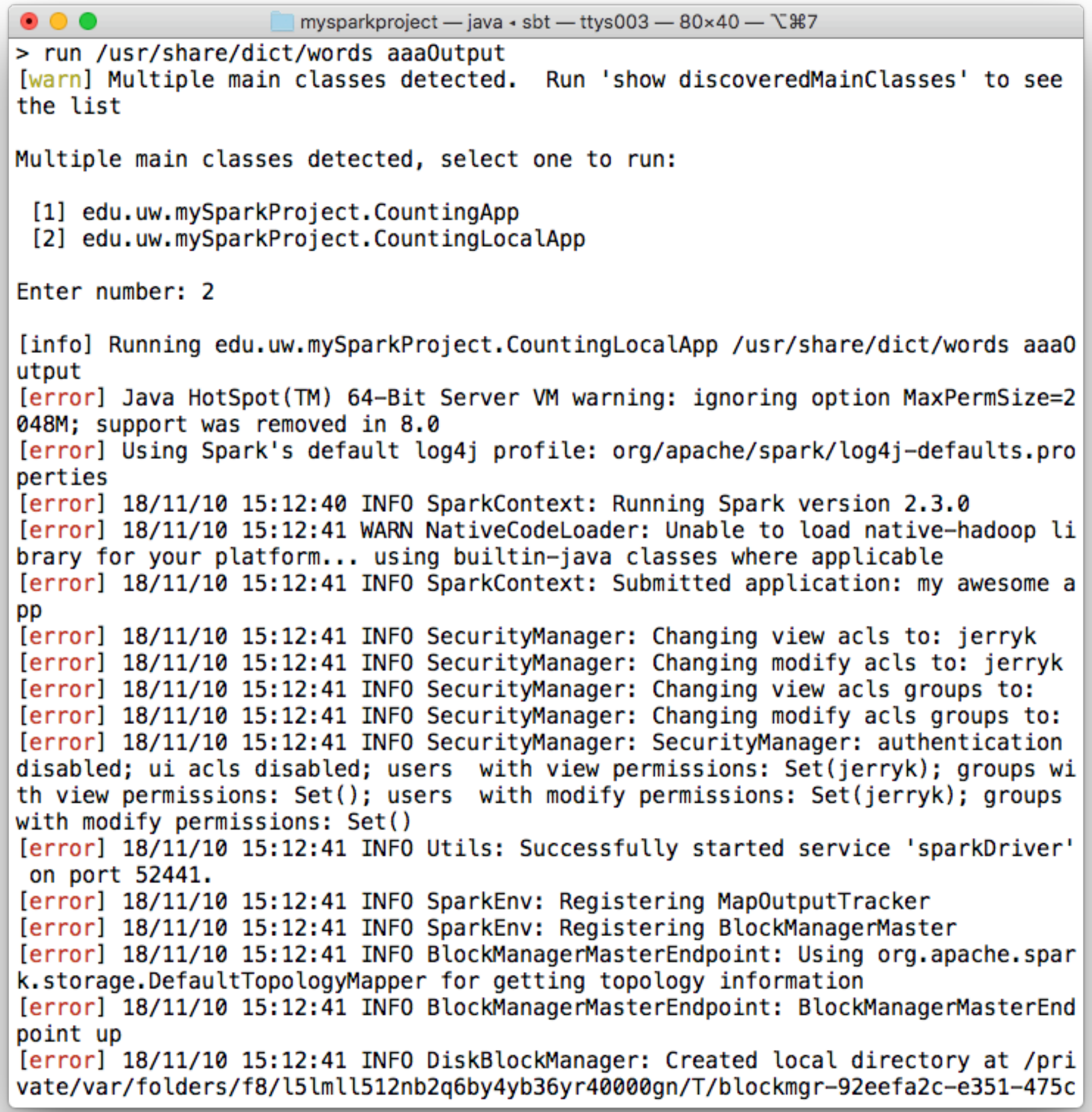
```
[jerryk@poopsmith ~/tmp/mysparkproject]$ sbt
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128M; suppor
t was removed in 8.0
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512M; sup
port was removed in 8.0
[info] Loading global plugins from /Users/jerryk/.sbt/0.13/plugins
[info] Loading project definition from /Users/jerryk/tmp/mysparkproject/project
[info] Set current project to mySparkProject (in build file:/Users/jerryk/tmp/my
sparkproject/)
> test
[info] Updating {file:/Users/jerryk/tmp/mysparkproject/}root...
[info] Resolving org.apache.directory.server#apacheds-kerberos-codec;2.0.0-M15 .
[info] Resolving org.apache.hadoop#hadoop-yarn-server-applicationhistoryservice;
[info] Resolving org.glassfish.jersey.containers#jersey-container-servlet;2.22.2
[info] Resolving org.glassfish.jersey.containers#jersey-container-servlet-core;2
[info] Resolving com.fasterxml.jackson.module#jackson-module-scala_2.11;2.6.7.1
[info] Resolving jline#jline;2.12.1 ...
[info] Done updating.
[info] Compiling 2 Scala sources to /Users/jerryk/tmp/mysparkproject/target/scal
a-2.11/classes...
[info] Compiling 1 Scala source to /Users/jerryk/tmp/mysparkproject/target/scala
-2.11/test-classes...
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=2048M; su
pport was removed in 8.0
[info] WordCountTest:
18/11/10 15:09:48 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
[info] - word count with Stop Words Removed
[info] ScalaTest
[info] Run completed in 4 seconds, 68 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.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
[success] Total time: 18 s, completed Nov 10, 2018 3:09:50 PM
>
```

Figure 5: Running unit tests in sbt

Now you have a project with passing unit tests that you can run.

To run the program itself, against a local Spark instance that sbt will start for you, type *run*, followed by the name of an input text file in which you want to count words and the name of a directory where you want the output of your job to go at the sbt prompt.

In response, sbt will list all of the classes in the project that have a *Main* that can be used as an entry point. Select *CountingLocalApp* when prompted, as shown in the screenshot.



```
> run /usr/share/dict/words aaaOutput
[warn] Multiple main classes detected.  Run 'show discoveredMainClasses' to see
the list

Multiple main classes detected, select one to run:

 [1] edu.uw.mySparkProject.CountingApp
 [2] edu.uw.mySparkProject.CountingLocalApp

Enter number: 2

[info] Running edu.uw.mySparkProject.CountingLocalApp /usr/share/dict/words aaaO
utput
[error] Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=2
048M; support was removed in 8.0
[error] Using Spark's default log4j profile: org/apache/spark/log4j-defaults.pro
perties
[error] 18/11/10 15:12:40 INFO SparkContext: Running Spark version 2.3.0
[error] 18/11/10 15:12:41 WARN NativeCodeLoader: Unable to load native-hadoop li
brary for your platform... using builtin-java classes where applicable
[error] 18/11/10 15:12:41 INFO SparkContext: Submitted application: my awesome a
pp
[error] 18/11/10 15:12:41 INFO SecurityManager: Changing view acls to: jerryk
[error] 18/11/10 15:12:41 INFO SecurityManager: Changing modify acls to: jerryk
[error] 18/11/10 15:12:41 INFO SecurityManager: Changing view acls groups to:
[error] 18/11/10 15:12:41 INFO SecurityManager: Changing modify acls groups to:
[error] 18/11/10 15:12:41 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users  with view permissions: Set(jerryk); groups wi
th view permissions: Set(); users  with modify permissions: Set(jerryk); groups
with modify permissions: Set()
[error] 18/11/10 15:12:41 INFO Utils: Successfully started service 'sparkDriver'
 on port 52441.
[error] 18/11/10 15:12:41 INFO SparkEnv: Registering MapOutputTracker
[error] 18/11/10 15:12:41 INFO SparkEnv: Registering BlockManagerMaster
[error] 18/11/10 15:12:41 INFO BlockManagerMasterEndpoint: Using org.apache.spar
k.storage.DefaultTopologyMapper for getting topology information
[error] 18/11/10 15:12:41 INFO BlockManagerMasterEndpoint: BlockManagerMasterEnd
point up
[error] 18/11/10 15:12:41 INFO DiskBlockManager: Created local directory at /pri
vate/var/folders/f8/l5lmll512nb2q6by4yb36yr40000gn/T/blockmgr-92eefa2c-e351-475c
```

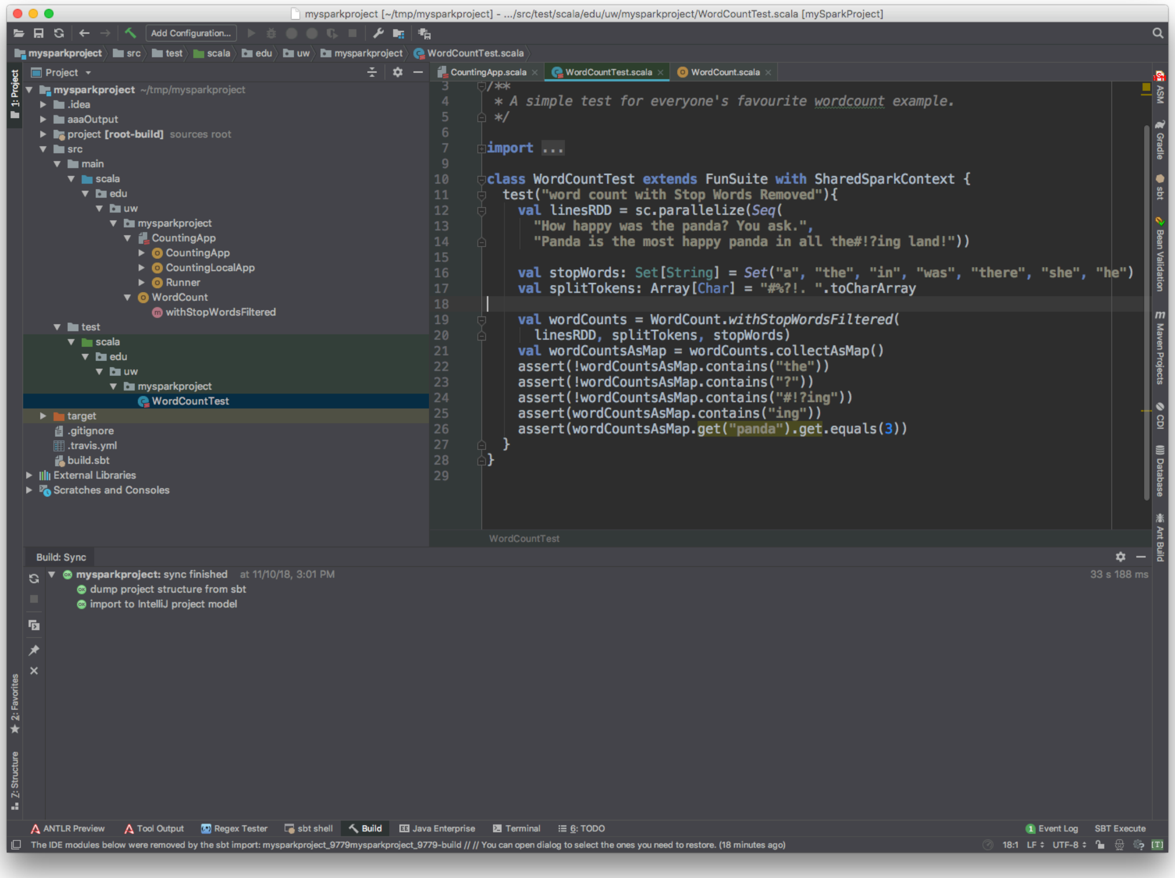
Figure 6: Running CountingLocalApp in sbt

A local Spark instance will be started, and you should see *success* printed in green at the end of the run. If you don't see success, then scroll back and check the output for errors. After your job succeeds, go look at the output directory you specified and inspect its contents.

Running the Starter Program and Tests from IntelliJ

Running our program and tests from sbt is quick and efficient, but sometimes we might want to run it from the IDE, especially if we want to use the debugger to single step through the program. Now, let's learn how to do that.

In IntelliJ, navigate to the *WordCountTest* class in the project structure. Open it in the editor by double clicking on it.



```
/**
 * A simple test for everyone's favourite wordcount example.
 */

import ...

class WordCountTest extends FunSuite with SharedSparkContext {
  test("word count with Stop Words Removed"){
    val linesRDD = sc.parallelize(Seq(
      "How happy was the panda? You ask.",
      "Panda is the most happy panda in all the#!?ing land!"))

    val stopWords: Set[String] = Set("a", "the", "in", "was", "there", "she", "he")
    val splitTokens: Array[Char] = "#%?!. ".toCharArray

    val wordCounts = WordCount.withStopWordsFiltered(
      linesRDD, splitTokens, stopWords)
    val wordCountsAsMap = wordCounts.collectAsMap()
    assert(!wordCountsAsMap.contains("the"))
    assert(!wordCountsAsMap.contains("?"))
    assert(!wordCountsAsMap.contains("#!?ing"))
    assert(wordCountsAsMap.contains("ing"))
    assert(wordCountsAsMap.get("panda").get.equals(3))
  }
}
```

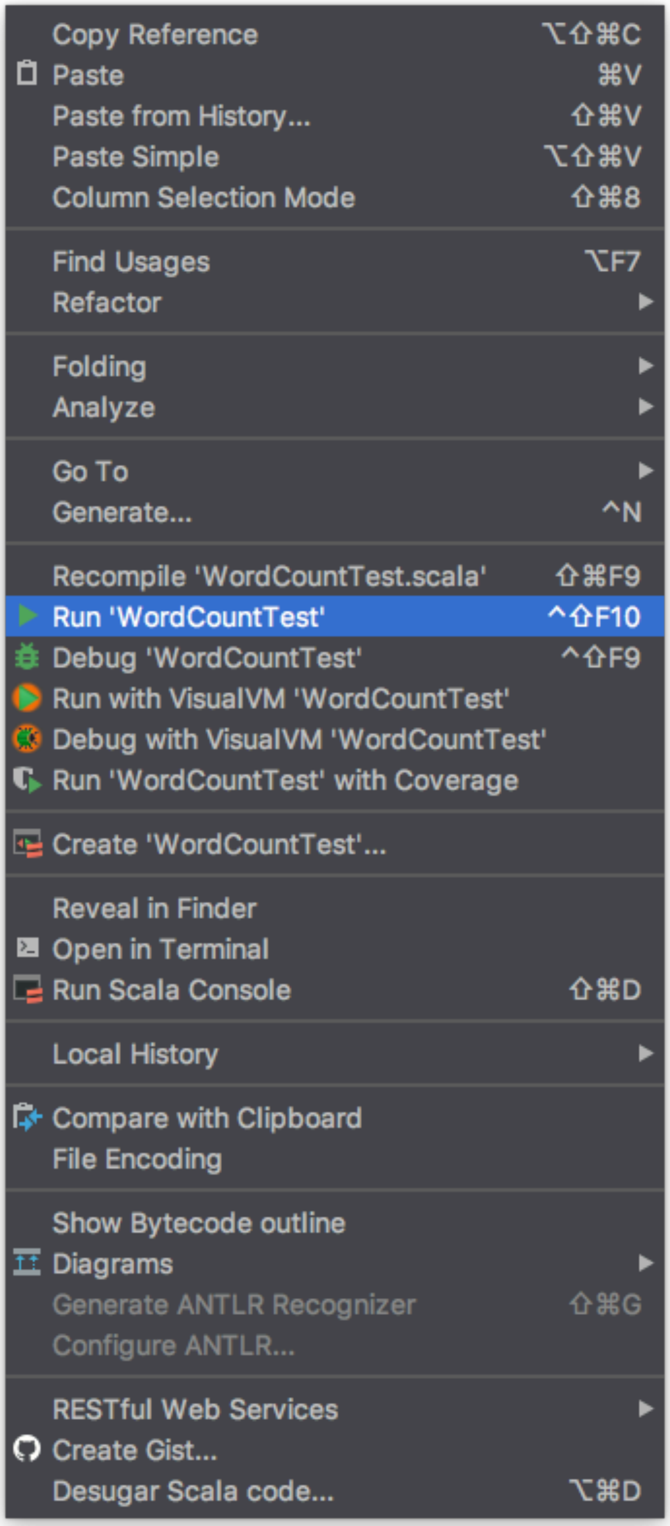
```
mysparkproject: sync finished   at 11/10/18, 3:01 PM     33 s 188 ms
    dump project structure from sbt
    import to IntelliJ project model
```

```
The IDE modules below were removed by the sbt import: mysparkproject_9779mysparkproject_9779-build // // You can open dialog to select the ones you need to restore. (18 minutes ago)
```

Figure 7: The WordCountTest source code

This file is a unit test implemented using the ScalaTest[1] framework. If you've used JUnit, NUnit or a similar framework in the past you get the general idea. To run the test, right click on its name, *WordCountTest*, and click *Run* in the menu that pops up.

[1] You can learn more about ScalaTest at http://www.scalatest.org/



Copy Reference ⌥⇧⌘C
Paste ⌘V
Paste from History... ⇧⌘V
Paste Simple ⌥⇧⌘V
Column Selection Mode ⇧⌘8
Find Usages ⌥F7
Refactor
Folding
Analyze
Go To
Generate... ^N
Recompile 'WordCountTest.scala' ⇧⌘F9
Run 'WordCountTest' ^⇧F10
Debug 'WordCountTest' ^⇧F9
Run with VisualVM 'WordCountTest'
Debug with VisualVM 'WordCountTest'
Run 'WordCountTest' with Coverage
Create 'WordCountTest'...
Reveal in Finder
Open in Terminal
Run Scala Console ⇧⌘D
Local History
Compare with Clipboard
File Encoding
Show Bytecode outline
Diagrams
Generate ANTLR Recognizer ⇧⌘G
Configure ANTLR...
RESTful Web Services
Create Gist...
Desugar Scala code... ⌥⌘D

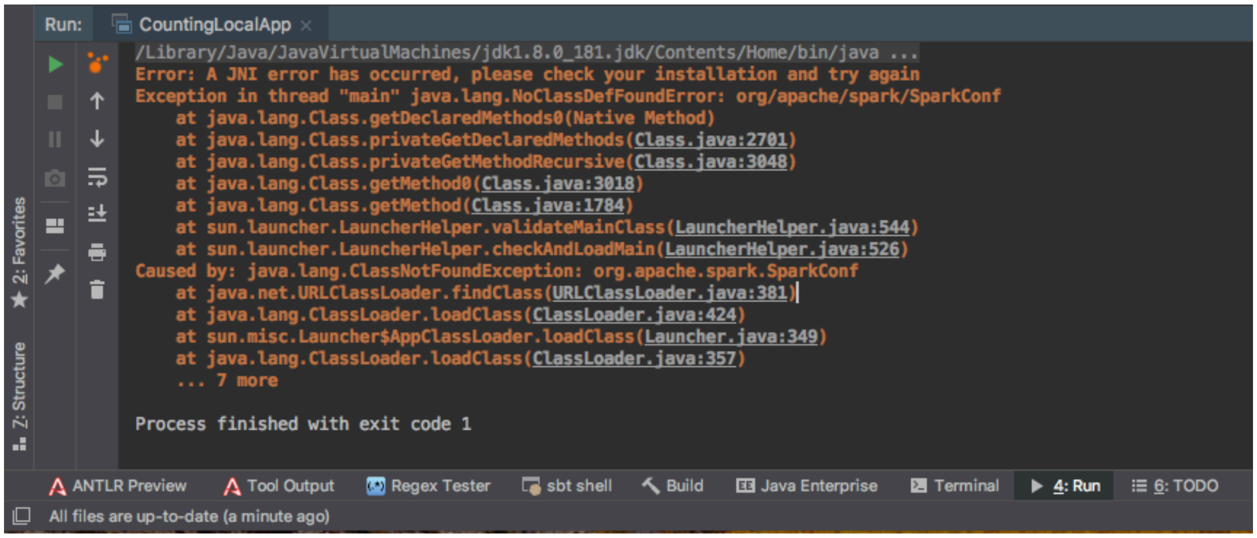
Figure 8: The Run Menu

IntelliJ will compile the test and the code under test, run it, and if all goes well you should see *All Tests Passed* displayed in the run window that appears at the bottom of the IDE.

Now you've run tests in IntelliJ as well as in sbt!

Next, we'll see how to run the tests from the IDE. We'd want to do this if, for example, we planned to use the debugger to single step or inspect variables in a running program.

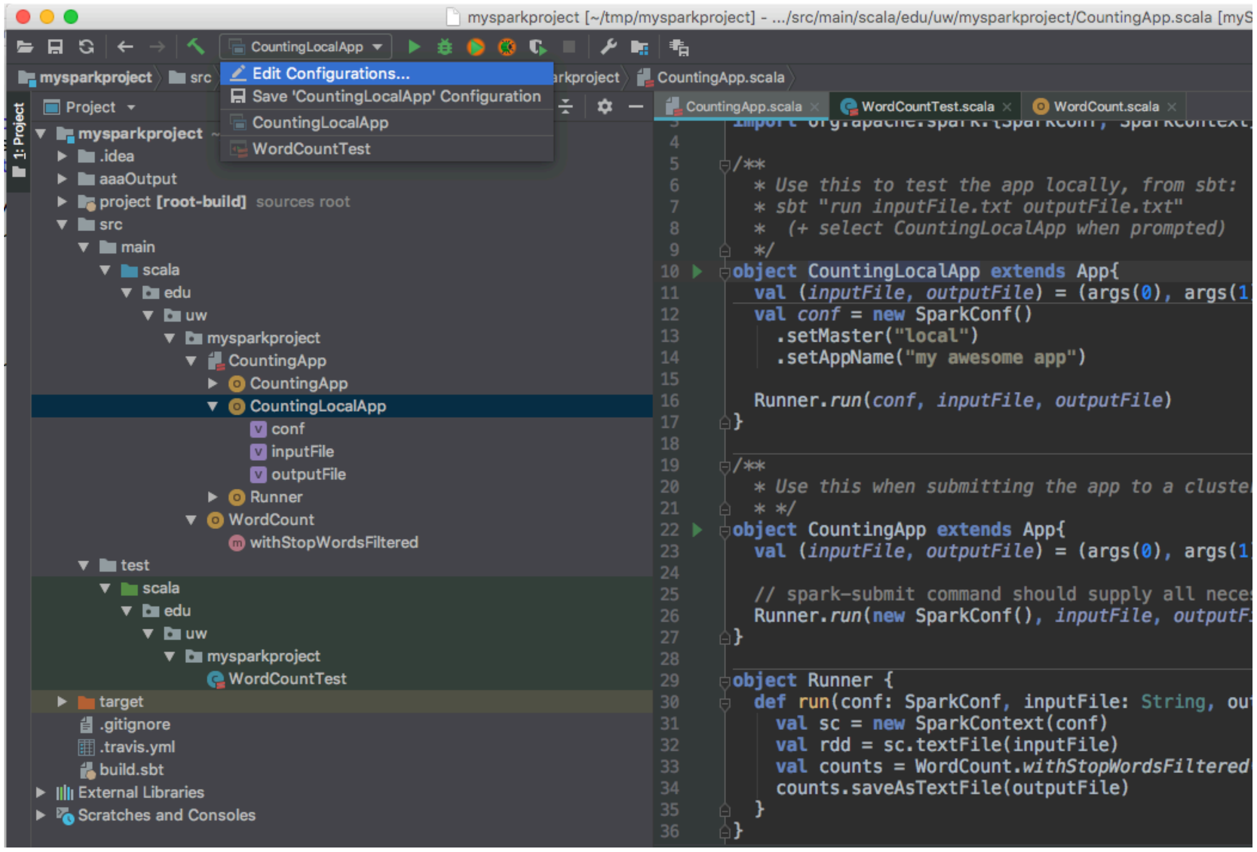
Navigate back to *CountingLocalApp*. Right click the name of the class and select *Run* from the popup menu like you did for the unit test. Woe befalls you. Before rending your garments look at the error output.



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/spark/SparkConf
    at java.lang.Class.getDeclaredMethods0(Native Method)
    at java.lang.Class.privateGetDeclaredMethods(Class.java:2701)
    at java.lang.Class.privateGetMethodRecursive(Class.java:3048)
    at java.lang.Class.getMethod0(Class.java:3018)
    at java.lang.Class.getMethod(Class.java:1784)
    at sun.launcher.LauncherHelper.validateMainClass(LauncherHelper.java:544)
    at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:526)
Caused by: java.lang.ClassNotFoundException: org.apache.spark.SparkConf
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 7 more

Process finished with exit code 1
```

Figure 9: A not quite right IntelliJ run configuration

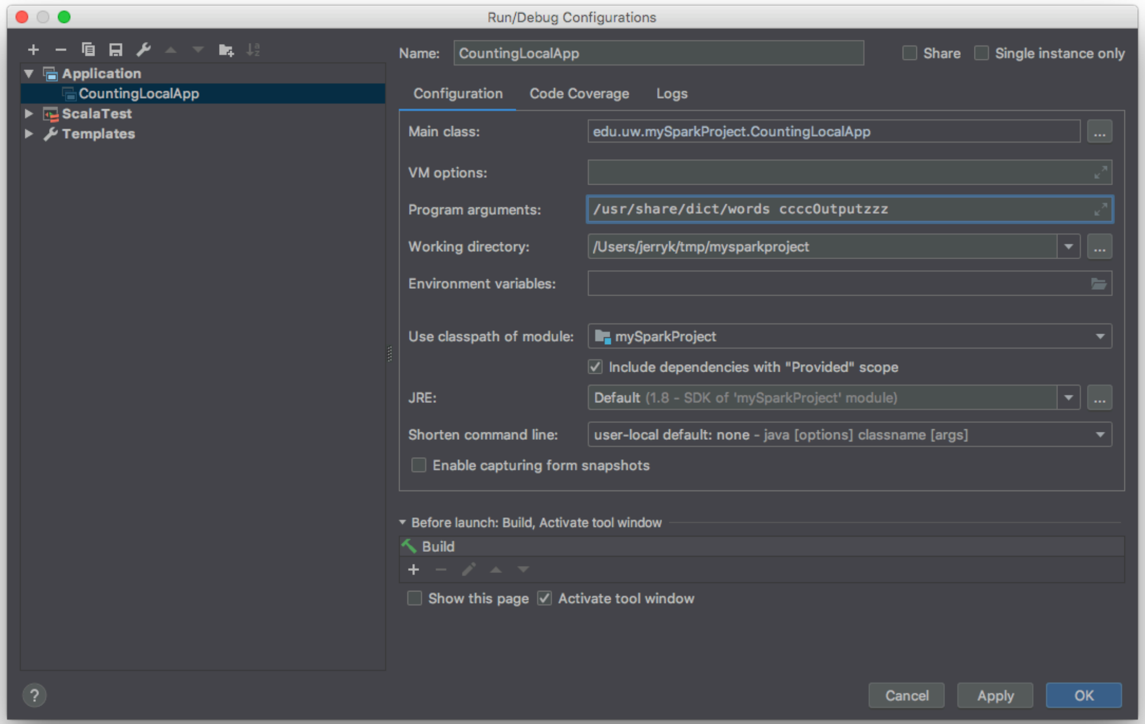
To fix this error, go to the toolbar at the top of the IntelliJ window and select *Edit Configuration* from the run dropdown where *CountingLocalApp*'s name should still be displayed.


mysparkproject [~/tmp/mysparkproject] - .../src/main/scala/edu/uw/mysparkproject/CountingApp.scala
CountingLocalApp
Edit Configurations...
Save 'CountingLocalApp' Configuration
CountingLocalApp
WordCountTest
CountingApp.scala
WordCountTest.scala
WordCount.scala

```
import org.apache.spark.{SparkConf, SparkContext}

/**
 * Use this to test the app locally, from sbt:
 * sbt "run inputFile.txt outputFile.txt"
 *  (+ select CountingLocalApp when prompted)
 */
object CountingLocalApp extends App{
  val (inputFile, outputFile) = (args(0), args(1
  val conf = new SparkConf()
    .setMaster("local")
    .setAppName("my awesome app")

  Runner.run(conf, inputFile, outputFile)
}

/**
 * Use this when submitting the app to a cluste
 * */
object CountingApp extends App{
  val (inputFile, outputFile) = (args(0), args(1

  // spark-submit command should supply all neces
  Runner.run(new SparkConf(), inputFile, outputF
}

object Runner {
  def run(conf: SparkConf, inputFile: String, ou
    val sc = new SparkContext(conf)
    val rdd = sc.textFile(inputFile)
    val counts = WordCount.withStopWordsFiltered
    counts.saveAsTextFile(outputFile)
  }
}
```

Figure 10: Editing a run configuration in IntelliJ

In the *Run/Debug Configurations* dialog that appears, check the *Include dependencies with "Provided" scope* box and click *OK*. While you're here, also fill in the *Program arguments* with the name of the text file you want to count words in and the name of the directory you'd like your output to appear in. Now try running the application again.


Run/Debug Configurations

Name: CountingLocalApp

Share ☐ Single instance only ☐

Configuration | Code Coverage | Logs

Main class: edu.uw.mySparkProject.CountingLocalApp

VM options:

Program arguments: /usr/share/dict/words ccccOutputzzz

Working directory: /Users/jerryk/tmp/mysparkproject

Environment variables:

Use classpath of module: mySparkProject

✓ Include dependencies with "Provided" scope

JRE: Default (1.8 - SDK of 'mySparkProject' module)

Shorten command line: user-local default: none - java [options] classname [args]

☐ Enable capturing form snapshots

Before launch: Build, Activate tool window

Build

☐ Show this page ✓ Activate tool window

Figure 11: IntelliJ Run/Debug Configurations dialog

If all has gone well, things should succeed! You now know how to create a Run/Debug configuration that lets you run the program locally in the IDE for debugging purposes.

The Anatomy of a Standalone Scala Spark Program

Now we'll dissect the template project and explain the key Scala and Spark elements it uses.

Open *CountingApp.scala* in the IDE's editor. At the top of the file there's a *package* declaration identifying what package the file's contents belong to. The package declaration means the same thing it does in Java[2]. Below the package declaration are some imports, in this case of classes that belong to the Spark framework, including the *SparkContext* that you remember from working with spark-shell.

CountingApp.scala defines three Scala *singleton objects*. A singleton object is a class that has exactly one instance[3]. We intend these classes to represent the entry points for an entire program, so it makes sense for them to be singletons.

[2] See https://docs.oracle.com/javase/tutorial/java/package/createpkgs.html

[3] People often create singleton objects in Java by making constructors private. Scala supports and enforces the notion as an intrinsic part of the language.

Now let's look more closely at these three singleton classes, namely *CountingLocalApp*, *CountingApp* and *Runner*.

CountingLocalApp is what we ran from sbt and from IntelliJ. Notice that it and *CountingApp* both extend a Scala *trait*[4] called *App.*

The *App*[5] trait is part of the Scala standard library and can be used to quickly turn an object into an executable program. Recall that to make a Java program executable it has to provide a public static void method called *main* that takes an array of Strings representing the program's command line arguments. When we extend *App* in Scala, the curly-brace delimited block that immediately follows *App* becomes the entry point of a standalone program whose arguments will be found in an array of Strings called *args*.

The first line of *CountingLocalApp* gets the 0-th and 1-st arguments (these are what we previously passed in as program parameters when running from sbt or IntelliJ) and wraps them in a pair (or 2-tuple). Next, the program creates a *SparkConf* setting the master and application name for our program. Finally, it passes that configuration, as well as the captured program arguments to the *run* method on the *Runner* object, about which we'll say more in a moment.

At this point, pause to contrast *CountingApp* with *CountingLocalApp*. The only meaningful difference is that for *CountingLocalApp* we explicitly constructed a SparkConf and set some things on it before passing it to *Runner.run()* whereas in *CountingApp* we created a *SparkConf* and didn't customize it. We do this because *CountingApp* is intended to serve as an entry point when we submit the job to a non-local Spark cluster for execution. In that case the SparkConf will have its defaults loaded from system properties and the classpath[6].

Now, turn your attention to the *Runner* object. It contains a *run* method which both of our App objects call. The run method takes a SparkConf and the paths for our input and output files. It creates a new SparkContext using the given configuration (just as spark-shell does on startup) and then the work of our program begins. In the template example we create an RDD by using the SparkContext to read a text file. Then we call *WordCount.withStopWordsFiltered()* which produces some new RDDs via flatMap, filter, map and reduceByKey similar to examples we've done in class. Take a look at the *WordCount.withStopWordsFiltered()* method and convince yourself you understand what it's doing.

[4] A Scala *trait* is similar to a Java interface. Classes and objects can extend traits, but traits can't be instantiated by themselves. A trait may provide implementations of methods it defines, something Java interfaces couldn't do before Java 8. For more details see https://docs.scala-lang.org/tour/traits.html

[5] See https://www.scala-lang.org/api/current/scala/App.html for App's full documentation.

[6] IntelliJ tip: the *Quick Documentation* function is very handy when you want to look up the documentation of some code you're using. You can find it under IntelliJ's *View* menu. The keyboard shortcut for it varies between operating systems but it's Ctrl+J on Mac OS X.

Now, you've seen what the framing and guts of a standalone Spark Scala program look like. When you have some time you might want to try writing a new one, by following the pattern established by *CountingApp* and *CountingLocalApp*.

Running your Spark Scala Program on a Real Cluster

To run out standalone program on a Spark cluster somewhere else, we need to be able to do two things:

- Package the program and its relevant dependencies into an artifact that we can send off to the cluster
- Submit the packaged artifact to the cluster for execution

Generating the Packaged Program with sbt

For a project descended from our template producing a package is easy. What we want is a so-called *assembly*, or *fat jar,* or *uber jar*[7]*.* Such a JAR file contains our program as well as dependencies specified in our build.sbt file.

To create the deployable fat jar, execute *assembly* in sbt.

After it runs you should find the newly generated file:

```
target/scala-2.11/mySparkProject-assembly-0.0.1.jar
```

The above is the JAR file we'll run in our sandbox.

Submitting the Packaged Program to a Cluster

The first thing we need to do is get the JAR into our Azure VM. We can do this with SSH (put in your username and Azure VM IP address where appropriate and mind the trailing colon):

```
scp target/scala-2.11/mySparkProject-assembly-0.0.1.jar \
jerryk@52.175.196.171:
```

Now SSH into your Azure VM. You should find the JAR sitting in your home directory. We need to scp it down into the sandbox, so from your Azure VM (note the *capital* 'P' and trailing colon):

```
scp -P 2222 mySparkProject-assembly-0.0.1.jar root@localhost:
```

[7] A java JAR file is an archive file that contains a collection of files representing the bytecode of Java classes and data files they may depend on. You can learn more at https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html

Now check in your sandbox's bash shell that the fat JAR is present in /root.

Now, it's time to run it! At your sandbox bash prompt:

```
spark-submit --class edu.uw.mySparkProject.CountingApp \
     --master yarn \
     --deploy-mode cluster \
     /root/mySparkProject-assembly-0.0.1.jar \
     file:///data/war_and_peace.txt \
     zzzMyOutputGonnaGoHereInHDFS
```

The class argument is the fully qualified Java class name of the object we want to run. We also have to give the path to the JAR file that contains our program as well as the arguments the program demands, in this case, the input file and the name of the directory we want the output to land in.

If it runs to completion look in HDFS at your specified output path to try and find the results. Remember that you ran spark-submit as root, so you'll expect to find your results in HDFS under */user/root*.

If the job failed, look at the terminal output. Common things to get wrong are invalid paths, specifying an output directory that already exists, or getting the class name wrong.

You have now taken a standalone Spark Scala program and submitted it to run in your sandbox!

Conclusion

If you've worked through this tutorial successfully you've done a bunch of stuff. You got sbt and IntelliJ IDEA working and used sbt to generate a skeletal Spark program. You learned how to run the program locally from both sbt and IntelliJ, and how to submit it for remote execution in your sandbox.

Don't be discouraged if the above steps seem complicated. There are a lot of moving parts, but for many types of Spark work you'll find you use them the same way over and over again, so what we've learned here can be used as a springboard for your own work. You can backtrack and learn more about any piece of the tooling or ecosystem as you need to, or as the interest strikes you. There's no need to feel like you've mastered each of these pieces all the way down the stack in just one sitting, but if you can follow the above, you can surely branch out and deepen your understanding of all of these things in the future.