Hands-On Exercise: Explore RDDs Using the Spark Shell

Files and Data Used in This Exercise

Exercise directory:	`$DEVSH/exercises/spark-shell`
Data files (local):	`$DEVDATA/frostroad.txt`
	`$DEVDATA/weblogs/*`

In this exercise, you will use the Spark shell to work with RDDs.

You will start by viewing and bookmarking the Spark documentation in your browser. Then you will start the Spark shell and read a simple text file into a Resilient Distributed Data Set (RDD). Finally, you will copy the weblogs data set to HDFS and use RDDs to transform the data.

Viewing the Spark Documentation

1. Start Firefox in your Virtual Machine and visit the Spark documentation on your local machine using the provided bookmark.

2. From the **Programming Guides** menu, select the **Spark Programming Guide**. Briefly review the guide. You may wish to bookmark the page for later review.

3. From the **API Docs** menu, select either **Scala** or **Python**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer you to this documentation.

Starting the Spark Shell

You may choose to do the remaining steps in this exercise using either Scala or Python. Follow the instructions below for Python, or skip to the next section for Scala.

Note: Instructions for Python are provided in **blue**, while instructions for Scala are in **red**.

Starting the Python Spark Shell

Follow these instructions if you are using Python to complete this exercise. Otherwise, skip this section and continue with Starting the Scala Spark Shell.

4. In a terminal window, start the `pyspark` shell:

```
$ pyspark
```

You may get several `INFO` and `WARNING` messages, which you can disregard. If you don't see the `In[n]>` prompt after a few seconds, press `Enter` a few times to clear the screen output.

5. Spark creates a `SparkContext` object for you called `sc`. Make sure the object exists:

```
pyspark> sc
```

> **Note on Shell Prompt**
>
> To help you keep track of which shell is being referenced in the instructions, the prompt will be shown here as either **`pyspark>`** or **`scala>`**. The actual prompt will vary depending on which version of Python or Scala you are using and what command number you are on.

Pyspark will display information about the `sc` object such as

```
<pyspark.context.SparkContext at 0x2724490>
```

6. Using command completion, you can see all the available SparkContext methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.

7. You can exit the shell by pressing `Ctrl+D` or by typing `exit`. However, stay in the shell for now to complete the remainder of this exercise.

Starting the Scala Spark Shell

Follow these instructions if you are using Scala to complete this exercise. Otherwise, skip this section and continue with Reading and Displaying a Text File. (Don't try to run both a Scala and a Python shell at the same time; doing so will cause errors and slow down your machine.)

8. In a terminal window, start the Scala Spark shell:

```
$ spark-shell
```

You may get several `INFO` and `WARNING` messages, which you can disregard. If you don't see the `scala>` prompt after a few seconds, press `Enter` a few times to clear the screen output.

9. Spark creates a `SparkContext` object for you called `sc`. Make sure the object exists:

```
scala> sc
```

> **Note on Shell Prompt**
>
> To help you keep track of which shell is being referenced in the instructions, the prompt will be shown here as either **`pyspark>`** or **`scala>`**. The actual prompt will vary depending on which version of Python or Scala you are using and which command number you are on.

Scala will display information about the `sc` object such as:

```
res0: org.apache.spark.SparkContext =
org.apache.spark.SparkContext@2f0301fa
```

10. Using command completion, you can see all the available SparkContext methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.

11. You can exit the shell at any time by typing `sys.exit` or pressing `Ctrl+D`. However, stay in the shell for now to complete the remainder of this exercise.

Reading and Displaying a Text File (Python or Spark)

12. Review the simple text file you will be using by viewing (without editing) the file in a text editor in a separate window (not the Spark shell). The file is located at: `$DEVDATA/frostroad.txt`.

13. Define an RDD to be created by reading in the test file on the local filesystem. Use the first command if you are using Python, and the second one if you are using Scala. (You only need to complete the exercises in Python *or* Scala. Do not attempt to run both shells at the same time; it will result in error messages and slow down your machine.)

```
pyspark> myrdd =

sc.textFile(\  "file:/home/training/train
ing_materials/\ data/frostroad.txt")
```

```
scala> val myrdd =

sc.textFile(  "file:/home/training/training_materials/data/fros
troad.txt")
```

- **Note:** In subsequent instructions, both Python and Scala commands will be shown but not noted explicitly; Python shell commands are in blue and preceded with `pyspark>`, and Scala shell commands are in red and preceded with `scala>`.

14. Spark has not yet read the file. It will not do so until you perform an operation on the RDD. Try counting the number of lines in the dataset:

```
pyspark> myrdd.count()
```

```
scala> myrdd.count()
```

The `count` operation causes the RDD to be materialized (created and populated). The number of lines (23) should be displayed, for example:

`Out[2]: 23` (Python) *or*
`res1: Long = 23` (Scala)

15. Try executing the `collect` operation to display the data in the RDD. Note that this returns and displays the entire dataset. This is convenient for very small RDDs like this one, but be careful using `collect` for more typical large datasets.

```
pyspark> myrdd.collect()
```

```
scala> myrdd.collect()
```

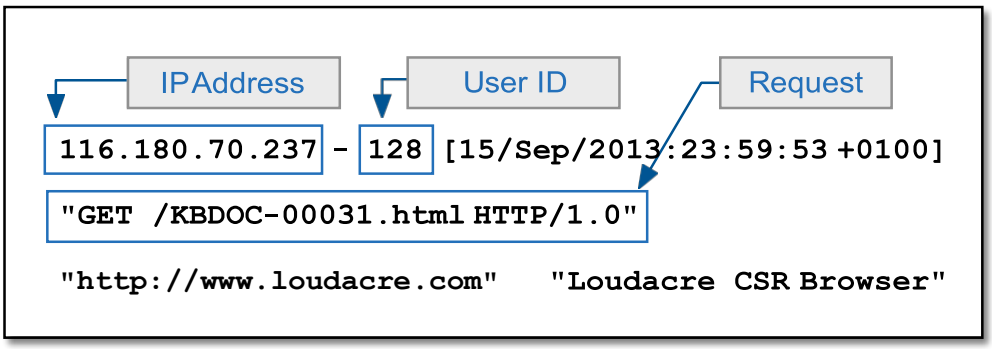
16. Using command completion, you can see all the available transformations and operations you can perform on an RDD. Type `myrdd.` and then the [TAB] key.

A Tip for PySpark Users: Controlling Log Messages

You may have noticed that by default, PySpark displays many log messages tagged `INFO`. If you find this output distracting, you may temporarily override the default logging level by using the command: `sc.setLogLevel("WARN")`. You can return to the prior level of logging with `sc.setLogLevel("INFO")` or by restarting the PySpark shell. Configuring logging will be covered later in the course.

Exploring the Loudacre Web Log Files

17. In this section you will be using data in `~/training_materials/data/weblogs`. Review one of the `.log` files in the directory. Note the format of the lines:



18. In the previous steps you used a data file residing on the local Linux filesystem. In the real world, you will almost always be working with distributed data, such as files stored on the HDFS cluster, instead. Create an HDFS directory called `loudacre` for the course (if you have not done so yet) and then copy the dataset from the local filesystem to the newly created HDFS directory. In a separate terminal window (not your Spark shell) execute:

```
$ hdfs dfs -mkdir /loudacre
$ hdfs dfs -put \
  ~/training_materials/data/weblogs/ /loudacre/
```

19. In the Spark shell, set a variable for the data files so you do not have to retype the path each time.

```
pyspark> logfiles="/loudacre/weblogs/*"
```

```
scala> val logfiles="/loudacre/weblogs/*"
```

20. Create an RDD from the data file.

```
pyspark> logsRDD = sc.textFile(logfiles)
```

```
scala> val logsRDD = sc.textFile(logfiles)
```

21. Create an RDD containing only those lines that are requests for JPG files.

```
pyspark> jpglogsRDD=\
logsRDD.filter(lambda line: ".jpg" in line)
```

```
scala> val jpglogsRDD=
logsRDD.filter(line => line.contains(".jpg"))
```

22. View the first 10 lines of the data using `take`:

```
pyspark> jpglogsRDD.take(10)
```

```
scala> jpglogsRDD.take(10)
```

23. Sometimes you do not need to store intermediate objects in a variable, in which case you can combine the steps into a single line of code. For instance, execute this single command to count the number of JPG requests. (The correct number is 64978.)

```
pyspark> sc.textFile(logfiles).filter(lambda line: \
 ".jpg" in line).count()
```

```
scala> sc.textFile(logfiles).
  filter(line => line.contains(".jpg")).count()
```

24. Now try using the `map` function to define a new RDD. Start with a simple map that returns the length of each line in the log file.

```
pyspark> logsRDD.map(lambda line: len(line)).take(5)
```

```
scala> logsRDD.map(line => line.length).take(5)
```

This prints out an array of five integers corresponding to the first five lines in the file. (The correct result is: 151, 143, 154, 147, 160.)

25. That is not very useful. Instead, try mapping to an array of words for each line:

```
pyspark> logsRDD \
  .map(lambda line: line.split(' ')).take(5)
```

```
scala> logsRDD.map(line => line.split(' ')).take(5)
```

This time Spark prints out five arrays, each containing the words in the corresponding log file line.

26. Now that you know how `map` works, define a new RDD containing just the IP addresses from each line in the log file. (The IP address is the first "word" in each line.)

```
pyspark> ipsRDD = \
  logsRDD.map(lambda line: line.split(' ')[0])
pyspark> ipsRDD.take(5)
```

```
scala> val ipsRDD =
  logsRDD.map(line =>line.split(' ')(0))
scala> ipsRDD.take(5)
```

27. Although `take` and `collect` are useful ways to look at data in an RDD, their output is not very readable. Fortunately, though, they return arrays, which you can iterate through:

```
pyspark> for ip in ipsRDD.take(10): print ip
```

```
scala> ipsRDD.take(10).foreach(println)
```

28. Finally, save the list of IP addresses as a text file:

```
pyspark> ipsRDD.saveAsTextFile("/loudacre/iplist")
```

```
scala> ipsRDD.saveAsTextFile("/loudacre/iplist")
```

- **Note:** If you re-run this command, you will not be able to save to the same directory because it already exists. Be sure to delete the directory using

either the `hdfs` command (in a separate terminal window) or the Hue file browser first.

29. In a terminal window or the Hue file browser, list the contents of the `/loudacre/iplist` folder. You should see multiple files, including several `part-xxxxx` files, which are the files containing the output data. "Part" (partition) files are numbered because there may be results from multiple tasks running on the cluster. Review the contents of one of the files to confirm that they were created correctly.

Bonus Exercise

Use RDD transformations to create a dataset consisting of the IP address and corresponding user ID for each request for an HTML file. (Disregard requests for other file types). The user ID is the third field in each log file line.

Display the data in the form *ipaddress/userid*, such as:

```
165.32.101.206/8
100.219.90.44/102
182.4.148.56/173
246.241.6.175/45395
175.223.172.207/4115
…
```

This is the end of the exercise

Hands-On Exercise: Process Data Files with Apache Spark

Files and Data Used in This Exercise:

Data files (local):	`$DEVDATA/activations/*`
	`$DEVDATA/devicestatus.txt` (Bonus)
Stubs:	`ActivationModels.pyspark`
	`ActivationModels.scalaspark`

In this exercise, you will parse a set of activation records in XML format to extract the account numbers and model names.

One of the common uses for Spark is doing data Extract/Transform/Load operations. Sometimes data is stored in line-oriented records, like the web logs in the previous exercise, but sometimes the data is in a multi-line format that must be processed as a whole file. In this exercise, you will practice working with file-based instead of line-based formats.

Important: This exercise depends on a previous exercise: "Access HDFS with the Command Line and Hue." If you did not complete that exercise, run the course catch- up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Reviewing the API Documentation for RDD Operations

1. Visit the Spark API page you bookmarked previously. Follow the link for the `RDD` class and review the list of available operations. (In the Scala API, the link will be near the top of the main window; in Python scroll down to the Core Classes area.)

Reviewing the Data

2. Review the data on the local Linux filesystem in the directory `$DEVDATA/activations`. Each XML file contains data for all the devices activated by customers during a specific month.

   Sample input data:

```
<activations>
   <activation timestamp="1225499258" type="phone">
      <account-number>316</account-number>
      <device-id>
          d61b6971-33e1-42f0-bb15-aa2ae3cd8680
      </device-id>
      <phone-number>5108307062</phone-number>
      <model>iFruit 1</model>
   </activation>
   …
</activations>
```

3. Copy the entire `activations` directory to `/loudacre` in HDFS.

```
$ hdfs dfs -put $DEVDATA/activations /loudacre/
```

Processing the Files

Follow the steps below to write code to go through a set of activation XML files and extract the account number and device model for each activation, and save the list to a file as `account_number:model`.

The output will look something like:

```
1234:iFruit     1
987:Sorrento F00L
4566:iFruit 1

…
```

4. Start with the `ActivationModels` stub script in the exercise directory: `$DEVSH/exercises/spark-etl`. (A stub is provided for Scala and Python; use whichever language you prefer.) Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this exercise. Copy the stub code into the Spark shell of your choice.

5. Use `wholeTextFiles` to create an RDD from the activations dataset. The resulting RDD will consist of tuples, in which the first value is the name of the file, and the second value is the contents of the file (XML) as a string.

6. Each XML file can contain many activation records; use `flatMap` to map the contents of each file to a collection of XML records by calling the provided `getActivations` function. `getActivations` takes an XML string, parses it, and returns a collection of XML records; `flatMap` maps each record to a separate RDD element.

7. Map each activation record to a string in the format `account-number:model`. Use the provided `getAccount` and `getModel` functions to find the values from the activation record.

8. Save the formatted strings to a text file in the directory `/loudacre/account-models`.

Bonus Exercise

If you have more time, attempt the following extra bonus exercise:

Another common part of the ETL process is data scrubbing. In this bonus exercise, you will process data in order to get it into a standardized format for later processing.

Review the contents of the file `$DEVDATA/devicestatus.txt`. This file contains data collected from mobile devices on Loudacre's network, including device ID, current status, location, and so on. Because Loudacre previously acquired other mobile providers' networks, the data from different subnetworks has a different format. Note that the records in this file have different field delimiters: some use commas, some use pipes (`|`), and so on. Your task is the following:

A. Upload the `devicestatus.txt` file to HDFS.

B. Determine which delimiter to use (hint: the character at position 19 is the first use of the delimiter).

C. Filter out any records which do not parse correctly (hint: each record should have exactly 14 values).

D. Extract the date (first field), model (second field), device ID (third field), and latitude and longitude (13th and 14th fields respectively).

E. The second field contains the device manufacturer and model name (such as `Ronin S2`). Split this field by spaces to separate the manufacturer from the model (for example, manufacturer `Ronin`, model `S2`). Keep just the manufacturer name.

F. Save the extracted data to comma-delimited text files in the `/loudacre/devicestatus_etl` directory on HDFS.

G. Confirm that the data in the file(s) was saved correctly.

The solutions to the bonus exercise are in `$DEVSH/exercises/spark- etl/solution/bonus`.

This is the end of the exercise

Hands-On Exercise: Use Pair RDDs to Join Two Datasets

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-pairs`

Data files (HDFS): `/loudacre/weblogs/*`
`/loudacre/accounts/*`

In this exercise, you will continue exploring the Loudacre web server log files, as well as the Loudacre user account data, using key-value pair RDDs.

Important: This exercise depends on two previous exercises: "Import Data from MySQL Using Apache Sqoop" and "Explore RDDs Using the Spark Shell." If you did not complete those exercises, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Exploring Web Log Files

Continue working with the web log files, as in earlier exercises.

Tip: In this exercise, you will be reducing and joining large datasets, which can take a lot of time. You may wish to perform the exercises below using a smaller dataset, consisting of only a few of the web log files, rather than all of them. Remember that you can specify a wildcard; `textFile("/loudacre/weblogs/*2.log")` would include only filenames ending with `2.log`.

1. Using map-reduce logic, count the number of requests from each user.

   a. Use `map` to create a pair RDD with the user ID as the key and the integer 1 as the value. (The user ID is the third field in each line.) Your data will look something like this:

(*userid*,1)
(*userid*,1)
(*userid*,1)
…

b. Use `reduceByKey` to sum the values for each user ID. Your RDD data will be similar to this:

(*userid*,5)
(*userid*,7)
(*userid*,2)
…

2. Use `countByKey` to determine how many users visited the site for each frequency. That is, how many users visited once, twice, three times, and so on.

a. Use `map` to reverse the key and value, like this:

(5,*userid*)
(7,*userid*)
(2,*userid*)
…

b. Use the `countByKey` action to return a map of *frequency*:*user-count* pairs.

3. Create an RDD where the user ID is the key, and the value is the list of all the IP addresses that user has connected from. (IP address is the first field in each request line.)

i. Hint: Map to `(userid,ipaddress)` and then use `groupByKey`.

(*userid*,20.1.34.55)
(*userid*,245.33.1.1)
(*userid*,65.50.196.141)
…

(*userid*,[20.1.34.55,74.125.239.98])
(*userid*,[75.175.32.10,245.33.1.1,66.79.233.99])
(*userid*,[65.50.196.141])
…

Joining Web Log Data with Account Data

Review the data located in `/loudacre/accounts` containing Loudacre's customer account data (previously imported from MySQL to HDFS using Sqoop). The first field in each line is the user ID, which corresponds to the user ID in the web server logs. The other fields include account details such as creation date, first and last name, and so on.

4. Join the accounts data with the weblog data to produce a dataset keyed by user ID which contains the user account information and the number of website hits for that user.

a. Create an RDD, based on the accounts data, consisting of key/value-array pairs: `(userid,[values…])`

(*userid1*,[*userid1*,2008-11-24 10:04:08,\N,Cheryl,West,4905 Olive Street,San Francisco,CA,…])
(*userid2*,[*userid2*,2008-11-23 14:05:07,\N,Elizabeth,Kerns,4703 Eva Pearl Street,Richmond,CA,…])
(*userid3*,[*userid3*,2008-11-02 17:12:12,2013-07-18 16:42:36,Melissa,Roman,3539 James Martin Circle,Oakland,CA,…])
…

b. Join the pair RDD with the set of user-id/hit-count pairs calculated in the first step.

```
(userid1,([userid1,2008-11-24
10:04:08,\N,Cheryl,West,4905 Olive Street,San
Francisco,CA,…],4))
(userid2,([userid2,2008-11-23
14:05:07,\N,Elizabeth,Kerns,4703    Eva    Pearl
Street,Richmond,CA,…],8))
(userid3,([userid3,2008-11-02 17:12:12,2013-07-18
16:42:36,Melissa,Roman,3539 James Martin
Circle,Oakland,CA,…],1))
…
```

c. Display the user ID, hit count, and first name (4th value) and last name (5th value) for the first 5 elements. The output should look similar to this:

```
userid1 6 Rick Hopper
userid2 8 Lucio Arnold
userid3 2 Brittany Parrott
…
```

Managed Memory Leak Error Message

When executing a join operation in Scala, you may see an error message such as this:

```
ERROR Executor: Managed memory leak detected
```

This message is a Spark bug and can be disregarded.

Bonus Exercises

If you have more time, attempt the following extra bonus exercises:

1. Use `keyBy` to create an RDD of account data with the postal code (9th field in the CSV file) as the key.

   Tip: Assign this new RDD to a variable for use in the next bonus exercise.

2. Create a pair RDD with postal code as the key and a list of names (Last Name,First Name) in that postal code as the value.

ii. Hint: First name and last name are the 4th and 5th fields respectively.

iii. Optional: Try using the `mapValues` operation.

3. Sort the data by postal code, then for the first five postal codes, display the code and list the names in that postal zone. For example:

```
--- 85003
Jenkins,Thad
Rick,Edward
Lindsay,Ivy

…
--- 85004
Morris,Eric
Reiser,Hazel
Gregg,Alicia
Preston,Elizabeth

…
```

This is the end of the exercise

Hands-On Exercise: Write and Run an Apache Spark Application

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-application`

Data files (HDFS): `/loudacre/weblogs`

Scala project:
`$DEVSH/exercises/spark-application/countjpgs_project`

Scala classes: `stubs.CountJPGs`
`solution.CountJPGs`

Python stub: `CountJPGs.py`

Python solution:
`$DEVSH/exercises/spark-application/python-solution/CountJPGs.py`

In this exercise, you will write your own Spark application instead of using the interactive Spark shell application.

Write a simple program that counts the number of JPG requests in a web log file. The name of the file should be passed to the program as an argument.

This is the same task as in the "Explore RDDs Using the Spark Shell" exercise. The logic is the same, but this time you will need to set up the `SparkContext` object yourself.

Depending on which programming language you are using, follow the appropriate set of instructions below to write a Spark program.

Before running your program, be sure to exit from the Spark shell.

Important: This exercise depends on a previous exercise: "Explore RDDs Using the Spark Shell." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Writing a Spark Application in Python

> **Editing Python Files**
>
> You may use any text editor you wish. If you don't have an editor preference, you may wish to use gedit, which includes language-specific support for Python.

1. If you are using Python, follow these instructions; otherwise, skip this section and continue to Writing a Spark Application in Scala below.

2. A simple stub file to get started has been provided in the exercise project: `$DEVSH/exercises/spark-application/CountJPGs.py`. This stub imports the required Spark class and sets up your main code block. Open the stub file in an editor.

3. Create a `SparkContext` object using the following code:

```
sc = SparkContext()
```

4. In the body of the program, load the file passed in to the program, count the number of JPG requests, and display the count. You may wish to refer back to the "Explore RDDs Using the Spark Shell" exercise for the code to do this.

5. At the end of the application, be sure to stop the Spark context:

```
sc.stop()
```

6. Change to the exercise working directory, then run the program, passing the name of the log file to process, for example:

```
$ cd $DEVSH/exercises/spark-application/
$ spark-submit CountJPGs.py /loudacre/weblogs/*
```

7. Once the program completes, you might need to scroll up to see your program output. (The correct number of JPG requests is 64978).

8. Skip the section below on writing a Spark application in Scala and continue with Submitting a Spark Application to the Cluster.

Writing a Spark Application in Scala

> **Editing Scala Files**
>
> You may use any text editor you wish. If you don't have an editor preference, you may wish to use gedit, which includes language-specific support for Scala. If you prefer to work in an IDE, Eclipse is included and configured for the Scala projects in the course. However, teaching use of Eclipse is beyond the scope of this course.

A Maven project to get started has been provided: `$DEVSH/exercises/spark-application/countjpgs_project`.

9. Edit the Scala class defined in `CountJPGs.scala` in `src/main/scala/stubs/`.

10. Create a `SparkContext` object using the following code:

```
val sc = new SparkContext()
```

11. In the body of the program, load the file passed to the program, count the number of JPG requests, and display the count. You may wish to refer back to the "Explore RDDs Using the Spark Shell" exercise for the code to do this.

12. At the end of the application, be sure to stop the Spark context:

```
sc.stop
```

13. Change to the project directory, then build your project using the following command:

```
$ cd \
 $DEVSH/exercises/spark-application/countjpgs_project
$ mvn package
```

14. If the build is successful, Maven will generate a JAR file called `countjpgs- 1.0.jar` in `countjpgs-project/target`. Run the program using the following command:

```
$ spark-submit \
--class stubs.CountJPGs \
target/countjpgs-1.0.jar /loudacre/weblogs/*
```

15. Once the program completes, you might need to scroll up to see your program output. (The correct number of JPG requests is 64978).

Submitting a Spark Application to the Cluster

In the previous section, you ran a Python or Scala Spark application using `spark- submit`. By default, `spark-submit` runs the application locally. In this section, run the application on the YARN cluster instead.

16. Re-run the program, specifying the cluster master in order to run it on the cluster. Use one of the commands below depending on whether your application is in Python or Scala.

    To run Python:

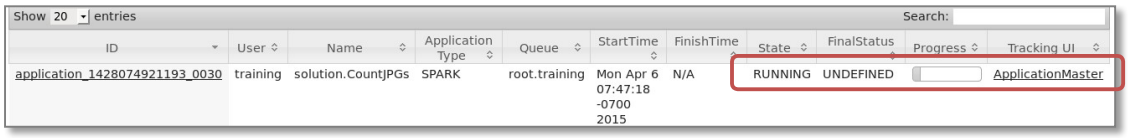
```
 $ spark-submit \
--master yarn-client \

CountJPGs.py /loudacre/weblogs/*
```

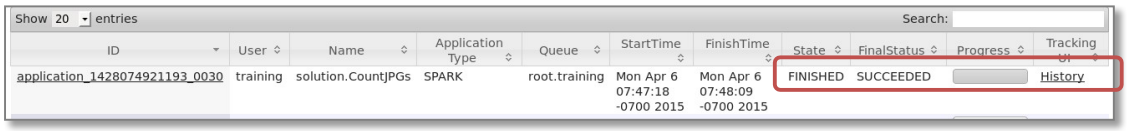
To run Scala:

```
$ spark-submit \
  --class stubs.CountJPGs \
  --master yarn-client \
  target/countjpgs-1.0.jar /loudacre/weblogs/*
```

17. After starting the application, open Firefox and visit the YARN Resource Manager UI using the provided bookmark (or going to URL `http://localhost:8088/`). While the application is running, it appears in the list of applications something like this:



After the application has completed, it will appear in the list like this:



This is the end of the exercise

Hands-On Exercise: Configure an Apache Spark Application

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-application`

Data files (HDFS): `/loudacre/weblogs`

Scala project:
`$DEVSH/exercises/spark-application/countjpgs_project`

Scala classes: `stubs.CountJPGs`
`solution.CountJPGs`

Python stub: `CountJPGs.py`

Python solution:
`$DEVSH/exercises/spark-application/python-solution/CountJPGs.py`

In this exercise, you will practice setting various Spark configuration options.

You will work with the `CountJPGs` program you wrote in the prior exercise.

Important: This exercise depends on a previous exercise: "Explore RDDs Using the Spark Shell." If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Setting Configuration Options at the Command Line

1. Change to the correct directory (if necessary) and re-run the `CountJPGs` Python or Scala program you wrote in the previous exercise, this time specifying an application name. For example:

```
$ cd $DEVSH/exercises/spark-application/
$ spark-submit --master yarn-client \
  --name 'Count JPGs' \
  CountJPGs.py /loudacre/weblogs/*
```

```
$ cd \
$DEVSH/exercises/spark-application/countjpgs_project
$ spark-submit --class stubs.CountJPGs \
  --master yarn-client \
  --name 'Count JPGs' \
  target/countjpgs-1.0.jar /loudacre/weblogs/*
```

2. Visit the Resource Manager UI again and note the application name listed is the one specified in the command line.

3. *Optional:* From the RM application list, follow the ApplicationMaster link (if the application is still running) or the History link to visit the Spark Application UI. View the **Environment** tab. Take note of the `spark.*` properties such as `master`, `appName`, and `driver` properties.

Setting Configuration Options in a Properties File

4. Using a text editor, create a file in the current working directory called `myspark.conf`, containing settings for the properties shown below:

```
spark.app.name          My Spark App
spark.master            yarn-client
spark.executor.memory   400M
```

5. Re-run your application, this time using the properties file instead of using the script options to configure Spark properties:

```
$ spark-submit --properties-file myspark.conf
  \ CountJPGs.py /loudacre/weblogs/*
```

```
$ spark-submit --properties-file myspark.conf \
  --class stubs.CountJPGs \
  target/countjpgs-1.0.jar /loudacre/weblogs/*
```

6. While the application is running, view the YARN UI and confirm that the Spark application name is correctly displayed as "My Spark App."

ID	User	Name	Application Type	Queue	StartTime
application_1433857140912_0001	training	My Spark App	SPARK	root.training	Wed Jun 10 08:35:13 -0700 2015

Setting Logging Levels

7. Copy the template file `/usr/lib/spark/conf/log4j.properties.template` to `/usr/lib/spark/conf/log4j.properties`. You will need to use superuser privileges to do this, so use the `sudo` command:

```
$ sudo cp \
/usr/lib/spark/conf/log4j.properties.template \
/usr/lib/spark/conf/log4j.properties
```

8. Load the new `log4j.properties` file into an editor. Again, you will need to use `sudo`. To edit the file with `gedit`, for instance, do this:

```
$ sudo gedit /usr/lib/spark/conf/log4j.properties
```

> **gedit Warnings**
>
> While using gedit with `sudo`, you may see `Gtk-WARNING` messages indicating permission issues or non-existent files. These can be disregarded.

9. The first line currently reads:

```
log4j.rootCategory=INFO, console
```

   Replace `INFO` with `DEBUG`:

```
log4j.rootCategory=DEBUG, console
```

10. Save the file, and then close the editor.
11. Rerun your Spark application. Notice that the output now contains both `INFO` and `DEBUG` messages, like this:

```
16/03/19 11:40:45 INFO MemoryStore: ensureFreeSpace(154293) called
with curMem=0, maxMem=311387750

16/03/19 11:40:45 INFO MemoryStore: Block broadcast_0 stored as values
to memory (estimated size 150.7 KB, free 296.8 MB)

16/03/19 11:40:45 DEBUG BlockManager: Put block broadcast_0 locally
took  79 ms

16/03/19 11:40:45 DEBUG BlockManager: Put for block broadcast_0
```

   Debug logging can be useful when debugging, testing, or optimizing your code, but in most cases it generates unnecessarily distracting output.

12. Edit the `log4j.properties` file again to replace `DEBUG` with `WARN` and try again. This time notice that no `INFO` or `DEBUG` messages are displayed, only `WARN` messages.

Note: Throughout the rest of the exercises, you may change these settings depending on whether you find the extra logging messages helpful or distracting. You can also override the current setting temporarily by calling `sc.setLogLevel` with your preferred setting. For example, in either Scala or Python, call:

```
> sc.setLogLevel("INFO")
```

This is the end of the exercise

Hands-On Exercise: View Jobs and Stages in the Spark Application UI

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-stages`

Data files (HDFS): `/loudacre/weblogs/*`

`/loudacre/accounts/*`

In this exercise, you will use the Spark Application UI to view the execution stages for a job.

In a previous exercise, you wrote a script in the Spark shell to join data from the accounts dataset with the weblogs dataset, in order to determine the total number of web hits for every account. Now you will explore the stages and tasks involved in that job.

Important: This exercise depends on previous exercises: "Explore RDDs Using the Spark Shell" and "Import Data from MySQL Using Apache Sqoop." If you did not complete those exercises, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Exploring Partitioning of File-Based RDDs

1. Start (or restart, if necessary) the Spark shell. Although you would typically run a Spark application on a cluster, your course VM cluster has only a single worker node that can support only a single executor. To simulate a more realistic multi-node cluster, run in local mode with three threads:

```
$ pyspark --master 'local[3]'
```

```
$ spark-shell --master 'local[3]'
```

2. Review the accounts dataset (`/loudacre/accounts/`) using Hue or the command line. Take note of the number of files.

3. Create an RDD based on a *single file* in the dataset, such as `/loudacre/accounts/part-m-00000`, and then call `toDebugString` on the RDD, which displays the number of partitions in parentheses `()` before the RDD file and ID. How many partitions are in the resulting RDD?

```
pyspark> accounts=sc.\
 textFile("/loudacre/accounts/part-m-00000")
pyspark> print accounts.toDebugString()
```

```
scala> var accounts=sc.
   textFile("/loudacre/accounts/part-m-00000")
scala> accounts.toDebugString
```

4. Repeat this process, but specify a minimum of three of partitions: `sc.textFile(filename,3)`. Does the RDD correctly have three partitions?

5. Finally, set the `accounts` variable to a new RDD based on *all the files* in the accounts dataset. How does the number of files in the dataset compare to the number of partitions in the RDD?

6. *Optional:* Use `foreachPartition` to print the first record of each partition.

Setting up the Job

7. Create an RDD of accounts, keyed by ID and with the string *first_name*,*last_name* for the value:

```
pyspark> accountsByID = accounts \
  .map(lambda s: s.split(',')) \
  .map(lambda values: \
     (values[0],values[4] + ',' + values[3]))
```

```
scala> val accountsByID = accounts.
  map(line => line.split(',')).
  map(values => (values(0),values(4)+','+values(3)))
```

8. Construct a `userReqs` RDD with the total number of web hits for each user ID:

   Tip: In this exercise, you will be reducing and joining large datasets, which can take a lot of time running on a single machine, as you are using in the course.
   Therefore, rather than use all the web log files in the dataset, specify a subset of web log files using a wildcard; for example, select only filenames ending in 2 by specifying `textFile("/loudacre/weblogs/*2.log")`.

```
pyspark> userReqs = sc \
  .textFile("/loudacre/weblogs/*2.log") \
  .map(lambda line: line.split()) \
  .map(lambda words: (words[2],1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```

```
scala> val userReqs = sc.
   textFile("/loudacre/weblogs/*2.log").
   map(line => line.split(' ')).
   map(words => (words(2),1)).
   reduceByKey((v1,v2) => v1 + v2)
```

9. Then join the two RDDs by user ID, and construct a new RDD with first name, last name, and total hits:

```
pyspark> accountHits = accountsByID.join(userReqs)\
  .values()
```

```
scala> val accountHits =
  accountsByID.join(userReqs).map(pair => pair._2)
```

10. Print the results of `accountHits.toDebugString` and review the output. Based on this, see if you can determine

a. How many stages are in this job?

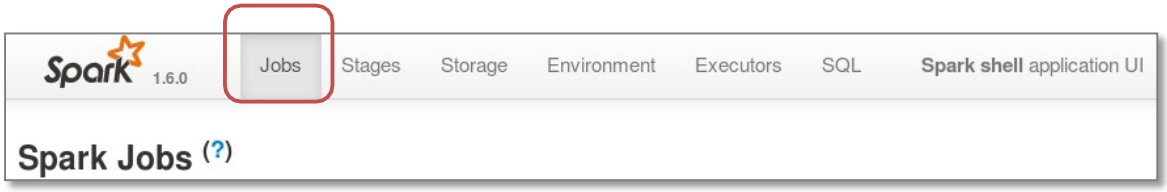
b. Which stages are dependent on which?

c. How many tasks will each stage consist of?

Running the Job and Reviewing the Job in the Spark Application UI

11. In your browser, visit the Spark Application UI by using the provided toolbar bookmark, or visiting URL `http://localhost:4040/`.

12. In the Spark UI, make sure the **Jobs** tab is selected. No jobs are yet running so the list will be empty.



13. Return to the shell and start the job by executing an action (`saveAsTextFile`):

```
pyspark> accountHits.\

  saveAsTextFile("/loudacre/userreqs"
```

```
scala> accountHits.
  saveAsTextFile("/loudacre/userreqs")
```

14. Reload the Spark UI Jobs page in your browser. Your job will appear in the Active Jobs list until it completes, and then it will display in the Completed Jobs List.

15. Click the job description (which is the last action in the job) to see the stages. As the job progresses you may want to refresh the page a few times.

    Things to note:

    a. How many stages are in the job? Does it match the number you expected from the RDD's `toDebugString` output?

b. The stages are numbered, but the numbers do not relate to the order of execution. Note the times the stages were submitted to determine the order. Does the order match what you expected based on RDD dependency?

c. How many tasks are in each stage?

d. The Shuffle Read and Shuffle Write columns indicate how much data was copied between tasks. This is useful to know because copying too much data across the network can cause performance issues.

16. Click the stages to view details about that stage. Things to note:

a. The Summary Metrics area shows you how much time was spend on various steps. This can help you narrow down performance problems.

b. The Tasks area lists each task. The Locality Level column indicates whether the process ran on the same node where the partition was physically stored or not. Remember that Spark will attempt to always run tasks where the data is, but may not always be able to, if the node is busy.

c. In a real-world cluster, the executor column in the Task area would display the different worker nodes that ran the tasks. (In this single- node cluster, all tasks run on the same host: `localhost`.)

17. When the job is complete, return to the **Jobs** tab to see the final statistics for the number of tasks executed and the time the job took.

18. *Optional:* Try re-running the last action. (You will need to either delete the `saveAsTextFile` output directory in HDFS, or specify a different directory name.) You will probably find that the job completes much faster, and that several stages (and the tasks in them) show as "skipped."

Bonus question: Which tasks were skipped and why?

This is the end of the exercise

Hands-On Exercise: Persist an RDD

Files and Data Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-persist`

Data files (HDFS): `/loudacre/weblogs/*`
`/loudacre/accounts/*`

Stubs: `SparkPersist.pyspark`
`SparkPersist.scalaspark`

In this exercise, you will practice how to persist RDDs.

Important: This exercise depends on previous exercises: "Explore RDDs Using the Spark Shell" and "Import Data from MySQL Using Apache Sqoop." If you did not complete those exercises, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

1. Copy the code in the `SparkPersist` stub script in the exercise directory (`$DEVSH/exercises/spark-persist`) into the Spark shell. (A stub is provided for Scala and Python; use whichever language you prefer.)

2. The stub code is very similar to the job setup code in the "View Jobs and Stages in the Spark Application UI" exercise. It sets up an RDD called `accountHits` that joins account data with web log data. However, this time you will start the job by performing a slightly different action than in that exercise: count the number of user accounts with a total hit count greater than five. Enter the code below into the shell:

```
pyspark> accountHits\
  .filter(lambda (firstlast,hitcount): hitcount > 5)\
  .count()
```

```
scala> accountHits.filter(pair => pair._2 > 5).count()
```

3. Persist the RDD to memory by calling `accountHits.persist()`.

4. In your browser, view the Spark Application UI and select the **Storage** tab. At this point, you have marked your RDD to be persisted, but you have not yet performed an action that would cause it to be materialized and persisted, so you will not yet see any persisted RDDs.

5. In the Spark shell, execute the count again.

6. View the RDD's `toDebugString`. Notice that the output indicates the persistence level selected.

7. Reload the **Storage** tab in your browser, and this time note that the RDD you persisted is shown. Click the RDD Name to see details about partitions and persistence.

8. Click the **Executors** tab and take note of the amount of memory used and available for your one worker node.

   Note that the classroom environment has a single worker node with a small amount of memory allocated, so you may see that not all of the dataset is actually cached in memory. In the real world, for good performance a cluster will have more nodes, each with more memory, so that more of your active data can be cached.

9. *Optional:* Set the RDD's persistence level to `DISK_ONLY` and compare the storage report in the Spark Application Web UI.

   - Hint: Set the RDD's persistence level to `StorageLevel.DISK_ONLY`. You will need to import the class first or use the fully qualified name of the class `StorageLevel` when invoking `persist()`.
   - Hint: Because you have already persisted the RDD at a different level, you will need to `unpersist()` first before you can set a new level.

This is the end of the exercise

Hands-On Exercise: Use Apache Spark SQL for ETL

Files and Data Used in This Exercise

Exercise directory: `$DEVSH/exercises/spark-sql`

MySQL database: `loudacre`

MySQL table: `webpage`

Output path (HDFS): `/loudacre/webpage_files`

In this exercise, you will use Spark SQL to load structured data from a Parquet file, process it, and store it to a new file.

The data you will work with in this exercise is from the `webpage` table in the `loudacre` database in MySQL. Although Spark SQL does allow you to directly access tables in a database using JDBC, doing so is not generally a best practice, because in a distributed environment it may lead to an unintentional Denial of Service attack on the database. So in this exercise, you will use Sqoop to import the data to HDFS first. You will use Parquet file format rather than a text file because this preserves the data's schema for use by Spark SQL.

Importing Data from MySQL Using Sqoop

1. In a terminal window, use Sqoop to import the `webpage` table from MySQL. Use Parquet file format.

```
$ sqoop import \
  --connect jdbc:mysql://localhost/loudacre \
  --username training --password training \
  --table webpage \
  --target-dir /loudacre/webpage \
  --as-parquetfile
```

2. Using Hue or the `hdfs` command line utility, list the data files imported to the `/loudacre/webpage` directory.

3. *Optional:* Download one of the generated Parquet files from HDFS to a local directory. Use `parquet-tools head` and `parquet-tools schema` to review the schema and some sample data. Take note of the structure of the data; you will use this data in the next exercise sections.

Creating a DataFrame from a Table

4. If necessary, start the Spark shell.

5. The Spark shell predefines a SQL context object as `sqlContext`. What type is the SQL context? In either Python or Scala, view the `sqlContext` object:

```
> sqlContext
```

6. Create a DataFrame based on the `webpage` table:

```
pyspark> webpageDF = sqlContext \
  .read.load("/loudacre/webpage")
```

```
scala> val webpageDF = sqlContext.
  read.load("/loudacre/webpage")
```

7. Examine the schema of the new DataFrame by calling `webpageDF.printSchema()`.

8. View the first few records in the table by calling `webpageDF.show(5)`.

   Note that the data in the `associated_files` column is a comma-delimited string. Loudacre would like to make this data available in an Impala table, but in order to perform required analysis, the `associated_files` data must be extracted and normalized. Your goal in the next section is to use the

DataFrames API to extract the data in the column, split the string, and create a new data file in HDFS containing each page ID, and its associated files in separate rows.

Querying a DataFrame

9. Create a new DataFrame by selecting the `web_page_num` and `associated_files` columns from the existing DataFrame:

```
python> assocFilesDF = \
webpageDF.select(webpageDF.web_page_num,\
webpageDF.associated_files)
```

```
scala> val assocFilesDF =
  webpageDF.select($"web_page_num",$"associated_files")
```

10. View the schema and the first few rows of the returned DataFrame to confirm that it was created correctly.

11. In order to manipulate the data using core Spark, convert the DataFrame into a Pair RDD using the `map` method. The input into the `map` method is a `Row` object. The key is the `web_page_num` value, and the value is the `associated_files` string.

In Python, you can dynamically reference the column value of the `Row` by name:

```
pyspark> aFilesRDD = assocFilesDF.map(lambda row: \
(row.web_page_num,row.associated_files))
```

In Scala, use the correct `get` method for the type of value with the column index:

```
scala> val aFilesRDD = assocFilesDF.
  map(row => (row.getAs[Short]("web_page_num"),
    row.getAs[String]("associated_files")))
```

12. Now that you have an RDD, you can use the familiar `flatMapValues` transformation to split and extract the filenames in the `associated_files` column:

```
pyspark> aFilesRDD2 = aFilesRDD \
  .flatMapValues( \
    lambda filestring:filestring.split(','))
```

```
scala> val aFilesRDD2 =
  aFilesRDD.flatMapValues(filestring =>
    filestring.split(','))
```

13. Import the `Row` class and convert the pair RDD to a `Row` RDD. (Note: this step is only necessary in Scala.)

```
scala> import org.apache.spark.sql.Row
scala> val aFilesRowRDD = aFilesRDD2.map(pair =>
Row(pair._1,pair._2))
```

14. Convert the RDD back to a DataFrame, using the original DataFrame's schema:

```
pyspark> aFileDF = sqlContext. \
createDataFrame(aFilesRDD2,assocFilesDF.schema)
```

```
scala> val aFileDF = sqlContext.
  createDataFrame(aFilesRowRDD,assocFilesDF.schema)
```

15. Call `printSchema` on the new DataFrame. Note that Spark SQL gave the columns the same names they had originally: `web_page_num` and `associated_files`. The second column name is no longer accurate, because the data in the column reflects only a *single* associated file.

16. Create a new DataFrame with the `associated_files` column renamed to `associated_file`:

```
pyspark> finalDF = aFileDF. \
  withColumnRenamed('associated_files', \
    'associated_file')
```

```
scala> val finalDF = aFileDF.
  withColumnRenamed("associated_files",
    "associated_file")
```

17. Call `finalDF.printSchema()` to confirm that the new DataFrame has the correct column names.

18. Call `show(5)` on the new DataFrame to confirm that the final data is correct.

19. Your final DataFrame contains the processed data, so save it in Parquet format (the default) in directory `/loudacre/webpage_files`.

```
pyspark> finalDF.write. \
  mode("overwrite"). \
  save("/loudacre/webpage_files")
```

```
scala> finalDF.write.
  mode("overwrite").
  save("/loudacre/webpage_files")
```

20. Using Hue or the HDFS command line tool, list the Parquet files that were saved by Spark SQL.

21. *Optional:* Use `parquet-tools schema` and `parquet-tools head` to review the schema and some sample data of the generated files.

22. *Optional:* In the Spark Web UI, try viewing the **SQL** tab. How many queries were completed as part of this exercise? How many jobs?

This is the end of the exercise

Hands-On Exercise: Write an Apache Spark Streaming Application

Files and Directories Used in This Exercise:

Exercise directory: `$DEVSH/exercises/spark-streaming`

Python stub: `stubs-python/StreamingLogs.py`

Python solution: `solution-python/StreamingLogs.py`

Scala project:

- **Project directory:** `streaminglogs_project`
- **Stub class:** `stubs.StreamingLogs`
- **Solution class:** `solution.StreamingLogs`

Test data (local): `$DEVDATA/weblogs/*`

Test script: `streamtest.py`

In this exercise, you will write a Spark Streaming application to count Knowledge Base article requests.

This exercise has two parts. First, you will review the Spark Streaming documentation. Then you will write and test a Spark Streaming application to read streaming web server log data and count the number of requests for Knowledge Base articles.

Reviewing the Spark Streaming Documentation

1. View the Spark Streaming API by opening the Spark API documentation for either Scala or Python and then:

   For Scala:

   - Scroll down and select the `org.apache.spark.streaming` package in the package pane on the left.

- Follow the links at the top of the package page to view the `DStream` and `PairDStreamFunctions` classes— these will show you the methods available on a DStream of regular RDDs and Pair RDDs respectively.

For Python:

- Go to the `pyspark.streaming` module.
- Scroll down to the `pyspark.streaming.DStream` class and review the available methods.

2. You may also wish to view the *Spark Streaming Programming Guide* (select **Programming Guides > Spark Streaming** on the Spark documentation main page).

Simulating Streaming Web Logs

To simulate a streaming data source, you will use the provided `streamtest.py` Python script, which waits for a connection on the host and port specified and, once it receives a connection, sends the contents of the file(s) specified to the client (which will be your Spark Streaming application). You can specify the speed (in lines per second) at which the data should be sent.

3. Change to the exercise directory.

```
$ cd $DEVSH/exercises/spark-streaming
```

4. Stream the Loudacre web log files at a rate of 20 lines per second using the provided test script.

```
$ python streamtest.py localhost 1234 20 \
  $DEVDATA/weblogs/*
```

This script will exit after the client disconnects, so you will need to restart the script whenever you restart your Spark application.

Writing a Spark Streaming Application

5. To help you get started writing a Spark Streaming application, stub files have been provided for you.

   For Python, start with the stub file `StreamingLogs.py` in the `$DEVSH/exercises/spark-streaming/stubs-python` directory, which imports the necessary classes for the application.

   For Scala, a Maven project directory calling `streaminglogs_project` has been provided in the exercise directory (`$DEVSH/exercises/spark- streaming`). To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogs.scala`, which imports the necessary classes for the application.

6. Define a Streaming context with a one-second batch duration.

7. Create a DStream by reading the data from the host and port provided as input parameters.

8. Filter the DStream to only include lines containing the string `KBDOC`.

9. To confirm that your application is correctly receiving the streaming web log data, display the first five records in the filtered DStream for each one-second batch. (In Scala, use the DStream `print` function; in Python, use `pprint`.)

10. For each RDD in the filtered DStream, display the number of items—that is, the number of requests for KB articles.

    Tip: Python does not allow calling `print` within a lambda function, so create a named defined function to print.

11. Save the filtered logs to text files in HDFS. Use the base directory name `/loudacre/streamlog/kblogs`.

12. Finally, start the Streaming context, and then call `awaitTermination()`.

Testing the Application

13. In a new terminal window, change to the correct directory for the language you are using for your application.

For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/spark-streaming
```

For Scala, change to the project directory for the exercise:

```
$ cd \
 $DEVSH/exercises/spark-streaming/streaminglogs_project
```

14. If you are using Scala, build your application JAR file using the `mvn package` command.

15. Use `spark-submit` to run your application locally and be sure to specify two threads; at least two threads or nodes are required to run a streaming application, while the VM cluster has only one. The `StreamingLogs` application takes two parameters: the host name and the port number to connect the DStream to. Specify the same host and port at which the test script you started earlier is listening.

```
$ spark-submit --master 'local[2]' \
  stubs-python/StreamingLogs.py localhost 1234
```

Note: Use **`solution-python`**`/StreamingLogs.py` to run the solution application instead.

```
$ spark-submit --master 'local[2]' \
  --class stubs.StreamingLogs \
  target/streamlog-1.0.jar localhost 1234
```

Note: Use `--class` **`solution`**`.StreamingLogs` to run the solution class instead.

16. After a few moments, the application should connect to the test script's simulated stream of web server log output. Confirm that for every batch of data received (every second), the application displays the first few Knowledge Base requests and the count of requests in the batch. Review the HDFS files the application saved in `/loudacre/streamlog`.

17. Return to the terminal window in which you started the `streamtest.py` test script earlier. Stop the test script by typing `Ctrl+C`. You do not need to wait until all the web log data has been sent.

> **Warning: Stopping Your Application**
>
> You *must* stop the test script before stopping your Spark Streaming application. If you attempt to stop the application while the test script is still running, you may find that the application appears to hang while it takes several minutes to complete. (It will make repeated attempts to reconnect with the data source, which the test script does not support.)

18. After the test script has stopped, stop your application by typing `Ctrl+C` in the taerminal window the application is running in.

This is the end of the exercise