

DEV 352 – MapR Streams and Spark Streaming  
Lab Guide

Spring 2016 – Version 5.1.0

# For use with the following courses:

DEV 352 – Developing MapR Streams Applications

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2016, MapR Technologies, Inc.  All rights reserved. All other trademarks cited here are the property of their respective owners.

Using This Guide

# Icons Used in the Guide

This lab guide uses the following ions to draw attention to different types of information:

|  |  |
| --- | --- |
|  | **Note**: Additional information that will clarify something, provide details, or help you avoid mistakes. |
|  | **CAUTION**: Details you **must** read to avoid potentially serious problems. |
|  | **Q&A**: A question posed to the learner during a lab exercise. |
|  | **Try This!** Exercises you can complete after class (or during class if you finish a lab early) to strengthen learning. |

# Lab Files

Here is a description of the materials (data files, licenses etc.) that are supplied to support the labs. These materials should be downloaded to your system before beginning the labs.

|  |  |
| --- | --- |
| DEV352\_LabFiles.zip | Scripts, data, and other files necessary to complete this lab. |
| DEV352\_LabGuide.pdf | Instructions for completing this lab (this document). |
| MapR\_Lab\_Environment\_ Connection\_Guide.pdf | Instructions for connecting to your MapR lab environment. |
| Software\_Setup\_Guide.pdf | Instructions for installing your Java lab environment. |

Lab 5: Developing MapR Streams Applications Using Kafka Java API

# Lab Overview

The goal of this lab is to develop and run a MapR Streams producer and consumer sending and receiving messages. Later you will add a Spark streaming consumer to the same topic. You will be using either the MapR Sandbox or a MapR cluster to run the lab exercises.

|  |  |  |
| --- | --- | --- |
| **Exercise** | **Required** | **Duration** |
| **5.1 Build and run MapR Streams Producer** | Yes | 15 min |
| **5.2 Finish code, build and run Spark Streaming Consumer** | Yes | 15 min |

# Exercise 5.0: Setup

## Set up for the Lab

**Copy files to the Sandbox or Cluster**

Download the files DEV352Data.zip and DEV352LabFiles.zip to your machine.

1. Copy DEV352Data.zip file to your sandbox or cluster user directory, as instructed in the Connect to MapR Sandbox or Cluster document.

scp DEV352Data.zip <username>@node-ip:/user/<username>/.

For example, if you are using the VirtualBox Sandbox, the command may look something like this, if you are in the ms-lab target folder where the jar file gets built:

$ scp -P 2222 DEV352Data.zip user01@127.0.0.1:/user/user01/.

1. Unzip data.zip

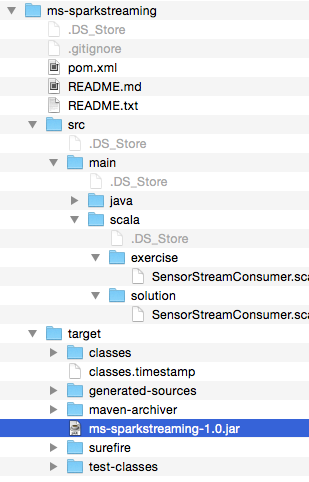
unzip DEV352Data.zip

ls /user/<username>/data

You should see the data files (sensordata.csv … ).

1. Now unzip DEV352LabFiles.zip on your laptop.

When you unzip the file it will create the following structure.



In these labs we will be using Java, Scala and maven. You may need to add a scala plugin to your IDE, if you have not already done the DEV 362 Lab 8.

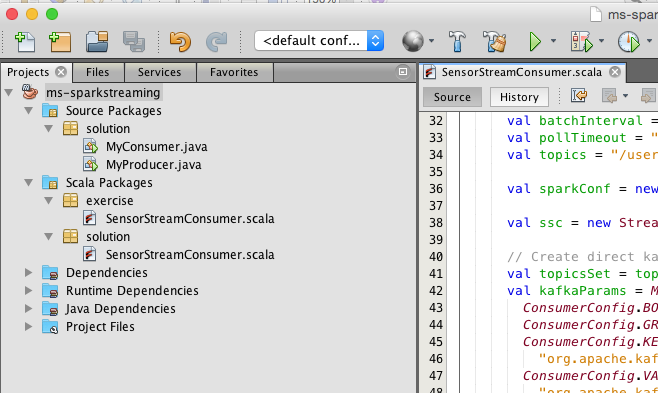
**Downloading an IDE**

You can use your choice of Netbeans, Intellij, Eclipse, or just a text editor with maven on the command line. You need to install your IDE of choice on your laptop, or alternatively install maven on the sandbox and use the command line. If you use Netbeans or Intellij you only need to add the scala plugin, maven is included. If you use Eclipse, depending on the version, you may need to add the maven and scala plugins.

* Netbeans:
  + <https://netbeans.org/downloads/>
  + click on tools plugins and add the scala plugin
* Eclipse with Scala and maven:
  + <http://scala-ide.org/download/sdk.html>
* Intellij
  + <https://www.jetbrains.com/idea/download/>
* maven (install on the sandbox, if you just want to edit the files on the sandbox)
  + https://maven.apache.org/download.cgi

**Open/Import the Project into your IDE**

There is an exercises package with stubbed code for you to finish and there is a solutions package with the complete solution. Open/Import the project into your IDE following the instructions below. Optionally you can just edit the scala files and use maven on the command line, or if you just want to use the prebuilt solution, you can copy the solution jar file from the target directory.



Here are some links for opening or importing a maven project with your IDE:

* Netbeans
  + <http://wiki.netbeans.org/MavenBestPractices#Open_existing_project>
* Intellij
  + <http://www.tutorialspoint.com/maven/maven_intellij_idea.htm>
* Eclipse
  + <http://scala-ide.org/docs/current-user-doc/gettingstarted/index.html>

# Exercise 5.1: Build the project

### Build a Maven project to create the jar file

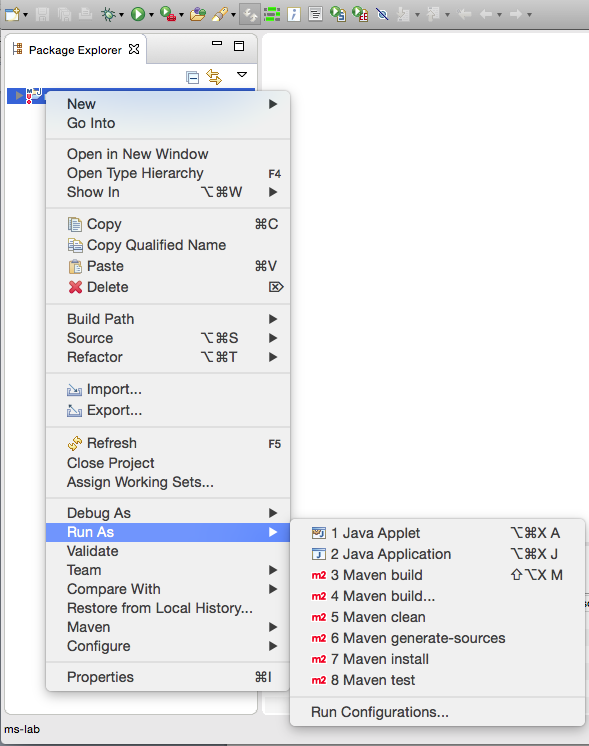
**Building the Project**

Here is how you build a maven project with your IDE:

* Netbeans
  + Right mouse click on the project and select build
* Intellij
  + Select **Buid menu > Rebuild Project** Option
* Eclipse
  + Right mouse click on the project and select run as maven install.

Building will create the **ms-sparkstreaming-1.0.jar** in the target folder. You copy this jar file to your sandbox or cluster node to run your application.

1. With Eclipse Select the project from Run menu.
2. Select **Run As** and **Maven install** as shown in the following image.



A jar file will be created in the project targetfolder.

# Exercise 5.2: Run the MapR Streams Producer

## Run the code on a sandbox or cluster

**Note:** Substitute your user ID for user01.

1. Copy the jar file *ms-sparkstreaming-1.0.jar* from the projecttarget folder to your sandbox or cluster node, as instructed in the MapR Lab Environment Connection Guide document.

For example, if you are using the VirtualBox Sandbox, the command may look something like this, if you are in the ms-lab target folder where the jar file gets built:

$ scp -P 2222 **ms-lab-1.0.jar** user01@127.0.0.1:/user/user01/.

1. Log in to the sandbox or cluster, as instructed in the MapR Lab Environment Connection Guide document. For example:

$ ssh user01@<*IP address*> -p <*port*>

1. Create the stream:

**maprcli stream create -path /user/user01/pump -produceperm p -consumeperm p -topicperm p**

**maprcli stream topic create -path /user/user01/pump -topic sensor -partitions 3**

1. Get info on the stream:

**maprcli stream info -path /user/user01/pump**

1. Run the MapR Streams producer:

**java -cp ms-sparkstreaming-1.0.jar:`mapr classpath` solution.MyProducer**

1. Run the consumer:

**java -cp ms-sparkstreaming-1.0.jar:`mapr classpath` solution.MyConsumer**

1. Get info on the stream topic:

**maprcli stream topic info -path /user/user01/pump -topic sensor -json**

**Troubleshooting:**

If you have problems running the producer or consumer, make sure that you have the correct name for the jar file, the Java class, and the package name.

**java -cp ms-sparkstreaming-1.0.jar:`mapr classpath` solution.MyProducer**

`mapr classpath` is a utility which sets the rest of the jars needed in the classpath.

# Exercise 5.3: Finish and Run Spark Streaming MapR Streams consumer (works also with Kafka)

**Solution Code Overview**

## Initializing the StreamingContext

First we create a [StreamingContext](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.StreamingContext), the main entry point for streaming functionality, with a 2 second [batch interval](http://spark.apache.org/docs/latest/streaming-programming-guide.html#setting-the-right-batch-interval).

|  |
| --- |
| val sparkConf = new SparkConf().setAppName("Stream")  // create a StreamingContext, the main entry point for all streaming functionality  val ssc = new StreamingContext(sparkConf, Seconds(2)) |

Next we configure Spark Streaming to receive data from MapR Streams, using the Kafka API.

We import KafkaUtils and create a direct input DStream as shown below. For more information refer to this documentation <http://spark.apache.org/docs/latest/streaming-kafka-integration.html>

The newer Spark Kafka direct approach is receiver-less, which is efficient, simplifies parallelism, and provides exactly once semantics. Instead of using receivers to receive data, this approach periodically polls for the latest message offsets in each topic+partition.

In the Kafka parameters, we specify:

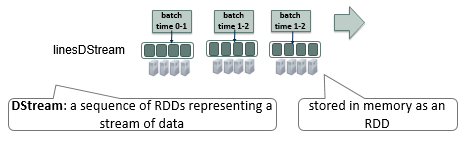
* bootstrap.servers: MapR Streams does not actually use this, but it Spark requires this to be set.
* auto.offset.reset:
  + **earliest** Reset the offset to the offset of the earliest message in the partition.
  + **latest** (default) Reset the offset to the offset of the latest message in the partition.

For more information refer to the Spark MapR Streams documentation <http://maprdocs.mapr.com/51/index.html#Spark/Spark_IntegrateMapRStreams_Consume.html>

And the MapR Streams configuration parameters for consumers. [**http://maprdocs.mapr.com/51/#MapR\_Streams/configuration\_parameters\_for\_consumers.html**](http://maprdocs.mapr.com/51/#MapR_Streams/configuration_parameters_for_consumers.html)

|  |
| --- |
| import org.apache.spark.streaming.kafka.v09.KafkaUtils  val topics = "/user/user01/pump:sensor"  val topicsSet = topics.split(",").toSet  val kafkaParams = Map[String, String](  ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG -> brokers,  ConsumerConfig.GROUP\_ID\_CONFIG -> groupId,  ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG ->  "org.apache.kafka.common.serialization.StringDeserializer",  ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG ->  "org.apache.kafka.common.serialization.StringDeserializer",  ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG -> “earliest”,  ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG -> "false",  "spark.kafka.poll.time" -> pollTimeout  )  // create a DStream that represents streaming data from a Kafka or MapR Streams source  val dStream = KafkaUtils.createDirectStream[String, String](ssc, kafkaParams, topicsSet) |

The dStream represents the stream of data. Internally a DStream is a sequence of RDDs, one RDD per batch interval.

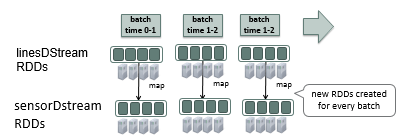


**Apply Transformations and output operations to DStreams**

Next we parse the message values (the second parameter) into Sensor objects, with the map operation on the sensorDstream.

|  |
| --- |
| // parse message value into sensor objects (not using key)  val sensorDStream = dStream.map(\_.\_2).map(parseSensor) |

The map operation applies the parseSensor function on the RDDs in the dStream, resulting in RDDs of Sensor objects.



Next we use the DStream [foreachRDD](https://spark.apache.org/docs/1.0.0/api/java/org/apache/spark/streaming/dstream/DStream.html) method to apply processing to each RDD in this DStream. We perform some sql on the data

|  |
| --- |
| // for each RDD. performs function on each RDD in DStream  sensorRDD.foreachRDD { rdd =>  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)  import sqlContext.implicits.\_  import org.apache.spark.sql.functions.\_  val sensorDF = rdd.toDF()  sensorDF.show()  sensorDF.registerTempTable("sensor")  val res = sqlContext.sql("SELECT resid, date, count(resid) as total FROM sensor GROUP BY resid, date")  println("sensor count ")  res.show  val res2 = sqlContext.sql("SELECT resid, date, avg(psi) as avgpsi FROM sensor GROUP BY resid,date")  println("sensor psi average")  res2.show  } |

**Start receiving data**

To start receiving data, we must explicitly call start() on the StreamingContext, then call awaitTermination to wait for the streaming computation to finish.

|  |
| --- |
| // Start the computation  ssc.start()  // Wait for the computation to terminate  ssc.awaitTermination() |

**Finish Code, Build and Run**

First, look at the solution code in the solutions folder. Then, run this code and observe the results.

1. Look at the code in the src/main/scala/solution directory. Open SensorStreamConsumer.scala .
2. Look in the code src/main/scala/exercise directory. Finish the code following the // TODOs
3. Build your code as instructed before to create an updated **ms-sparkstreaming-1.0.jar** in the target folder.
4. Copy the jar file *ms-sparkstreaming-1.0.jar* from the projecttarget folder to your sandbox or cluster node.

For example, if you are using the VirtualBox Sandbox, the command may look something like this, if you are in the ms-lab target folder where the jar file gets built:

$ scp -P 2222 **ms-lab-1.0.jar** user01@127.0.0.1:/user/user01/.

1. Log in to the sandbox or cluster, as instructed in the MapR Lab Environment Connection Guide document. For example:

$ ssh user01@<*IP address*> -p <*port*>

1. In your sandbox terminal window run the streaming app, Spark submit **Format:** spark-submit --driver-class-path `hbase classpath` \ --class **package.class** applicationjarname.jar. If you are running the solution replace the exercises package name with solutions. For more information on Spark Submit http://spark.apache.org/docs/latest/submitting-applications.html

**spark-submit --class exercises.SensorStreamConsumer --master local[2] ms-sparkstreaming-1.0.jar**

to submit the solution:

**spark-submit --class solutions.SensorStreamConsumer --master local[2] ms-sparkstreaming-1.0.jar**

1. If you want to send more messages, ssh into another terminal window and run the producer again:

**java -cp ms-sparkstreaming-1.0.jar:`mapr classpath` solution.MyProducer**

## Observe Streaming Application in Web UI

Launch the Spark Streaming UI in the browser with you sandbox or cluster ipaddress and port 4040: http:://ipaddress:4040. Click on the Streaming tab. Note if you are running on the cluster port 4040 may already be taken by another user, you can see which port you got when the shell starts.

When you are finished use **ctl c** to stop the streaming application.

step 4: Spark streaming app writing to hbase

first make sure that you have the correct version of HBase installed, it should be 1.1.1:

cat /opt/mapr/hbase/hbaseversion

1.1.1

Next make sure the Spark HBase compatibility version is correctly configured here:

cat /opt/mapr/spark/spark-1.5.2/mapr-util/compatibility.version

hbase\_versions=1.1.1

If this is not 1.1.1 fix it.

Create an hbase table to write to:

launch the hbase shell

$hbase shell

create '/user/user01/sensor', {NAME=>'data'}, {NAME=>'alert'}, {NAME=>'stats'}

Step 4: start the streaming app writing to HBase

spark-submit --class solution.HBaseSensorStream --master local[2] ms-sparkstreaming-1.0.jar

ctl-c to stop

Scan HBase to see results:

$hbase shell

scan '/user/user01/sensor' , {'LIMIT' => 5}

scan '/user/user01/sensor' , {'COLUMNS'=>'alert', 'LIMIT' => 50}

Step 5:

exit, run spark (not streaming) app to read from hbase and write daily stats

spark-submit --class solution.HBaseReadRowWriteStats --master local[2] ms-sparkstreaming-1.0.jar

$hbase shell

scan '/user/user01/sensor' , {'COLUMNS'=>'stats', 'LIMIT' => 50}

cleanup if you want to re-run:

maprcli stream topic delete -path /user/user01/pump -topic sensor

maprcli stream topic create -path /user/user01/pump -topic sensor -partitions 3

hbase shell

truncate '/user/user01/sensor'