HDP Developer: Storm

Lab Guide

Title HDP Developer: Storm

Version: Rev 1

Date: June 1, 2015

Table of Contents

Lab: Configuring a Storm Development Environment

Setting up a Storm development environment using Eclipse and Gradle

Table 1. *About this Lab*

Objective:	Setup an environment for developing Storm applications for Hadoop written in Java using Eclipse and Gradle.
File locations:	`/root/storm/labs/WordCount`
Successful outcome:	You will have a new Eclipse project defined and configured for developing Java Storm applications.
Before you begin	This lab assumes you have already completed the steps in the course setup guide and imported the classroom VM into `VMWare Player/Fusion`.
Related lesson:	

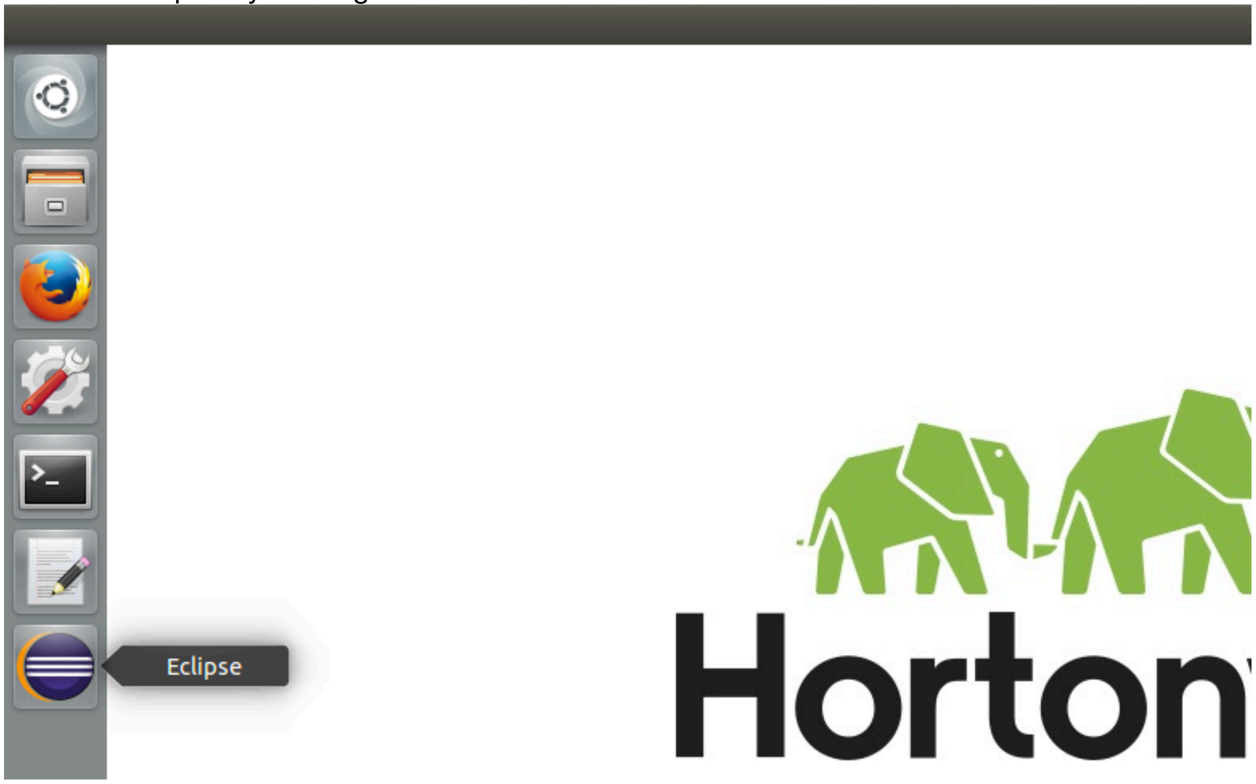
Perform the following steps:

Step 1: Start the VM

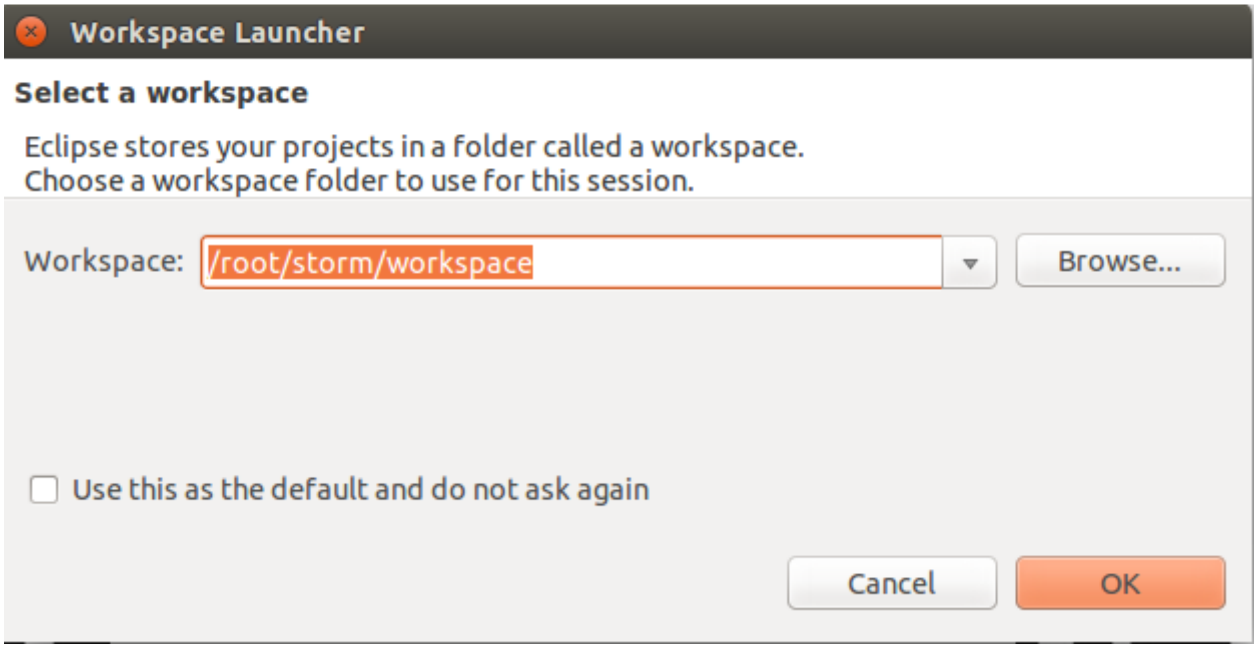
1.1. Using VMWare Player or VirtualBox, or using an AWS instance URL provided by your instructor, start up the classroom virtual machine and login. Login as `root`, and the password is `hadoop`.

Step 2: Start Eclipse

2.1. Start Eclipse by clicking on the shortcut on the left-hand toolbar:



2.2. Make sure the workspace folder is `/root/storm/workspace`:



Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /root/storm/workspace

☐ Use this as the default and do not ask again

Step 3: Create a New Eclipse Project

3.1. Open a Terminal window in your VM by clicking on the `Terminal` shortcut on the left-hand toolbar, or by using the Ctrl+Alt+T keyboard shortcut.

3.2. From the Terminal window, change directories to `/root/storm/workspace`:

```
# cd ~/storm/workspace/
```

3.3. You are going to create a project named WordCount. Start by making a new subdirectory of workspace named WordCount:

```
# mkdir WordCount
```

3.4. Copy the provided `build.gradle` file in the `labs/WordCount` folder into the `workspace/WordCount` folder:

```
# cd WordCount
# cp ~/storm/labs/WordCount/build.gradle .
```

3.5. View the contents of `build.gradle` using the more command:

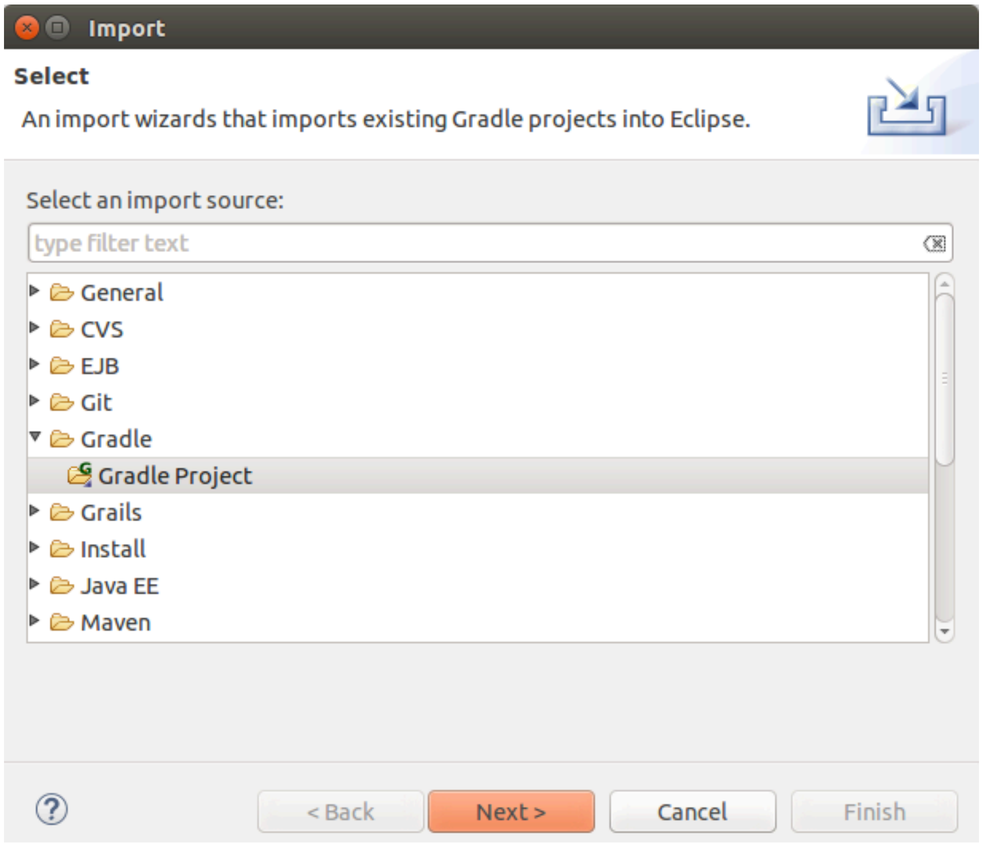
```
# more build.gradle
project.ext.mainclass = 'wordcount.WordCountJob'
project.ext.archiveName = 'wordcount.jar'

apply from: '/root/storm/labs/build.gradle'
```

Notice it defines the name of the JAR file and the `main` class within that JAR file. The other settings are inherited from the `build.gradle` file in the `/root/storm/labs` folder.

Step 4: Import the Project into Eclipse

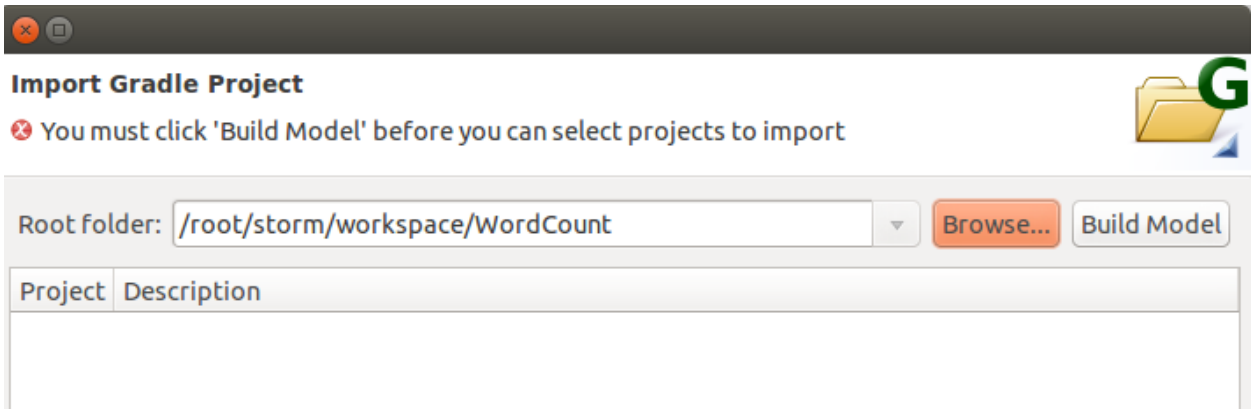
4.1. From the `Eclipse` menu, select `File -> Import…`.

4.2. Expand the `Gradle` folder and select `Gradle Project`:



4.3. Click the `Next` button.

4.4. Click the `Browse…` button next to the "`Root folder:`" textbox and select your `/root/storm/workspace/WordCount` folder:



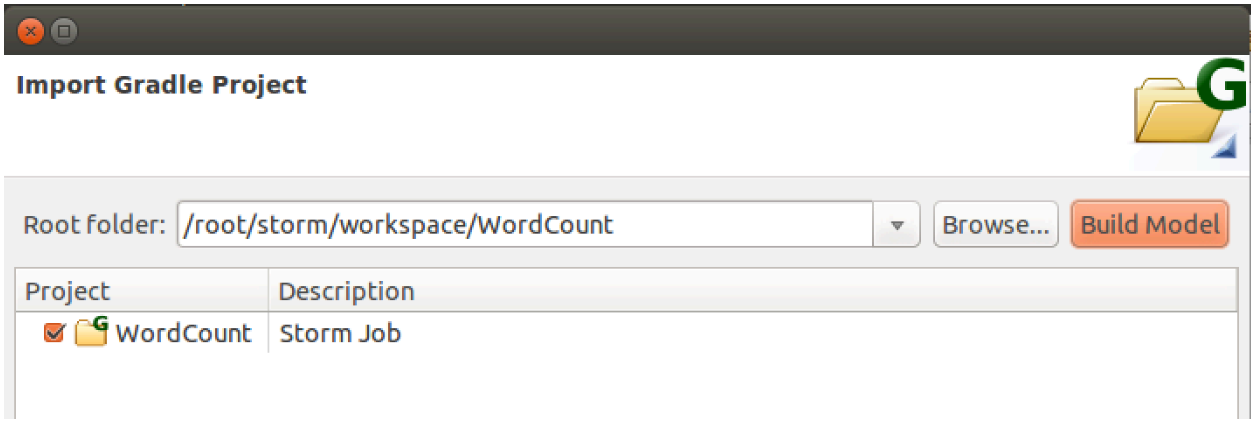
Import Gradle Project

You must click 'Build Model' before you can select projects to import

Root folder: /root/storm/workspace/WordCount

4.5. Click the `Build Model` button next to the `Browse...` button. This will cause `WordCount` to appear in the list of available projects.

4.6. Check the box next to WordCount:



Import Gradle Project

Root folder: /root/storm/workspace/WordCount

Project	Description
✓ WordCount	Storm Job

4.7. Click the `Finish` button. Wait for the project to be imported into Eclipse.

4.8. You should now see `WordCount` as a Gradle project in Eclipse in the Project Explorer window. Your Eclipse is project is ready to go. It will be used in a future lab to develop a Storm application.

Step 5: Start the HDP Cluster

5.1. From the Terminal window, run the following command (which should be in your `PATH`, so you can run the command from any directory):

```
# storm_cluster.sh
```

5.2. Wait for the script to complete. This script is starting up 5 nodes (using Docker) that create an HDP cluster. Three of the nodes are master nodes named `namenode`, `resourcemanager`, and `hiveserver`. The other 2 nodes are worker nodes named `node1` and `node2` and have the DataNode and NodeManager processes running on them.

5.3. Run the following command:

```
# hdfs dfsadmin -report
```

Scroll up through the output of the report and verify the number of available DataNodes is 2.

5.4. Enter the following command:

```
# yarn node -list
```

Verify you have 2 NodeManagers in your cluster.

Lab: Storm WordCount

Develop a simple Storm application

Table 2. About this Lab

Objective:	Write a WordCount application in Storm.
File locations:	`/root/storm/workspace/WordCount`
Successful outcome:	
Before you begin	You need to have created the `WordCount` project and imported it into Eclipse, which you did in the lab Configuring a Storm Development Environment.
Related lesson:	Configuring a Storm Development Environment

Perform the following steps:

Step 1: Define a New Package

1.1. Right-click on the `WordCount` project and select `New -> Source Folder`.

1.2. Enter `src/main/java` for the name and click the `Finish` button.

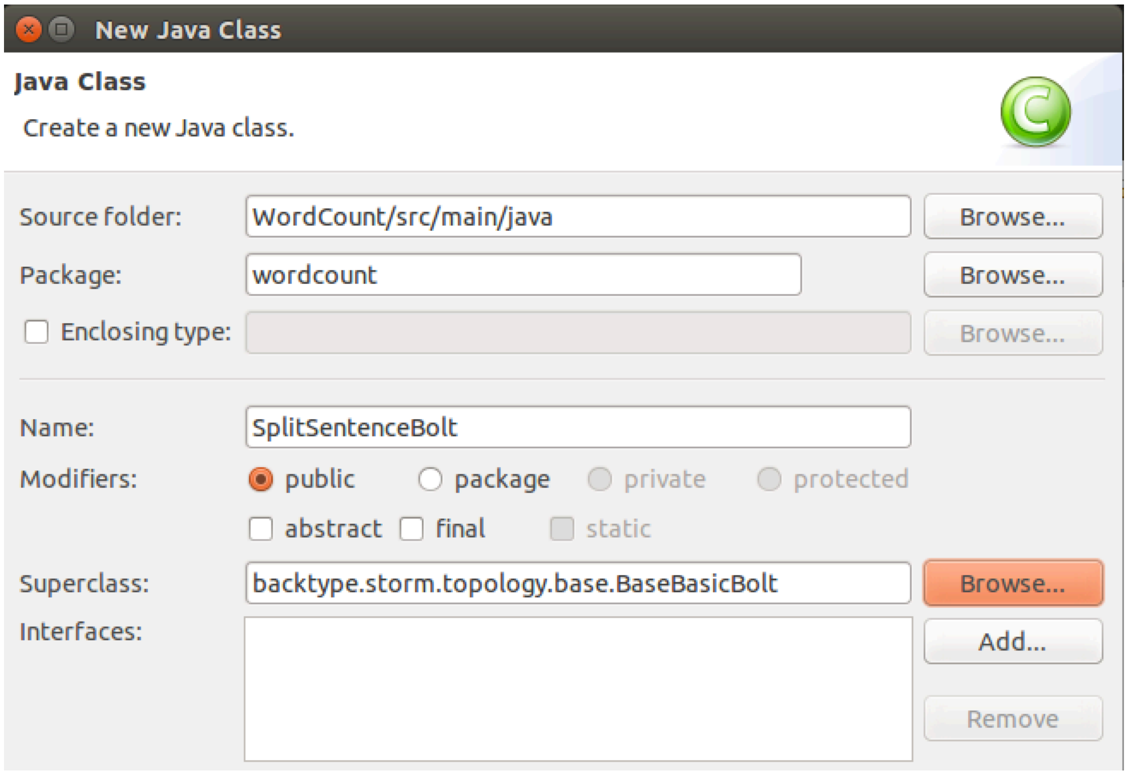
1.3. Right-click on the `src/main/java` folder and select `New -> Package`.

1.4. Enter `wordcount` for the name of the package and click the `Finish` button.

Step 2: Create the `SplitSentenceBolt` Class

2.1. Right-click on the `wordcount` package and select `New -> Class`.

2.2. Name the class `SplitSentenceBolt`. Have it extend the `backtype.storm.topology.base.BaseBasicBolt` class, as shown here:



Source folder: WordCount/src/main/java
Package: wordcount
☐ Enclosing type:
Name: SplitSentenceBolt
Modifiers: [✓ public] package private protected
☐ abstract ☐ final ☐ static
Superclass: backtype.storm.topology.base.BaseBasicBolt
Interfaces:

2.3. Click the `Finish` button and the new class will open in the Eclipse editor.

2.4. Notice your new class has the `execute` and `declareOutputFields` methods declared.

Step 3: Define the `SplitSentenceBolt` Methods

3.1. In the `declareOutputFields` method, declare a single field called "word":

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
     declarer.declare(new Fields("word"));
}
```

3.2. In the `execute` method, split the incoming text into words and emit them. The `StringUtils` class is in the `org.apache.commons.lang` package:

```
public void execute(Tuple input, BasicOutputCollector collector) {
     String[] words = StringUtils.split(input.getString(0));
     for (String word : words) {
          collector.emit(new Values(word));
     }
}
```

3.3. Save your changes to `SplitSentenceBolt.java`.

Step 4: Create the `WordCountBolt` Class

4.1. Add a class named `WordCountBolt` to the `wordcount` package. Have the class extend `backtype.storm.topology.base.BaseBasicBolt`.

4.2. In the declareOutputFields method, declare two fields: “word” and “count”:

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
       declarer.declare(new Fields("word", "count"));
}
```

4.3. Add a `java.util.HashMap` field to `WordCountBolt` to store the words and the count of each word:

```
Map<String, Integer> counts = new HashMap<String, Integer>();
```

4.4. In the execute method, get the word from the input Tuple, then add one to the count of that word. Then emit the word and its count:

```
public void execute(Tuple input, BasicOutputCollector collector) {
       String word = input.getString(0);
       Integer count = counts.get(word);
       if (count == null)
         count = 0;
       count++;
       counts.put(word, count);
       collector.emit(new Values(word, count));
}
```

4.5. Save your changes to `WordCountBolt.java`.

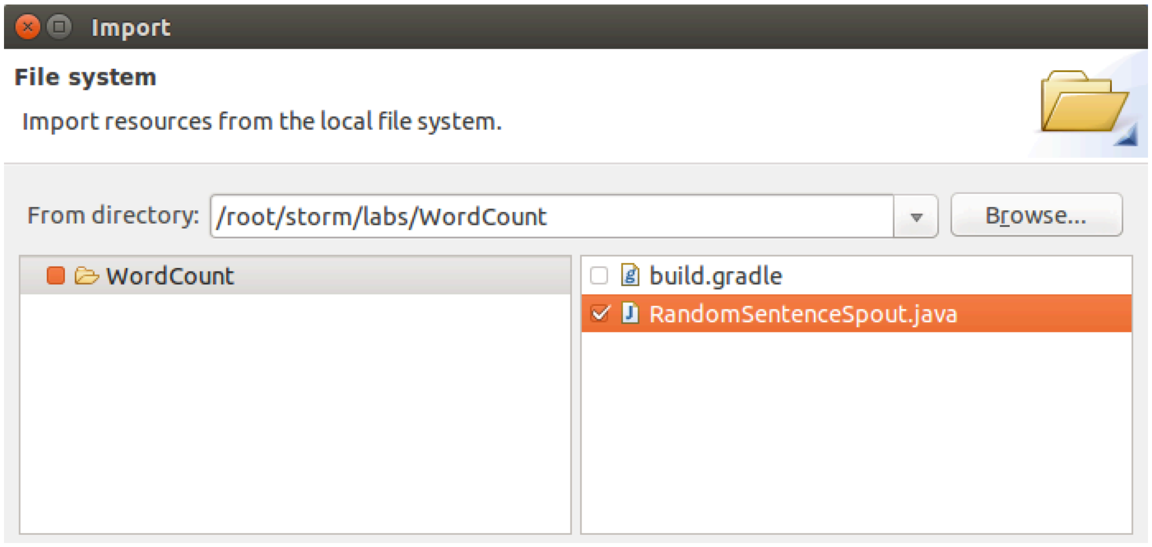
Step 5: Add the `RandomSentenceSpout` class to the Project

5.1. There is a class written for you that generates random sentences. To add this class to the project, right-click on the `wordcount` package folder in Eclipse and select Import….

5.2. Select `File System` and click the `Next` button.

5.3. Click the `Browse…` button and select the `/root/storm/labs/WordCount` folder.

5.4. Place a check mark next to `RandomSentencesSpout.java` and select `Finish`.



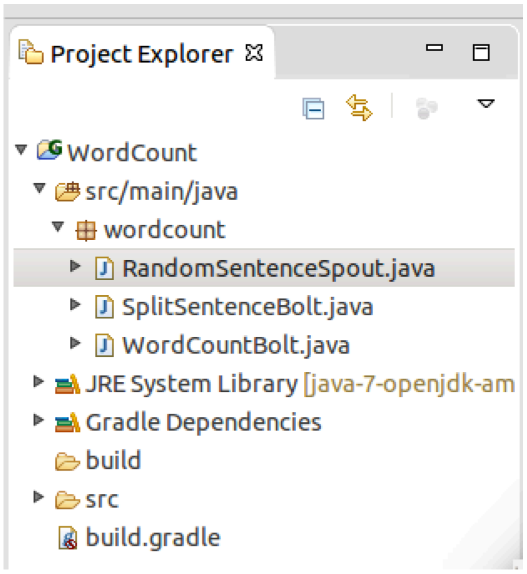
File system

Import resources from the local file system.

From directory: /root/storm/labs/WordCount

- WordCount
- [ ] build.gradle
- [✓] RandomSentenceSpout.java

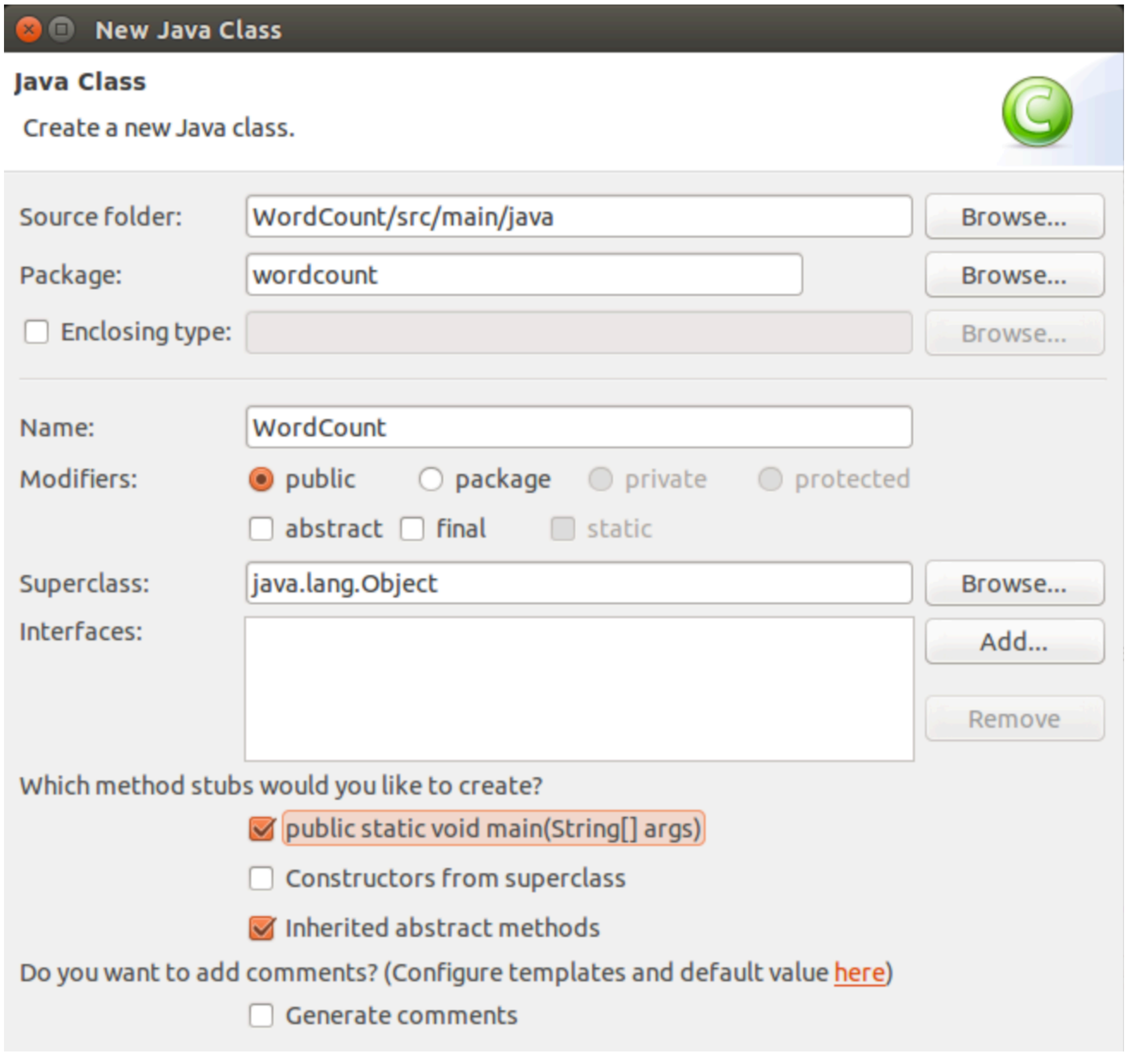
5.5. You should now see `RandomSentenceSpout.java` in the `wordcount` package:



```
WordCount
  src/main/java
    wordcount
      RandomSentenceSpout.java
      SplitSentenceBolt.java
      WordCountBolt.java
  JRE System Library [java-7-openjdk-am
  Gradle Dependencies
  build
  src
  build.gradle
```

Step 6: Write the `WordCount` Program

6.1. Add a class named `WordCount` to the `wordcount` package. Check the box to have Eclipse stub out the `main` method for you:


New Java Class

Java Class

Create a new Java class.

Source folder: WordCount/src/main/java

Package: wordcount

☐ Enclosing type:

Name: WordCount

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

6.2. Within the `main` method of `WordCount`, instantiate a new `TopologyBuilder`:

```
TopologyBuilder builder = new TopologyBuilder();
```

6.3. Add the `RandomSentenceSpout` to the topology with a parallelism hint of 5:

```
builder.setSpout("spout", new RandomSentenceSpout(), 5);
```

6.4. Add the `SplitSentenceBolt` to the topology with a parallelism hint of 8. Assign the "spout" as its input, using a shuffle grouping:

```
builder.setBolt("split",
                new SplitSentenceBolt(),
                8).shuffleGrouping("spout");
```

6.5. Add the `WordCountBolt` to the topology with a parallelism hint of 12. Assign the "split" bolt as its input, and use a fields grouping on the "word" field:

```
builder.setBolt("count",
                new WordCountBolt(),
                    12).fieldsGrouping("split", new Fields("word"));
```

6.6. Instantiate a `Config` object. Set the maximum task parallelism to 3 and the number of workers to 3:

```
Config conf = new Config();
conf.setDebug(true);
conf.setMaxTaskParallelism(3);
conf.setNumWorkers(3);
```

Setting debug to true will allow you to view the word counts in the log files.

6.7. Submit the topology using the `StormSubmitter` class:

```
StormSubmitter.submitTopology("word-count",
                                    conf,
                                    builder.createTopology());
```

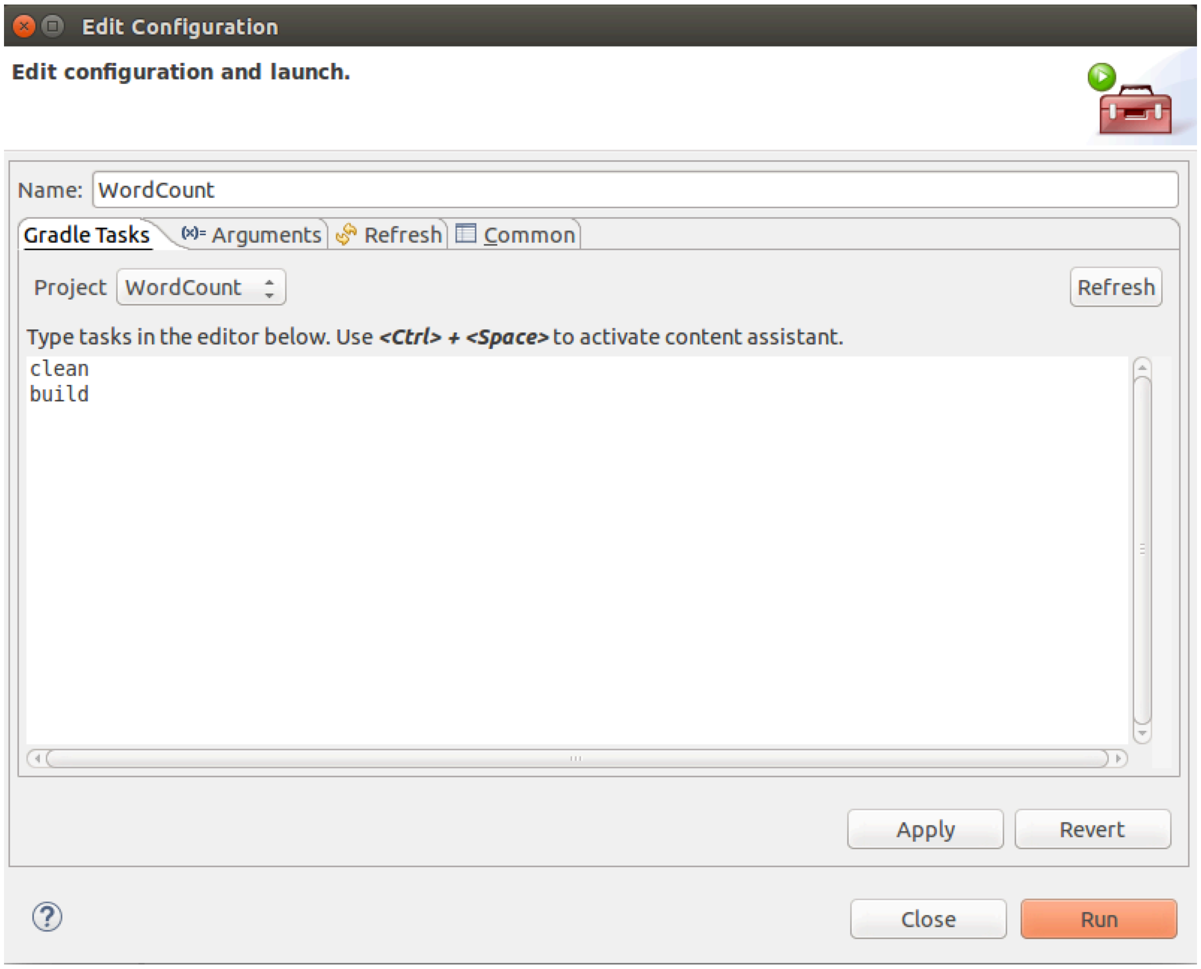
6.8. Add a `try/catch` block around the method call above.

6.9. Save your changes to `WordCount.java`.

Step 7: Build the JAR

7.1. Right-click on the `WordCount` project and select `Run As -> Gradle Build`.

7.2. In the `Edit Configuration` dialog, type in “`clean`” and “`build`” in the list of `Gradle Tasks`, as shown here:



Name: WordCount

Project WordCount

Type tasks in the editor below. Use *<Ctrl> + <Space>* to activate content assistant.

```
clean
build
```

7.3. Click the `Run` button to clean and build the project. Make sure the `WordCount` project builds successfully.

Step 8: Start the Topology

8.1. Open a Terminal window and change to the `WordCount` project folder:

```
# cd ~/storm/workspace/WordCount
```

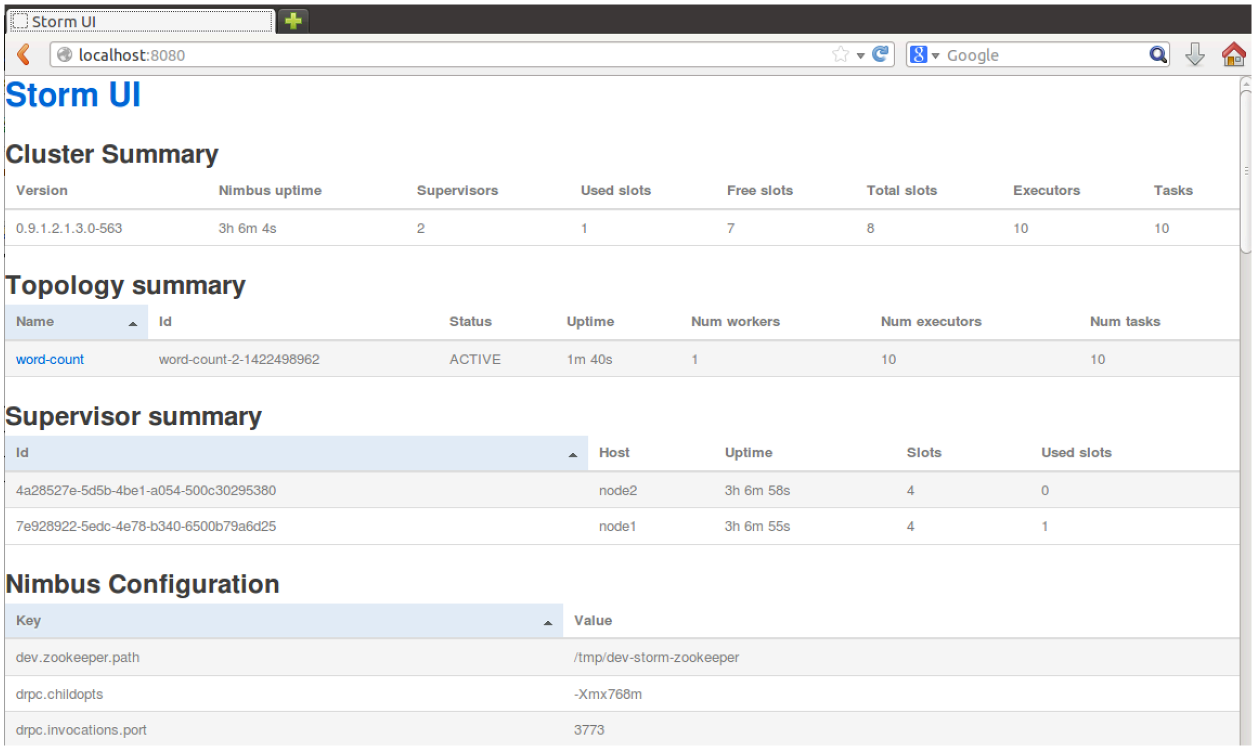
8.2. Run the program by entering the following command all on a single line:

```
# storm jar storm-wordcount.jar wordcount.WordCount
```

Step 9: Verify the Topology is Running

9.1. Point your Web browser to http://localhost:8080/

9.2. You should see the Storm UI and your word-count topology listed as ACTIVE:



Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1.2.1.3.0-563	3h 6m 4s	2	1	7	8	10	10

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
word-count	word-count-2-1422498962	ACTIVE	1m 40s	1	10	10

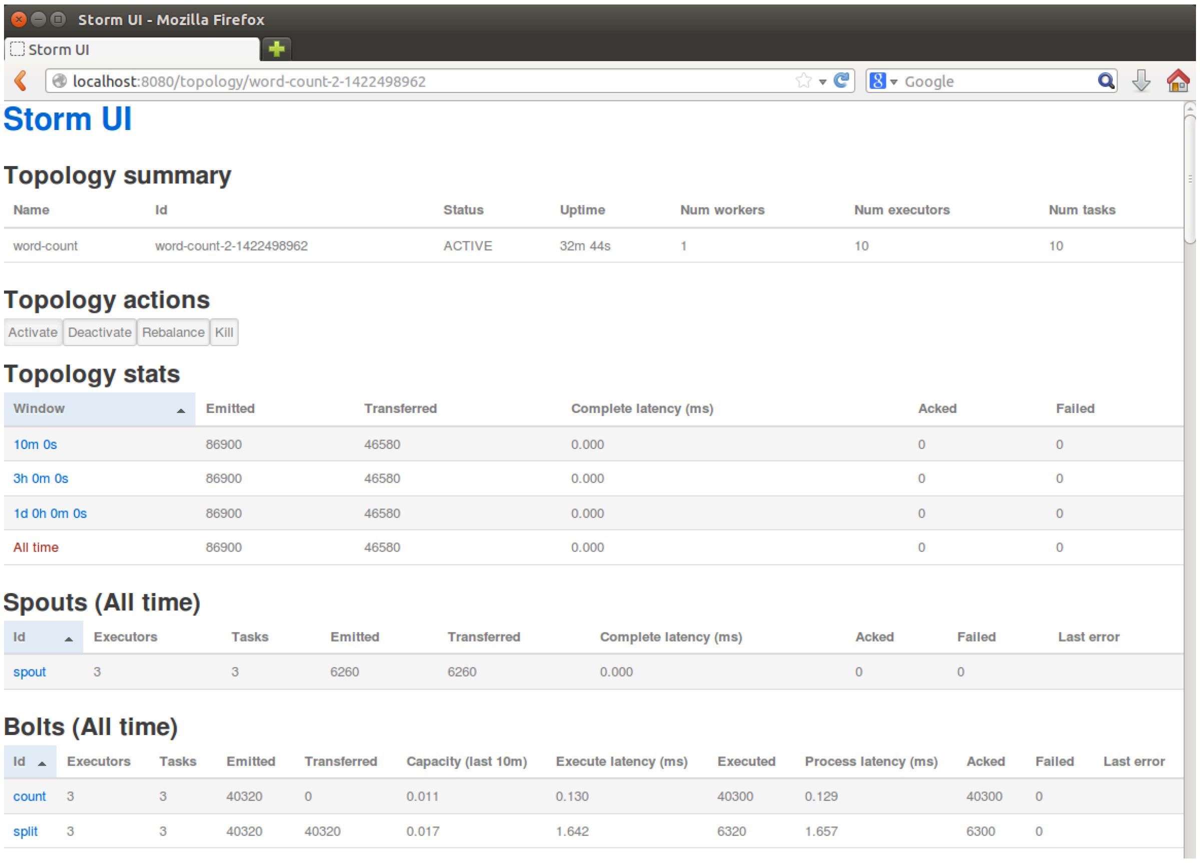
Supervisor summary

Id	Host	Uptime	Slots	Used slots
4a28527e-5d5b-4be1-a054-500c30295380	node2	3h 6m 58s	4	0
7e928922-5edc-4e78-b340-6500b79a6d25	node1	3h 6m 55s	4	1

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx768m
drpc.invocations.port	3773

9.3. Click on the **word-count** link to view the **Topology Summary** page for your word-count topology:



Storm UI - Mozilla Firefox
Storm UI
localhost:8080/topology/word-count-2-1422498962
Google

Storm UI

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
word-count	word-count-2-1422498962	ACTIVE	32m 44s	1	10	10

Topology actions

Activate
Deactivate
Rebalance
Kill

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	86900	46580	0.000	0	0
3h 0m 0s	86900	46580	0.000	0	0
1d 0h 0m 0s	86900	46580	0.000	0	0
All time	86900	46580	0.000	0	0

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
spout	3	3	6260	6260	0.000	0	0	

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Last error
count	3	3	40320	0	0.011	0.130	40300	0.129	40300	0	
split	3	3	40320	40320	0.017	1.642	6320	1.657	6300	0	

9.4. Notice your spout and bolts appear on the summary page. Click on their links to view the details of the “spout”, and also the “count” and “split” bolts.

Step 10: View the Word Counts

10.1. Your **WordCount** topology does not output any results to HDFS, but it does display the words being counted in the log files. To view the log files, start by clicking on the word-count topology from the main Storm UI page.

10.2. Second, click on the “count” bolt to view the **Component Summary** page for the bolt.

10.3. At the bottom of the “count” summary page, notice the port number is a link in the Executors section, and there are three executors. Click on any of the port numbers to view the corresponding log file.

10.4. Look carefully at the output in the log file. Notice that the emitted values are logged, so you will see entries like the following:

```
[INFO] Emitting: count default [two, 4149]
[INFO] Emitting: count default [keeps, 3911]
[INFO] Emitting: count default [and, 8067]
[INFO] Emitting: count default [away, 3911]
```

10.5. Browse some of the other log files from the other executors, which will contain similar entries.

Step 11: Kill the Topology

11.1. There are two ways to kill a topology. In the Storm UI, you can click on the "Kill" button on the Topology Summary page. Or, you run the following command from the terminal:

```
# storm kill word-count
```

11.2. Refresh the Storm UI page, and you should no longer see the word-count topology.

Result: Congratulations, you have just written your first Storm application. Your word-count topology did not output any data to HDFS (or any destination), but you will learn how to accomplish that in future labs.

Lab: Using Storm Multilang Support

Use a Python bolt in a Storm topology

Table 3. About this Lab

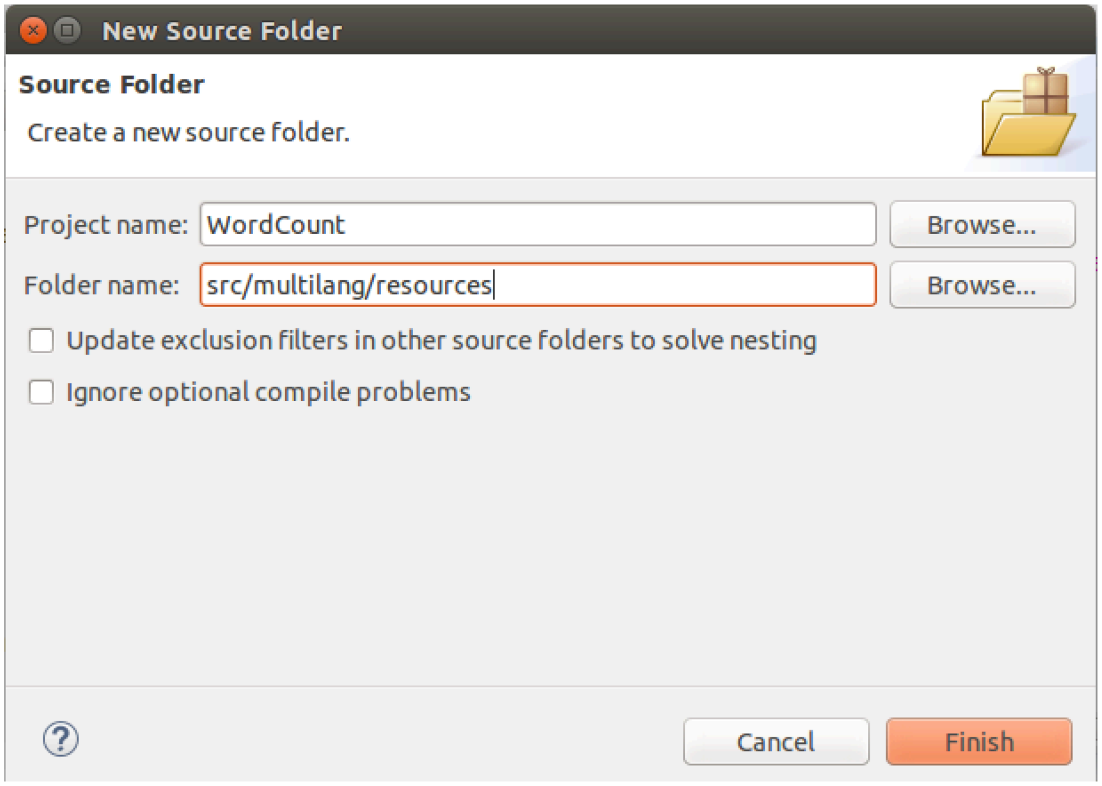
Objective:	Write a WordCount application in Storm that uses a bolt written in Python.
File locations:	`/root/storm/labs/Multilang`
Successful outcome:	
Before you begin	You need to have completed the `WordCount` project from the previous lab.
Related lesson:	Storm WordCount

Perform the following steps:

Step 1: Write a Python Bolt

1.1. The Python code has been written for you. You just need to add it to the Eclipse project. Right-click on the **WordCount** project folder and select `New -> Source Folder`.

1.2. Name the folder `src/multilang/resources`:


New Source Folder
Source Folder
Create a new source folder.

Project name: WordCount

Folder name: src/multilang/resources

Browse...
Browse...
Update exclusion filters in other source folders to solve nesting
Ignore optional compile problems
Cancel
Finish

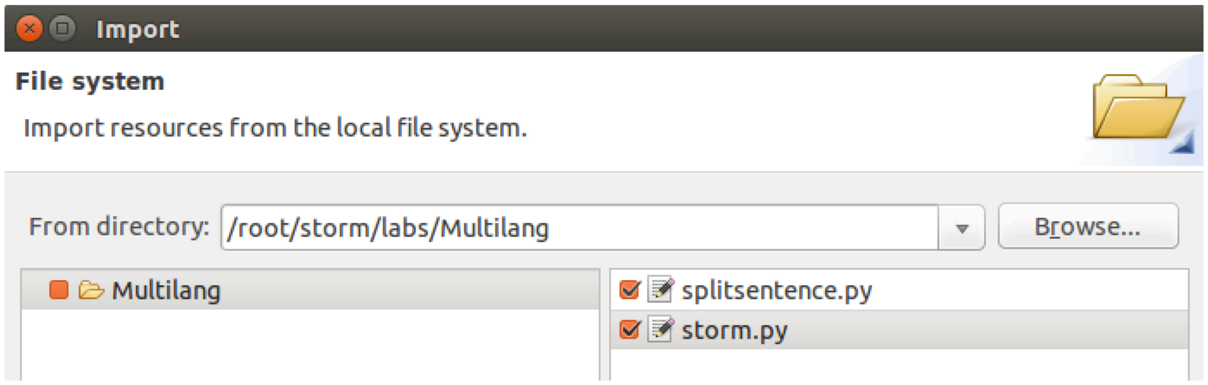
1.3. Click the `Finish` button to create the folder.

1.4. Right-click on the new `src/multilang/resources` folder and select `Import`….

1.5. Select **File System** and click the `Next` button.

1.6. Click the `Browse…` button and select the `/root/storm/labs/Multilang` folder.

1.7. Check the `splitsentences.py` and `storm.py` files, then click the `Finish` button:



From directory: /root/storm/labs/Multilang

Multilang

✓ splitsentence.py

✓ storm.py

1.8. Expand the `src/multilang/resources` folder in Eclipse, then right-click on `splitsentences.py` and select `Open With -> Text Editor` to view the script:

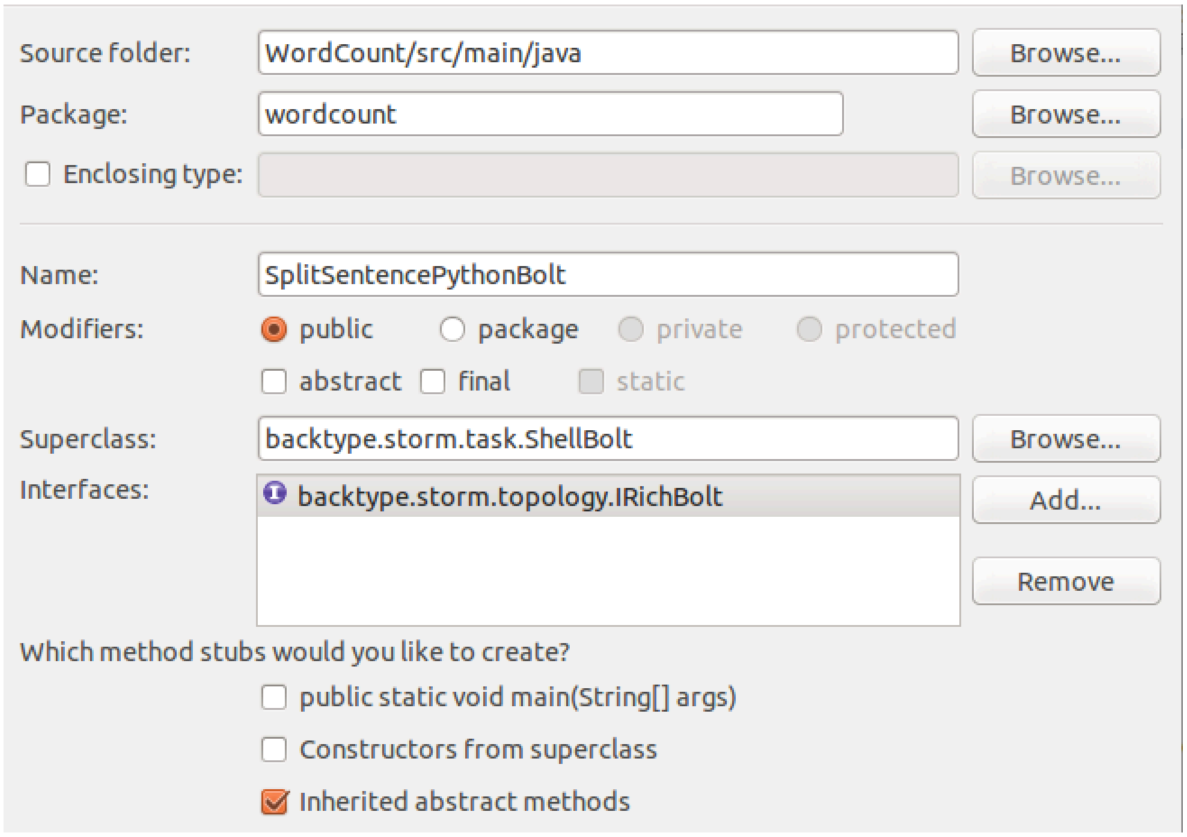
```
import storm

class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
            storm.emit([word])

SplitSentenceBolt().run()
```

1.9. Notice the code defines a Python class named `SplitSentenceBolt` which contains a `process` function that splits a line of text into words using a space as a delimiter.

Step 2: Define a ShellBolt

2.1. Right-click on the `wordcount` package and select `New -> Class`.

2.2. Add a new class named `SplitSentencePythonBolt` that extends `ShellBolt` and implements `IRichBolt`, as shown below:



Source folder:	WordCount/src/main/java	Browse...
Package:	wordcount	Browse...
☐ Enclosing type:		Browse...
Name:	SplitSentencePythonBolt	
Modifiers:	[✓ public] ○ package ○ private ○ protected	
	☐ abstract ☐ final ☐ static	
Superclass:	backtype.storm.task.ShellBolt	Browse...
Interfaces:	backtype.storm.topology.IRichBolt	Add... / Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

2.3. In the `declareOutputFields` method, declare a single field named “word”:

```
declarer.declare(new Fields("word"));
```

2.4. Add a no-argument constructor that passes the Python script name to the parent constructor:

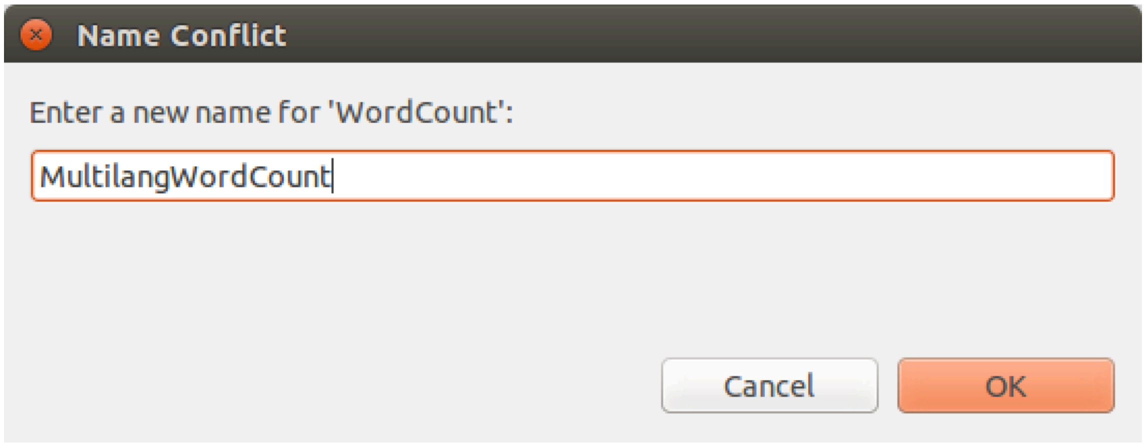
```
public SplitSentencePythonBolt() {
        super("python", "splitsentence.py");
}
```

2.5. Save your changes to `SplitSentencePythonBolt.java`.

Step 3: Copy the WordCount Class

3.1. The `WordCount` class only needs a couple of minor changes, so you will use it as a starting point for configuring the `ShellBolt`. Click on `WordCount.java` in the **Project Explorer** view of Eclipse and press **Ctrl+c** to copy it.

3.2. Press **Ctrl+v** to paste the file and a "**Name Conflict**" dialog window appears. Change the name to `MultilangWordCount`:



Enter a new name for 'WordCount':

MultilangWordCount

3.3. Press `OK` to close the dialog.

3.4. Double-click on `MultilangWordCount.java` to open the file in the Eclipse editor.

Step 4: Configure the ShellBolt

4.1. Within the main method of MultilangWordCount, change the `SplitSentenceBolt` to a `SplitSentencePythonBolt`.

4.2. In the call to `submitTopology`, change the name of the topology from "`word-count`" to "`multilang-word-count`".

4.3. Save your changes to `MultilangWordCount.java`.

Step 5: Run the Topology

5.1. Right-click on the **WordCount** project and select `Run As -> Gradle Build` to rebuild the JAR file.

5.2. Run the Storm JAR with the following command:

```
# storm jar storm-wordcount.jar wordcount.MultilangWordCount
```

Step 6: Verify the Topology

6.1. Go to the Storm UI and verify that the multilang-word-count topology is ACTIVE.

6.2. View the log files of the "count" bolt to verify that it is processing words and keeping track of their count.

Step 7: Kill the Topology

7.1. When you are done verifying that the topology is running successfully, kill it.

Result: In this lab, you deployed a Storm topology that uses a bolt written in Python.

Lab: Processing Log Files

Write a Storm topology

Table 4. About this Lab

Objective:	Write a Storm topology for processing log file entries.
File locations:	/root/storm/labs/LogTopology
Successful outcome:	
Before you begin	Make sure your cluster is up and running, and open Eclipse.
Related lesson:	Storm WordCount

Perform the following steps:

Step 1: Locate the Eclipse Project

1.1. Open Eclipse. You should see a project named `LogTopology`.

1.2. Notice the project contains a package named `log` and a spout named `LogFileSpout` that is written for you. View the code of the `LogFileSpout` class. Notice it reads in and then emits the lines of text from a given file.

1.3. Put the file `~/storm/labs/LogTopology/node1.log` into HDFS into the /user/root folder:

```
# hadoop fs -put ~/storm/labs/LogTopology/node1.log /user/root/
```

Step 2: Write the LogSplitterBolt

2.1. Add a new class named `LogSplitterBolt` to the `log` package that extends `BaseBasicBolt`.

2.2. The lines of text emitted from the LogFileSpout will look like the following (all on a single line):

```
2015-01-05 03:10:02,163 INFO  impl.MetricsSinkAdapter
(MetricsSinkAdapter.java:start(195)) - Sink ganglia started
```

2.3. Write the `LogSplitterBolt` so that it emits three values: the date, log level, and message. You will need to split the incoming line of text into these three values. HINT: The date is the first 24 characters; then you can split the remaining text on the first whitespace (because the log level will not have any spaces in it). The following code might help:

```
String date = input.getString(0).substring(0, 23);
String errormessage = input.getString(0).substring(24);
String [] words = errormessage.split("\\s+");
collector.emit(new Values(date, words[0], words[1]));
```

Step 3: Write the LogFilter Bolt

3.1. Write a bolt called `LogFilterBolt` that filters the log message given a log level. You will need to add a field to the class that stores the log level being searched, and add a constructor that initializes that field. The bolt should only emit the messages that match the given filter.

For example, if the search string is "WARN", then the `LogFilterBolt` should only emit messages of type WARN, and only the message (not the date or log level).

Step 4: Build the Topology

4.1. Write a class named `LogTopology` that contains the `main` method to build the Storm topology. Build the topology using the `LogFileSpout`, and then attach the `LogSplitterBolt` and then the `LogFilterBolt`. Create a `LogFileSpout` that reads the log entries from `node1.log`. Pass in the file `"hdfs://namenode:8020/user/root/node1.log"` in the `LogFileSpout` constructor. Use "WARN" as the log level filter.

4.2. Run the topology locally (as opposed to on the cluster):

```
LocalCluster cluster = new LocalCluster();
         cluster.submitTopology("logfilter", conf, builder.createTopology());
```

Step 5: Run the Topology in Eclipse

5.1. Right-click on the LogToplogy class and select `Run As -> Java Application`. You should see the output in the Eclipse Console window.

Step 6: Verify the Output

6.1. Scroll through the output and make sure your logs are being filtered properly. Add `System.out.println` calls to your code if you want to verify the messages being emitted by your bolts.

Result: In this lab, you wrote a Storm topology that uses two bolts: one to split a log message into separate strings, and a second that filters the log messages on a given log level. You also saw how to run a Storm topology locally, which is useful when developing and testing a topology.

Lab: Integrating Kafka with Storm

Consuming Kafka messages in a Storm Topology

Table 5. About this Lab

Objective:	Write a Storm topology that consumes messages from a Kafka spout.
File locations:	`/root/storm/labs/Kafka`
Successful outcome:	You will be able to send messages from the command line to a Kafka topic, and your Storm topology will read the messages.
Before you begin	Start the cluster and open Eclipse.
Related lesson:	n/a

Perform the following steps:

Step 1: SSH onto `node1`

1.1. Open a `Terminal` window.

1.2. A Kafka server is already running on `node1` of your cluster. SSH onto `node1`:

```
# ssh node1
```

The password is `hadoop`.

Step 2: Define a Topic

2.1. Use the `kafka-topics.sh` script (which is in the `PATH` of `node1`) to create a new topic named `my_topic`. ZooKeeper is running on `namenode:2181`. Use a replication factor of 1, and also use 1 partition.

2.2. Use the `--list` option of `kafka-topics.sh` to verify your topic was created successfully.

Step 3: Test the Topic

3.1. Run the following command, which allows you to send messages to a topic from the console:

```
# kafka-console-producer.sh --broker-list localhost:9092 --topic my_topic
```

3.2. Type in some text and hit `Enter`. Each line of text you enter sends a message to `my_topic`.

3.3. Hit `Ctrl+c` to stop the process.

3.4. To consume your messages, run the following program:

```
# kafka-console-consumer.sh --zookeeper namenode:2181 --topic my_topic \
--from-beginning
```

3.5. You should see the messages that you typed in previously.

3.6. Hit `Ctrl+c` to stop the process.

Step 4: Open the Eclipse Project

4.1. Locate the `Kafka` project in Eclipse. Notice it contains a package named `kafka` and two classes: `KafkaTopology` and SimpleBolt.

Step 5: Create a KafkaSpout

5.1. Open the file `KafkaTopology.java`.

5.2. Within the `main` method, create a new `KafkaSpout` object. You will need a `ZkHosts` object for `"namenode:2181"`. The name of the topic is "`my_topic`", and use "`/my_topic`" for the ZooKeeper location for storing the offset. Use any string you want for the topic ID.

Step 6: Create a Topology

6.1. Instantiate a new `TopologyBuilder` object.

6.2. Add your `KafkaSpout` object to the topology. Give it the name "`my_topic_spout`" and set the number of tasks to 1.

6.3. Add a new `SimpleBolt` object to the topology named "`simplebolt`".

Step 7: Create a `LocalCluster`

7.1. Instantiate a new `backtype.storm.Config` object.

7.2. Instantiate a new `backtype.storm.LocalCluster` object.

7.3. Submit your topology to the `LocalCluster` object, which will run the code locally instead of on a cluster.

7.4. Save your changes to `KafkaTopology.java`.

Step 8: Run the Topology

8.1. Right-click on `KafkaTopology.java` and select `Run As -> Java Application`. The output should appear in the `Console` tab of Eclipse.

8.2. Run the `kafka-console-producer.sh` process and send some messages to `my_topic`. You should see the messages displayed in the `Console` in Eclipse as they get read by your spout and processed by the `SimpleBolt` object.

Result: You Storm topology is consuming messages from a Kafka topic.

Lab: Using Trident

Writing a Simple Trident Application

Table 6. About this Lab

Objective:	Write a Trident word count application.
File locations:	`/root/storm/labs/TridentWordCount`
Successful outcome:	The count of words in a text document.
Before you begin	Start the cluster and open Eclipse.
Related lesson:	n/a

Perform the following steps:

Step 1: Review the Project

1.1. Open the `TridentWordCount` project in Eclipse.

1.2. Notice a spout named `LineReaderSpout` has been written for you in the `trident` package. This spout reads in a text document and emits each line of text as a tuple.

Step 2: Define a Topology

2.1. Open `TridentWordCount.java` in the `trident` package.

2.2. Within `main`, create a new `Config` object and assign a new property named "`inputFile`" as the name of the file to process. In our example, we will use a simple file that is in the `/root` folder:

```
Config config = new Config();
config.put("inputFile", "/root/install_course.sh");
```

2.3. Instantiate a new `LineReaderSpout` object.

2.4. Instantiate a new `TridentTopology` object.

2.5. Attach the `LineReaderSpout` object as a stream to the topology.

2.6. Use the `each` operation to split the incoming lines of text into words.

2.7. Use the `groupBy` operation to group the result by word.

2.8. Use the `aggregate` operation with a new `Count` object to count the number of values in each group.

2.9. Use the each operation with a new `Debug` instance to display the result of each word counted:

```
.each(new Fields("word","count"), new Debug())
```

Your Trident topology should now be complete.

Step 3: Submit the Topology

3.1. Instantiate a `LocalCluster` object and submit your Trident topology along with the `Config` object.

Step 4: Run the Application

4.1. Save your changes to `TridentWordCount.java`.

4.2. Run `TridentWordCount` as a Java application from within Eclipse.

4.3. Watch the output in the Console window. The application should start successfully, but it may take a minute or so for the actual word counts to be displayed.

Result: Your Trident topology should output the word count of the given text input file.

Lab: Using Trident with Kafka

Writing a Trident Toplogy that uses Kafka

Table 7. About this Lab

Objective:	Write a Trident topology that consumes messages from a Kafka topic.
File locations:	`/root/storm/labs/TridentKafkaTopology`
Successful outcome:	The Trident topology reads messages from the Kafka topic, and a query displays the count of specified words.
Before you begin	Start the cluster and open Eclipse.
Related lesson:	n/a

Perform the following steps:

Step 1: Import the Project

1.1. Open the `TridentKafkaTopology` project in Eclipse.

1.2. Open the file `SentenceProducer.java` in the `kafka` package.

1.3. Notice the `SentenceProducer` application publishes the same sentences 10,000 times to a topic named "`sentences`".

Step 2: Define the Topic

2.1. From a Terminal window, define a new topic named `sentences` using the ZooKeeper instance on `namenode:2181`. Use 1 for the replication factor and the number of partitions.

Step 3: Publish Messages to the Topic

3.1. In Eclipse, run the `SentenceProducer` application by right-clicking on `SentenceProducer.java` and selecting `Run As -> Java Application`. (There is no output, so just check the Console for any error messages.)

Step 4: Configure the TridentKafkaConfig Instance

4.1. Open `TridentKafkaWordCount.java`.

4.2. Locate the static `buildTopology` method. Notice there is a comment showing where to add your new code for this lab.

4.3. Instantiate a new `TridentKafkaConfig` object. Use `namenode:2181` for the `ZkHosts`, and the name of the topic is "`sentences`".

4.4. Set the scheme of the `TridentKafkaConfig` object to a new `StringScheme`:

```
spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
```

4.5. Set the `forceFromStart` property to `true`:

```
spoutConfig.forceFromStart = true;
```

4.6. Instantiate a new `TransactionalTridentKafkaSpout` instance, using your `TridentKafkaConfig` instance:

```
TransactionalTridentKafkaSpout kafkaSpout =
      new TransactionalTridentKafkaSpout(spoutConfig);
```

Step 5: Configure a TridentTopology

5.1. Instantiate a new `TridentTopology` instance.

5.2. Configure the topology to process the stream from the `kafkaSpout`:

```
TridentState wordCounts = topology.newStream("spout", kafkaSpout)
              .parallelismHint(16)
              .each(new Fields("str"), new Split(), new Fields("word"))
              .groupBy(new Fields("word"))
              .persistentAggregate(new MemoryMapState.Factory(), new Count(),
new Fields("count"))
              .parallelismHint(16);
```

5.3. Now define a distributed query on the word counts, using the `LocalDRPC` reference passed in to the method:

```
topology.newDRPCStream("words", drpc)
            .each(new Fields("args"), new Split(), new Fields("word"))
            .groupBy(new Fields("word"))
            .stateQuery(wordCounts, new Fields("word"), new MapGet(), new
Fields("count"))
            .each(new Fields("count"), new FilterNull())
            .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

5.4. Build the topology and return it from the buildTopology method:

```
return topology.build();
```

5.5. Save your changes to `TridentKafkaWordCount.java`.

Step 6: Submit the Topology

6.1. The `main` method is written for you. Notice it invokes the static `buildTopology` method and submits it to a local cluster. Then a `for` loop executes the `LocalDRPC` query 100 times, sleeping for one second inbetween queries.

6.2. Right-click on `TridentKafkaWordCount` and select `Run As -> Java Application`.

6.3. Check the output in Console. As the query executes, you should its output and the number of occurrences of the strings "good" and "happy".

6.4. Feel free to change the words in the query and run the process again to view the count of other words in the stream.

Result: You now have a Trident topology processing messages from a Kafka spout.