Lecture 08

Spark Streaming

Zoran B. Djordjević

Spark Streaming Sources

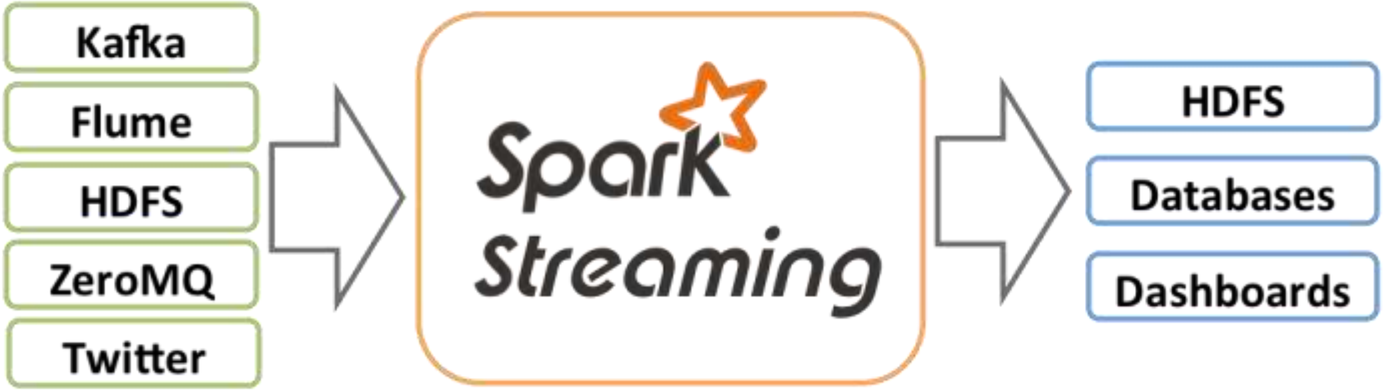
- This lecture follows Apache Spark

"Spark Streaming Programming Guide"
http://spark.apache.org/docs/latest/streaming-programming-guide.html

- Most of what we said about Hadoop and Spark so far had a batch processing model in mind. You will collect large volumes of data and then use Hadoop or Spark to analyze those data.
- Many scenarios in IT and other industries cannot wait for decisions to be made after many hours, days or months. Decisions often have to be made on the fly, right now, or with a very small latency. Spark Streaming attempts to address that need.
- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Spark Streaming Sources & Destinations

- Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be written to HDFS, RDBMS and NoSQL databases, regular file systems.
- Data can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Processed data can be pushed out to filesystems, databases, and live dashboards. One can apply Spark's machine learning and graph processing algorithms on data streams.

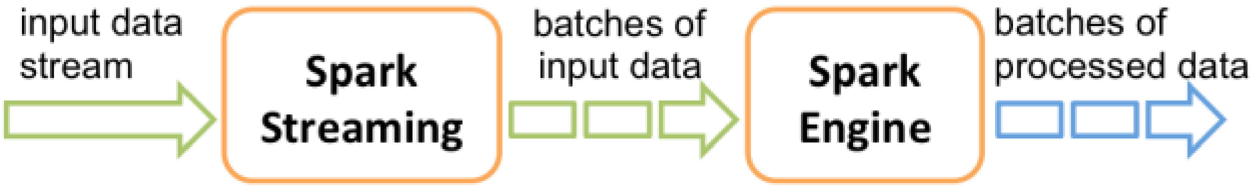


Discretized Streams

- Spark has the concept of RDDs. Spark Streaming provides an abstraction called *DStreams*, or *Discretized Streams*.
- A DStream is a sequence of data arriving over time, i.e. a sequence of RDDs arriving at each time step (hence the name "discretized").
- DStreams can be created from various input sources, such as Flume, Kafka, AWS Kinesi, Twitter or HDFS.
- DStreams offer two types of operations: *transformations*, which yield a new DStream, and *output operations*, which write data to an external systems, HDFS, Dashboards or Databases.
- DStreams provide many of the same operations available on RDDs, plus new operations related to time, such as sliding windows.
- Unlike batch programs, Spark Streaming applications need additional setup in order to operate 24/7.
- *Checkpointing* is the main mechanism Spark Streaming provides for reliable operation and restart operations. Checkpointing stores data in a reliable file system such as HDFS.

Internals of Data Stream

- Spark Streaming receives live input data streams and divides the data into “micro” batches, which are then processed by the Spark engine to generate the final stream of results in batches.



- Discretized stream or `DStream` consumes a continuous stream of data but treats it in a discretized manner.
- One can write Spark Streaming programs in Scala, Java or Python.
- There are a few API calls that are either different or yet not available in Python.

JavaNetworkWordCount.java

- We start Spark Streaming program by creating a `JavaStreamingContext` object, which is the main entry point for all streaming functionality. In the following we are creating a local `StreamingContext` with two execution threads, and a batch interval of 1 second.

```
import org.apache.spark.*;
import org.apache.spark.api.java.function.*;
import org.apache.spark.streaming.*;
import org.apache.spark.streaming.api.java.*;
import scala.Tuple2;
public class JavaNetworkWordCount
// Local StreamingContext with two working thread and batch interval of 1 second
SparkConf conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount");
JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(1));
```

- Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

```
JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

- `JavaReceiverInputDStream` represents the stream of data received from the data server.
- Each record in this stream is a line of text. Then, we split the lines by space into words.

```
JavaDStream<String> words = lines.flatMap(
  new FlatMapFunction<String, String>() {
    @Override public Iterable<String> call(String x) {
      return Arrays.asList(x.split(" "));
    } });
```

- flatMap is a DStream operation that creates a new DStream with multiple records from each record in the source DStream. Each line will be split into multiple words. This transformation is performed by FlatMapFunction object. Java convenience classes define DStream transformations.

JavaNetworkWordCount, counting words

- Subsequently, we count those words

```
JavaPairDStream<String, Integer> pairs = words.map(
  new PairFunction<String, String, Integer>() {
    @Override public Tuple2<String, Integer> call(String s) throws
Exception {
      return new Tuple2<String, Integer>(s, 1);
    }
  });
JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    @Override public Integer call(Integer i1, Integer i2) throws Exception
{
      return i1 + i2;
    }
  });
wordCounts.print();     // Print a few of the counts to the console
```

- The words DStream is mapped (one-to-one transformation) to a DStream of (word, 1) pairs, using a `PairFunction` object. Then, it is reduced to get the frequency of words in each batch of data, using a `Function2` object.
- Finally, `wordCounts.print()` will print a few of the counts generated every second.

Start and Exit processing

- When the above lines are executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet.
- To start the processing after all the transformations have been setup, we finally call

```
jssc.start();                 // Start the computation
jssc.awaitTermination();      // Wait for the computation to terminate
```

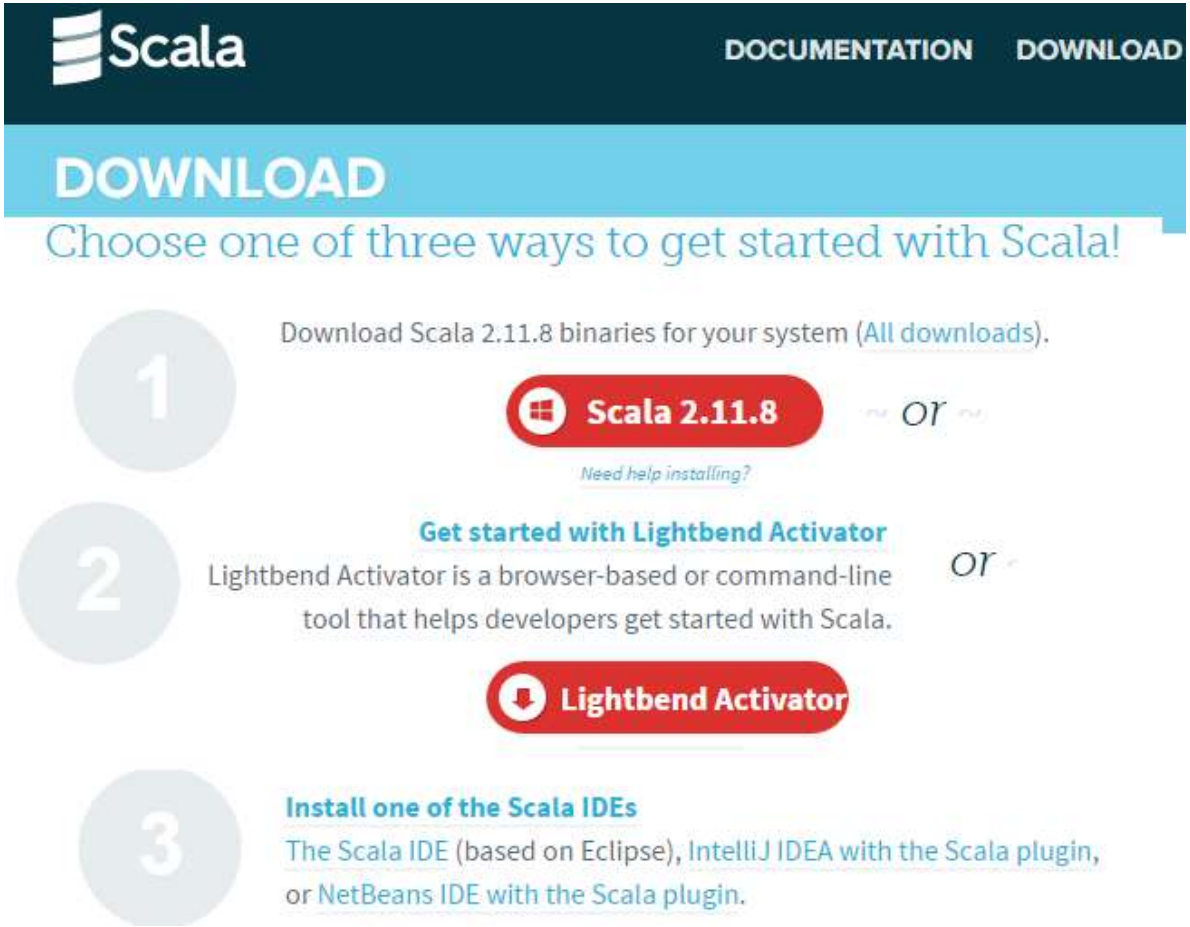
Spark Examples

- Spark has a bunch examples. You can get them all if you download Spark source code from `spark.apache.org/donwloads.html`
- Select and download package type: `Source Code`. The tar file will have the name: `spark-1.6.1.tgz.`
- On Linux you open that archive with command
- `$ tar -zxvf spark-1.6.1.tgz`
- On Windows you can use 7-zip twice.
- When you expand (untar) that file you will find an `examples` directory with `pom.xml` file in it.
- `pom.xml` is Mavern directives file and the whole `examples` directory contains artifacts of a Maven project.
- We will move that directory to the shared directory visible to our VM

Setting Eclipse Up

- We are running Eclipse Luna on Cloudera VM. We would prefer to have Java 1.8.
- We are stuck with Java 1.7. Next we have to install Scala 2.11.8
- For Scala 2.11.8 we go to http://www.scala-lang.org/download/
- Download Scala 2.11.8 binaries for your system (All downloads).
- We will also have to install one of the Scala IDEs (The Scala IDE based on Eclipse)

http://www.scala-lang.org/download/



Scala DOCUMENTATION DOWNLOAD

DOWNLOAD

Choose one of three ways to get started with Scala!

1 Download Scala 2.11.8 binaries for your system (All downloads).

Scala 2.11.8 ~ or ~

Need help installing?

2 Get started with Lightbend Activator

Lightbend Activator is a browser-based or command-line tool that helps developers get started with Scala. or

Lightbend Activator

3 Install one of the Scala IDEs

The Scala IDE (based on Eclipse), IntelliJ IDEA with the Scala plugin, or NetBeans IDE with the Scala plugin.

Scala Binaries

- Eclipse or some similar IDE (like IntelliJ) might want to have Scala code and execution environment which are not wrapped inside Eclipse. Perhaps you will need to modify code and run it from the command line. In that case untar `scala-2.11.8.tgz`, set SCALA_HOME and modify the PATH

```
$ tar -zxvf scala-2.11.8.tgz
```

- We are listing the whole environment setup here since you might have to do it on your new clusters of machine, in the Cloud, for example.

Install sbt

```
$ curl https://bintray.com/sbt/rpm/rpm | \
   sudo tee /etc/yum.repos.d/bintray-sbt-rpm.repo
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
146   146    0   146    0     0    218      0 --:--:-- --:--:-- --:--:--  1659
#bintray--sbt-rpm - packages by  from Bintray
[bintray--sbt-rpm]
name=bintray--sbt-rpm
baseurl=http://dl.bintray.com/sbt/rpm
gpgcheck=0
enabled=1

$ sudo yum install sbt
[cloudera@quickstart bin]$ which sbt
/usr/bin/sbt
```

Set Environmental Variables, Linux

- **JAVA_HOME**

```
$ echo $JAVA_HOME
/usr/java/jdk1.7.0_67-cloudera
JAVA_HOME=/usr/java/jdk1.7.0_67-cloudera
```

- **HADOOP_HOME**
- Spark needs to know about Hadoop's HDFS libraries
- On Linux we have `hadoop-common.jar` in directory:

```
HADOOP_HOME=/usr/lib/hadop
```

- **SBT_HOME**
- Download sbt installer from

```
http://www.scala-sbt.org/0.13/docs/Installing-sbt-on-Linux.html
SBT_HOME=/usr/share/sbt-launcher-packaging
```

- **SCALA_HOME**, we moved untared scala distribution to `/usr/lib`

```
$ sudo chown -R root:root /usr/lib/scala-2.11.8
SCALA_HOME=/usr/lib/scala-2.11.8/
```

```
M2_HOME=/usr/local/apache-maven/apache-maven-3.0.4
```

- Place above variables in your `.bash_profile` and source that file if you need to do something with Spark on the command prompt

Set Environmental Variables, Windows

JAVA_HOME=`C:\Program Files\Java\jdk1.8.0_73`

M2_HOME=`E:\Programs\maven-3.2.2`

HADOOP_HOME=`E:\CLASSES\code\hadoop-common-2.2.0-bin-master`

- Spark needs to know about Hadoop's HDFS libraries
- For Windows you need: `hadoop-common-2.2.0-bin-master.zip` which can be fetched from

https://codeload.github.com/srccodes/hadoop-common-2.2.0-bin/zip/master

- That archive contains `hadoop.lib` and `hadoop.dll` files.

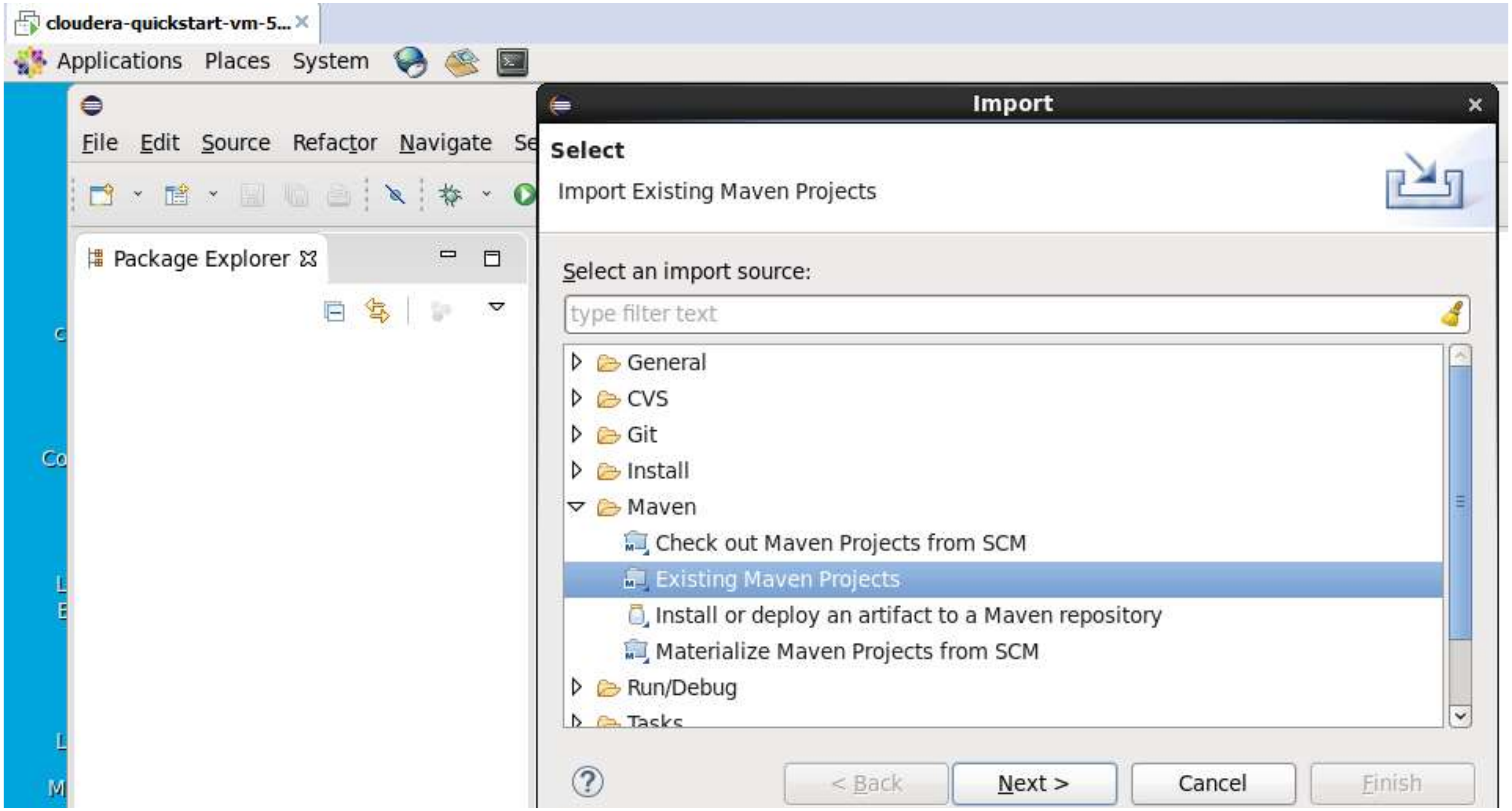
SBT_HOME=`E:\Programs\sbt`

- Download sbt installer from http://www.scala-sbt.org/0.13/docs/Installing-sbt-on-Windows.html

SCALA_HOME=`C:\Program Files (x86)\scala`

Setup Eclipse, Spark Streaming Examples

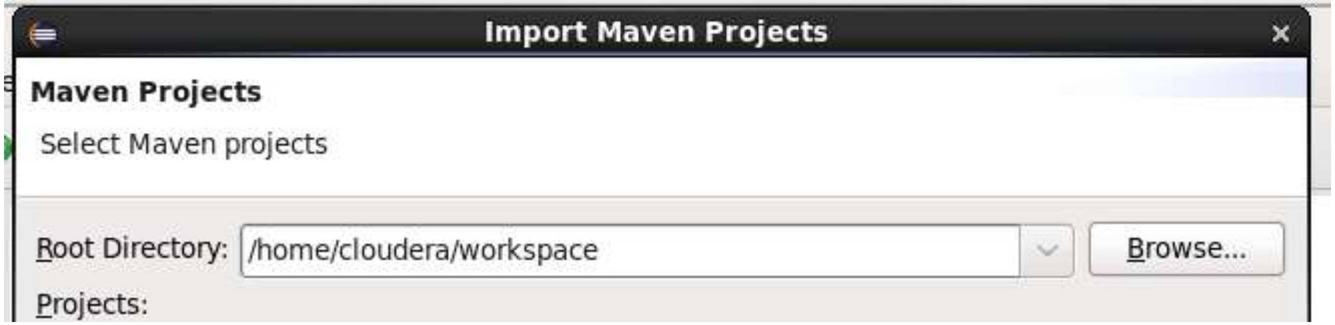
- When we open Eclipse, we first create a project. To run Spark Streaming examples we will not create a new project but rather do the following:
- File > Import > Maven > Existing Maven Projects. Select Next



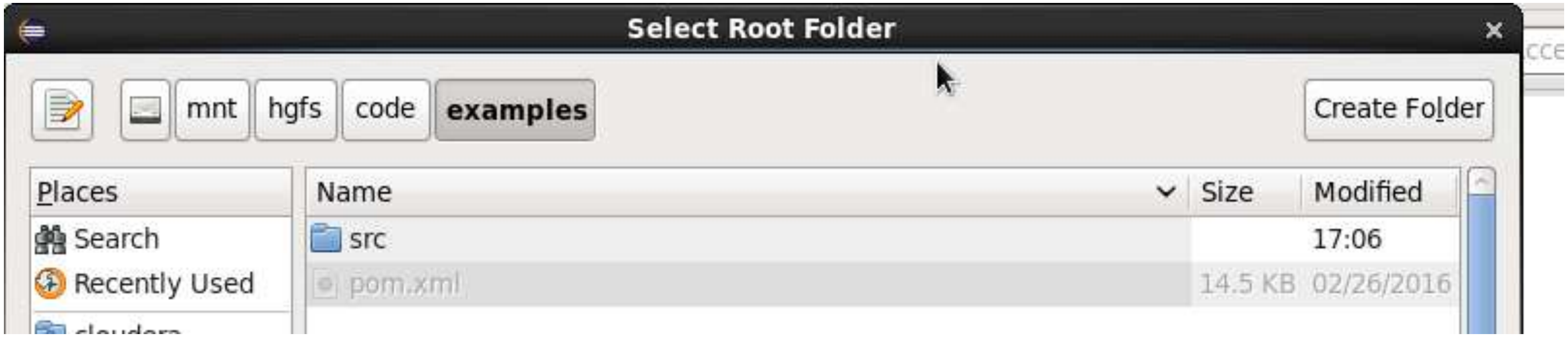
cloudera-quickstart-vm-5...
Applications Places System
File Edit Source Refactor Navigate Se
Package Explorer
Import
Select
Import Existing Maven Projects
Select an import source:
type filter text
General
CVS
Git
Install
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM
Run/Debug
Tasks
< Back
Next >
Cancel
Finish

Import Maven Project

- Browse to the folder where your examples resides. In my cases that is the shared folder `/mnt/hgfs/code` with `examples` subdirectory.



Root Directory: /home/cloudera/workspace

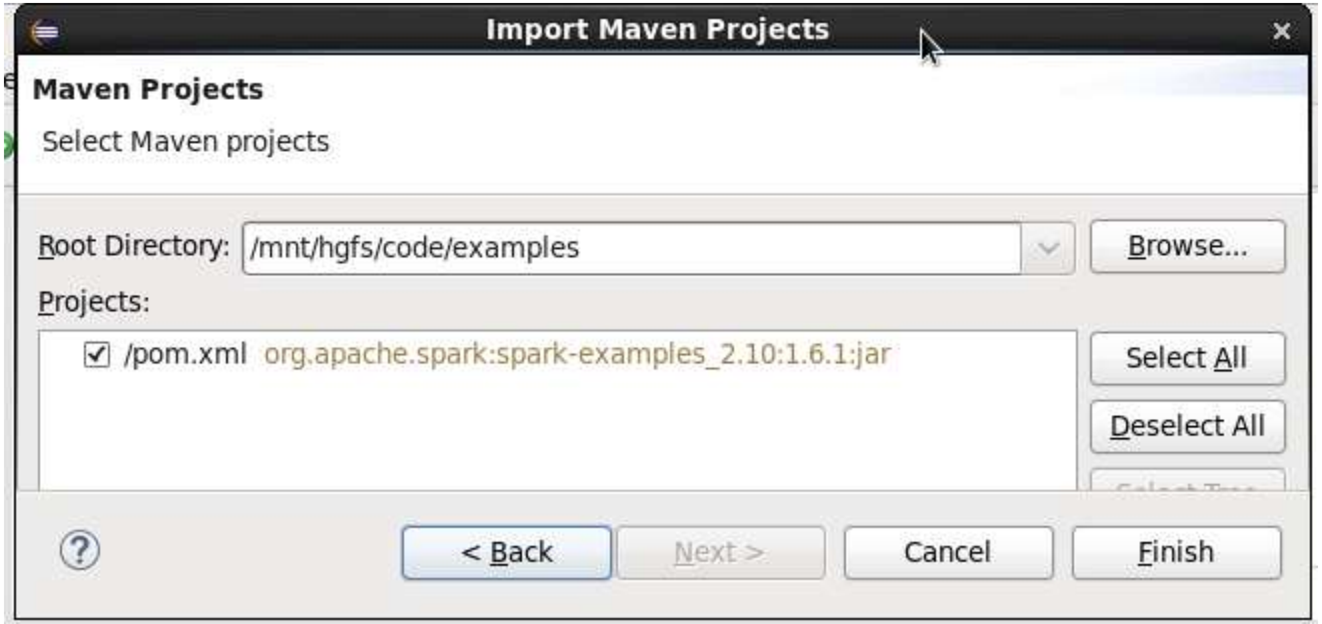


Name	Size	Modified
src		17:06
pom.xml	14.5 KB	02/26/2016

- By seeing `pom.xml` Eclipse recognizes that this is a Maven project. Clink OK on the bottom of the Wizard

Once more

- On the bottom of the next Wizard click Finish again



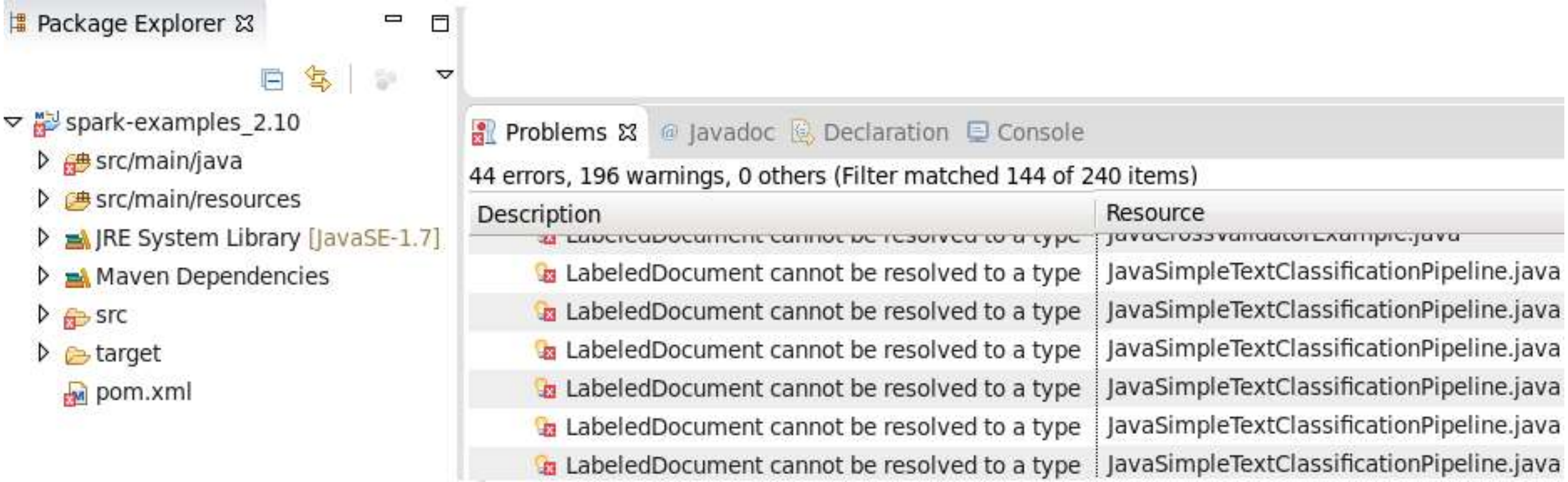
Root Directory: /mnt/hgfs/code/examples

Projects:

☑ /pom.xml org.apache.spark:spark-examples_2.10:1.6.1:jar

spark-examples_2.10 project

- Eclipse read project name from the pom.xml file and came back with a large number of errors.



44 errors, 196 warnings, 0 others (Filter matched 144 of 240 items)

Description	Resource
LabeledDocument cannot be resolved to a type	JavaCrossValidatorExample.java
LabeledDocument cannot be resolved to a type	JavaSimpleTextClassificationPipeline.java
LabeledDocument cannot be resolved to a type	JavaSimpleTextClassificationPipeline.java
LabeledDocument cannot be resolved to a type	JavaSimpleTextClassificationPipeline.java
LabeledDocument cannot be resolved to a type	JavaSimpleTextClassificationPipeline.java
LabeledDocument cannot be resolved to a type	JavaSimpleTextClassificationPipeline.java
LabeledDocument cannot be resolved to a type	JavaSimpleTextClassificationPipeline.java

- Let us try to get rid of them.
- Perhaps we need to add Spark classes to project's Build Path
- If we click on Maven Dependencies we will see a ton of libraries. Maven perhaps took care of the Build Path.
- Practically all errors are coming from three unresolved references: Document, LabeledDocument and StreamingExamples
- If we search for those classes, they will appear among Scala classes.
- They also suggest that our Scala setup on Eclipse is not working

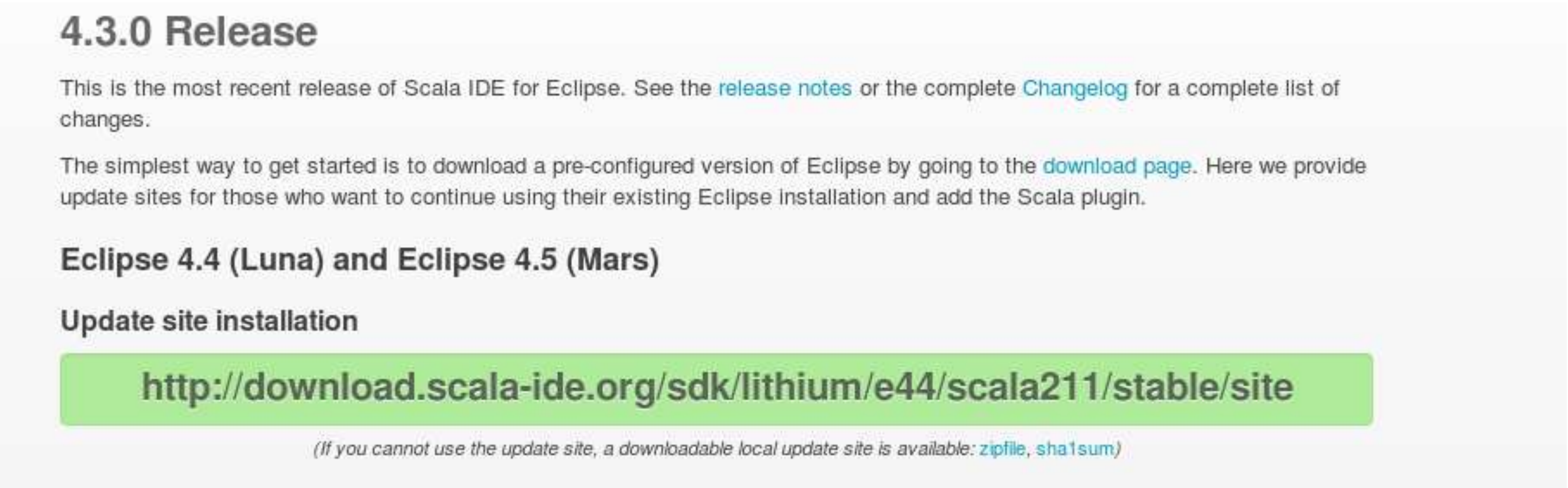
Scala IDE

- On slide 8 (Setting Eclipse Up) item #3 Install one of the Scala IDEs had The Scala IDE (based on Eclipse). If we follow that link we will get to page:

http://scala-ide.org/download/current.html

- The page describes ScalaIDE, and provides you with a nice video explaining how to proceed.
- The page and the video tell us that we can install ScalaIDE as a plugin for Eclipse. The link to that plug is provided:

http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site



4.3.0 Release

This is the most recent release of Scala IDE for Eclipse. See the release notes or the complete Changelog for a complete list of changes.

The simplest way to get started is to download a pre-configured version of Eclipse by going to the download page. Here we provide update sites for those who want to continue using their existing Eclipse installation and add the Scala plugin.

Eclipse 4.4 (Luna) and Eclipse 4.5 (Mars)

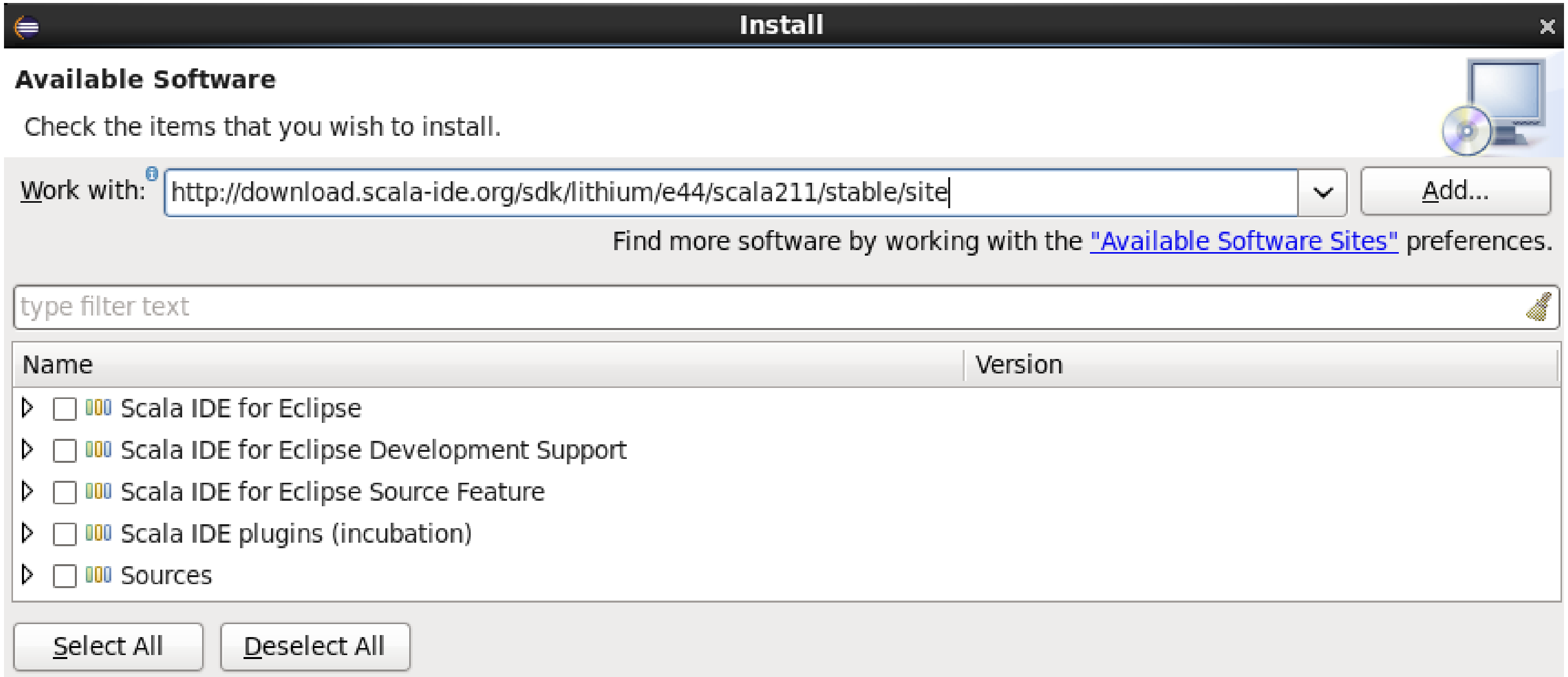
Update site installation

http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site

(If you cannot use the update site, a downloadable local update site is available: zipfile, sha1sum)

Install ScalaIDE as Eclipse plugin

- We go to Eclipse > Help > Install New Software and in the top field "Work with" we enter the URL from the previous page. Eclipse discovers several packages and lists them.



Install

Available Software

Check the items that you wish to install.

Work with: http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site

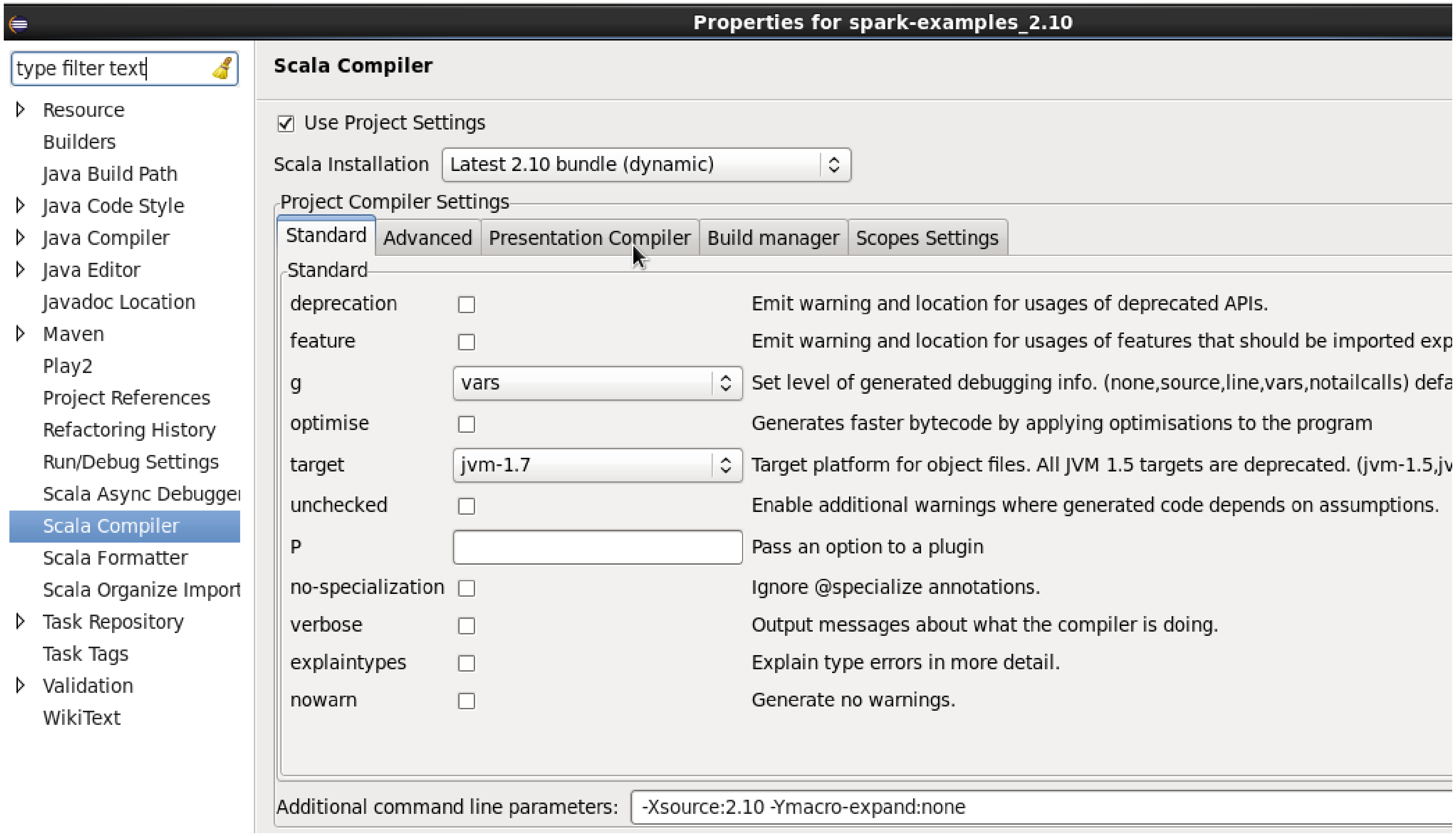
Find more software by working with the "Available Software Sites" preferences.

Name	Version
▷ □ Scala IDE for Eclipse	
▷ □ Scala IDE for Eclipse Development Support	
▷ □ Scala IDE for Eclipse Source Feature	
▷ □ Scala IDE plugins (incubation)	
▷ □ Sources	

- We `Select All` and hit `Next`, and Next and then Check `"I Accept Apache license …"` and `Finish` on the following screen. You also say `"Yes"` to any question Eclipse asks.

Scala Nature

- Select you project and at the bottom of the menu, select Scala. This opens a new menu. Select “Add Scala Nature”.
- When you now go to Project > Properties you will see Scala Compiler in the left bar.
- Select Scala Compiler and change Scala Installation to: `Latest 2.10 bundle(dynamic)`



Scala Compiler

☑ Use Project Settings

Scala Installation: Latest 2.10 bundle (dynamic)

Project Compiler Settings

Standard

deprecation	□	Emit warning and location for usages of deprecated APIs.
feature	□	Emit warning and location for usages of features that should be imported exp
g	vars	Set level of generated debugging info. (none,source,line,vars,notailcalls) defa
optimise	□	Generates faster bytecode by applying optimisations to the program
target	jvm-1.7	Target platform for object files. All JVM 1.5 targets are deprecated. (jvm-1.5,jv
unchecked	□	Enable additional warnings where generated code depends on assumptions.
P		Pass an option to a plugin
no-specialization	□	Ignore @specialize annotations.
verbose	□	Output messages about what the compiler is doing.
explaintypes	□	Explain type errors in more detail.
nowarn	□	Generate no warnings.

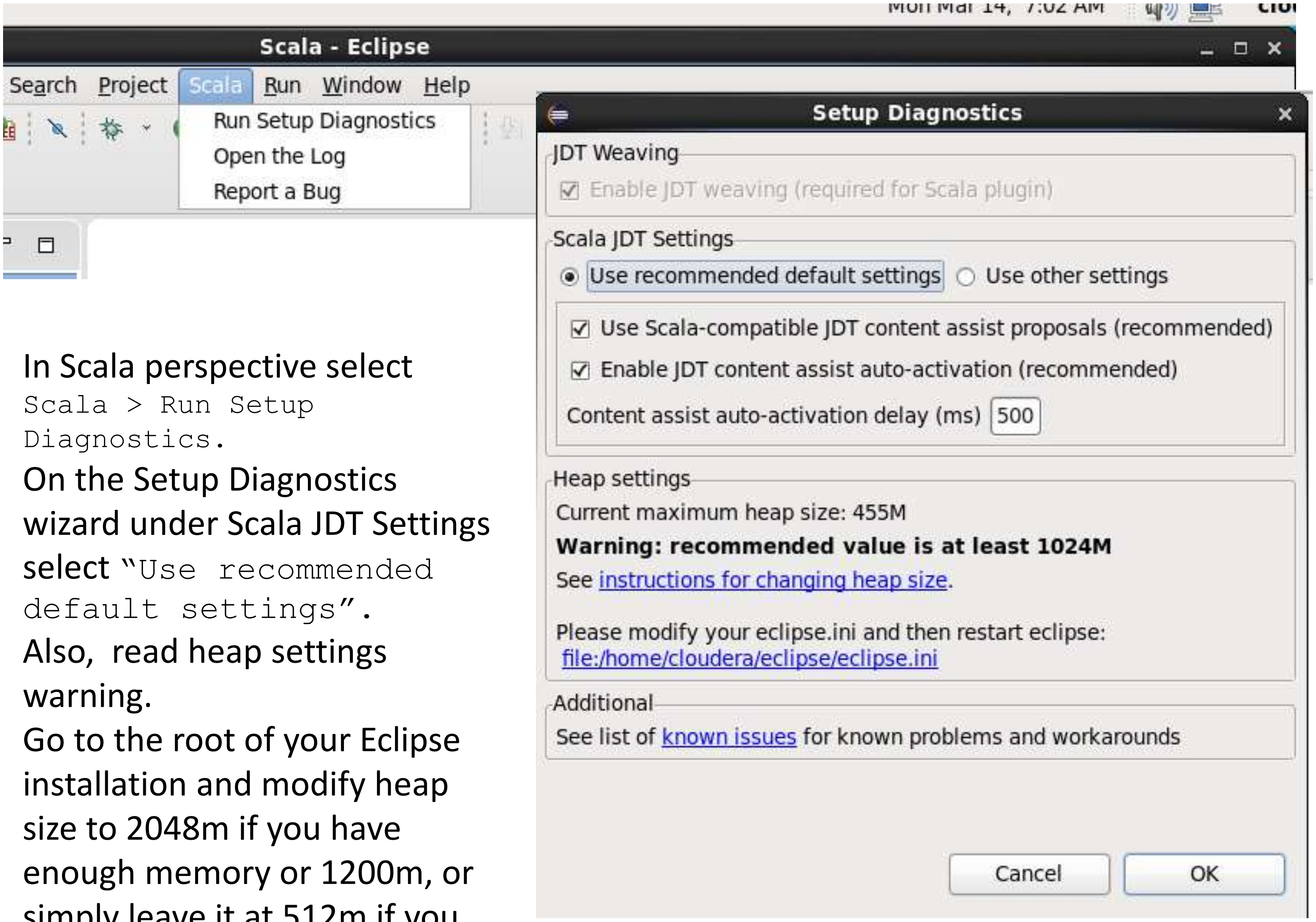
Additional command line parameters: -Xsource:2.10 -Ymacro-expand:none

Change Eclipse Heap Size

- One of the recommendations the installation process had was to change the Heap size from 500 MB to more than 1024 MB.
- In folder `/home/cloudera/eclipse` locate file `eclipse.ini` and change the value of parameter `-Xmx512 to -Xmx1212m.` Change to a higher value if your VM has plenty of memory.
- When you restart Eclipse, it should not complain about the heap size, any more.

- One new thing appeared in Eclipse. We now have Scala Perspective. When you want to work with Scala code, you should switch to that perspective.

In Scala Perspective, run Setup Diagnostics



Setup Diagnostics

JDT Weaving

☑ Enable JDT weaving (required for Scala plugin)

Scala JDT Settings

● Use recommended default settings ○ Use other settings

☑ Use Scala-compatible JDT content assist proposals (recommended)

☑ Enable JDT content assist auto-activation (recommended)

Content assist auto-activation delay (ms) 500

Heap settings

Current maximum heap size: 455M

Warning: recommended value is at least 1024M

See instructions for changing heap size.

Please modify your eclipse.ini and then restart eclipse:
file:/home/cloudera/eclipse/eclipse.ini

Additional

See list of known issues for known problems and workarounds

- In Scala perspective select `Scala > Run Setup Diagnostics`.
- On the Setup Diagnostics wizard under Scala JDT Settings select "`Use recommended default settings`".
- Also, read heap settings warning.
- Go to the root of your Eclipse installation and modify heap size to 2048m if you have enough memory or 1200m, or simply leave it at 512m if you have none.

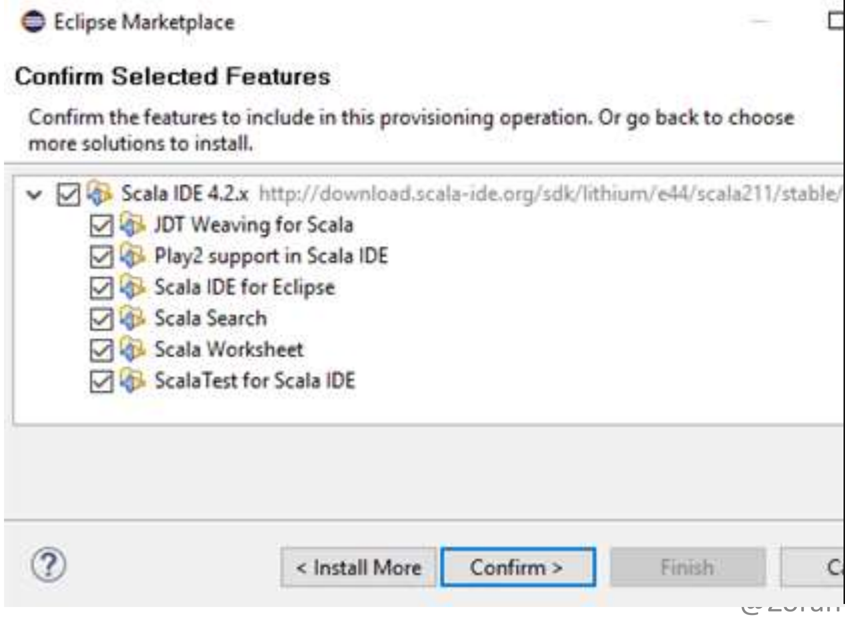
Eclipse Marketplace

- Select `Help>Eclipse Marketplace`
- In the top field enter:

  `Scala IDE`
- Select `Scala IDE 4.2`

and hit `Install`, and `Confirm` to include all options.

- Accept the license and hit `Finish.`
- Say Yes and OK whatever you are asked

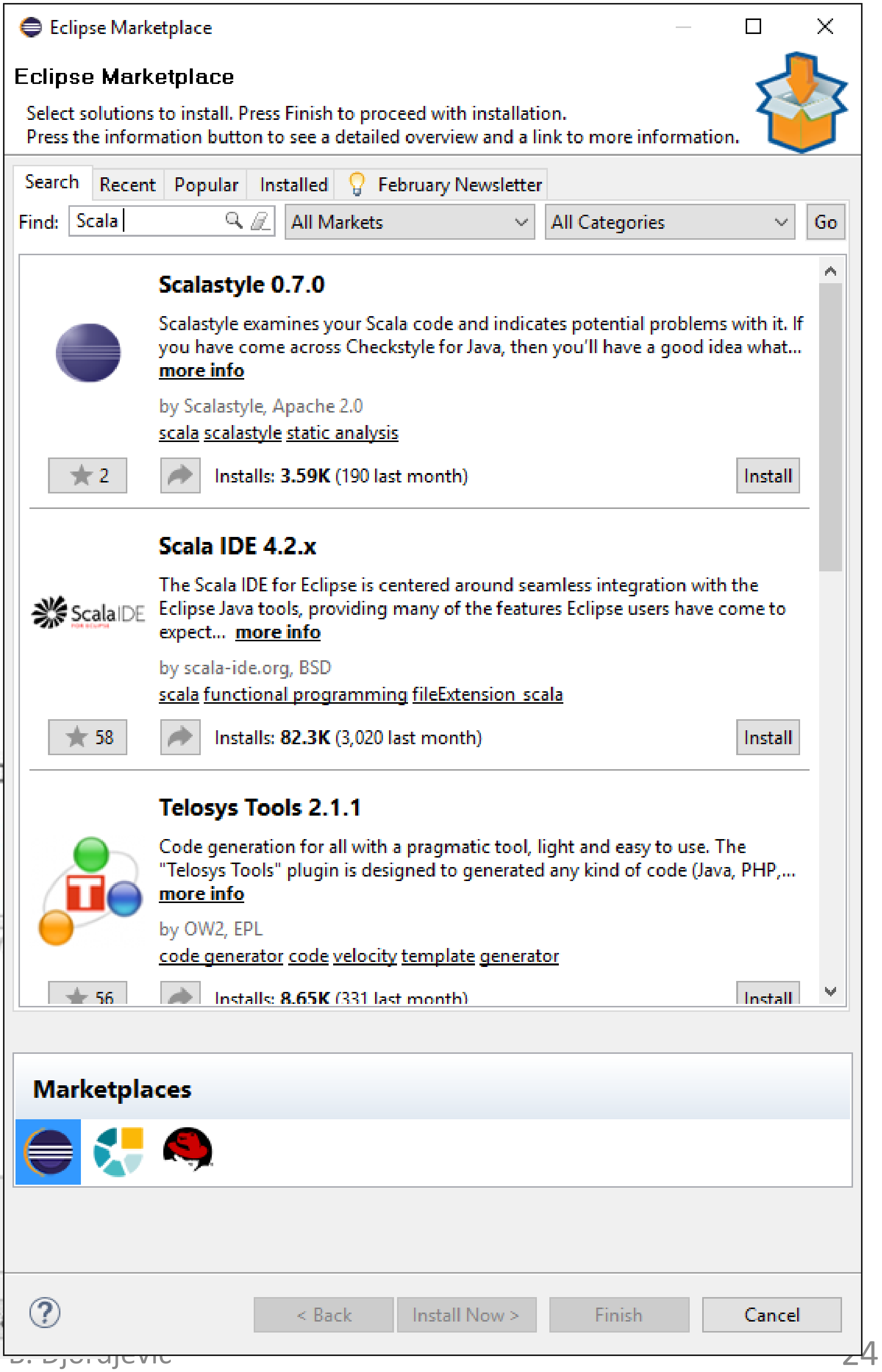


Eclipse Marketplace

Confirm Selected Features

Confirm the features to include in this provisioning operation. Or go back to choose more solutions to install.

- ☑ Scala IDE 4.2.x http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/
  - ☑ JDT Weaving for Scala
  - ☑ Play2 support in Scala IDE
  - ☑ Scala IDE for Eclipse
  - ☑ Scala Search
  - ☑ Scala Worksheet
  - ☑ ScalaTest for Scala IDE



Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

Find: Scala | All Markets | All Categories | Go

Scalastyle 0.7.0

Scalastyle examines your Scala code and indicates potential problems with it. If you have come across Checkstyle for Java, then you'll have a good idea what... more info

by Scalastyle, Apache 2.0

scala scalastyle static analysis

★ 2 Installs: **3.59K** (190 last month) Install

Scala IDE 4.2.x

The Scala IDE for Eclipse is centered around seamless integration with the Eclipse Java tools, providing many of the features Eclipse users have come to expect... more info

by scala-ide.org, BSD

scala functional programming fileExtension_scala

★ 58 Installs: **82.3K** (3,020 last month) Install

Telosys Tools 2.1.1

Code generation for all with a pragmatic tool, light and easy to use. The "Telosys Tools" plugin is designed to generated any kind of code (Java, PHP,... more info

by OW2, EPL

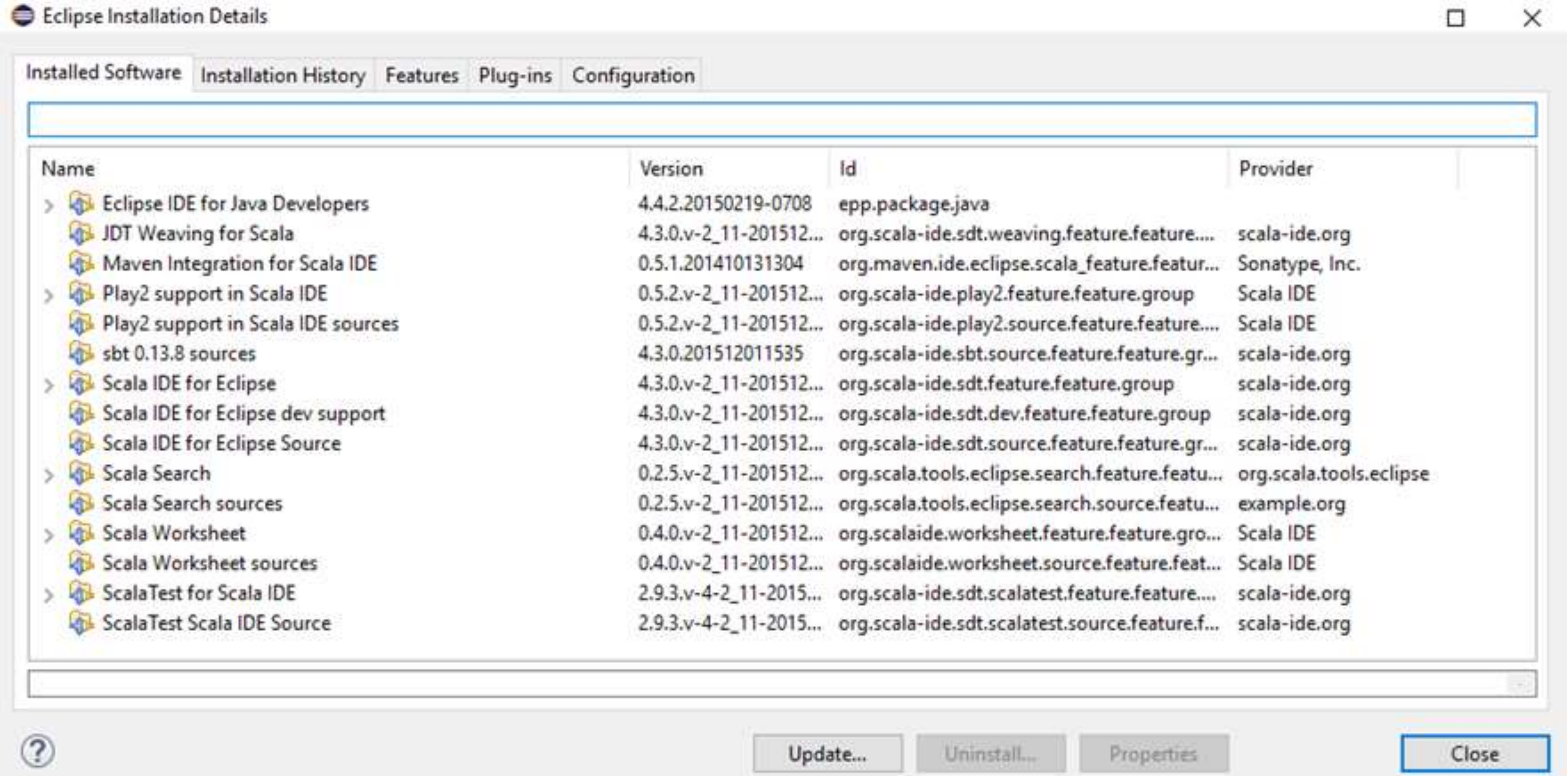
code generator code velocity template generator

★ 56 Installs: **8.65K** (331 last month) Install

Marketplaces

Installed Software

- If you select Help > Installed Details you will see which packages got installed:

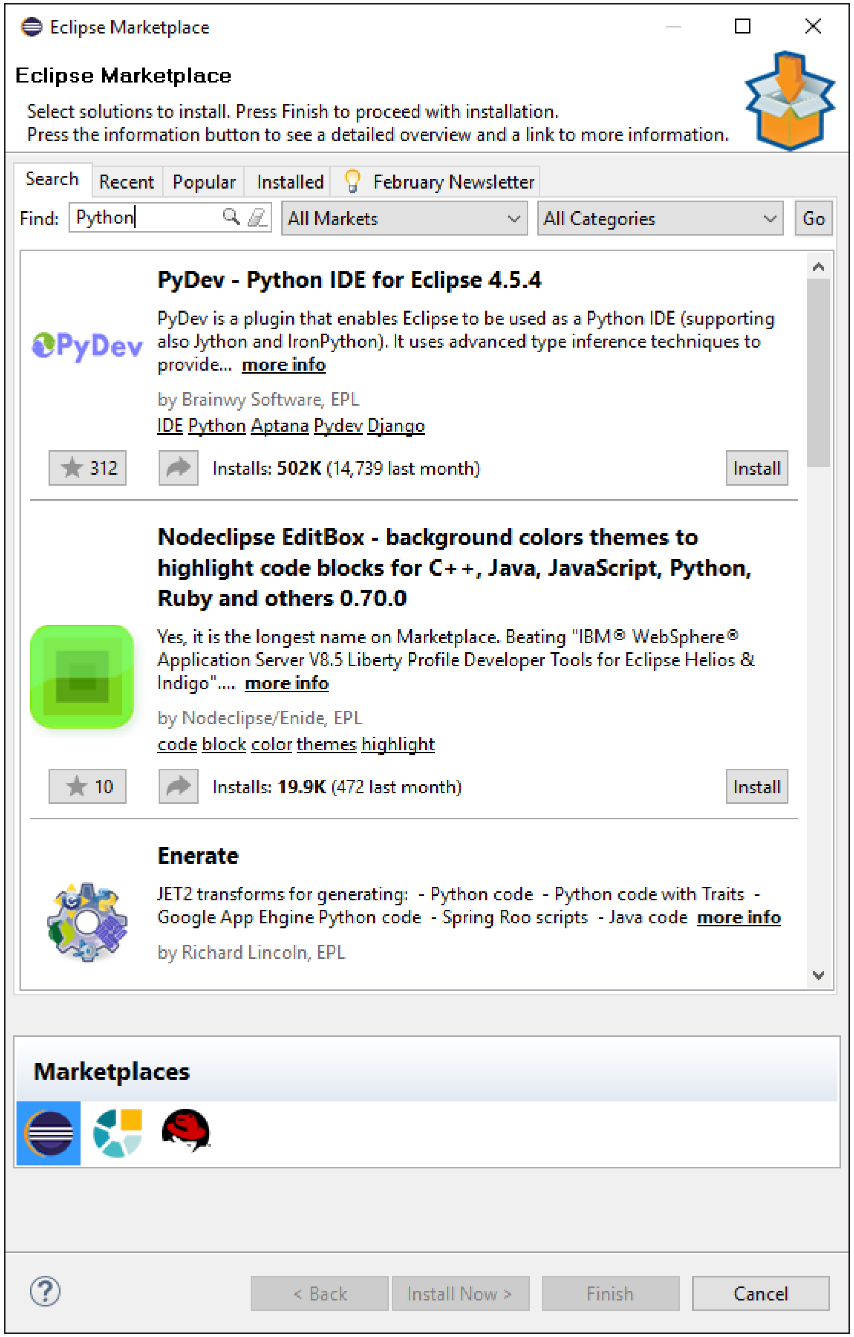


Name	Version	Id	Provider
› Eclipse IDE for Java Developers	4.4.2.20150219-0708	epp.package.java	
JDT Weaving for Scala	4.3.0.v-2_11-201512...	org.scala-ide.sdt.weaving.feature.feature....	scala-ide.org
Maven Integration for Scala IDE	0.5.1.201410131304	org.maven.ide.eclipse.scala_feature.featur...	Sonatype, Inc.
› Play2 support in Scala IDE	0.5.2.v-2_11-201512...	org.scala-ide.play2.feature.feature.group	Scala IDE
Play2 support in Scala IDE sources	0.5.2.v-2_11-201512...	org.scala-ide.play2.source.feature.feature....	Scala IDE
sbt 0.13.8 sources	4.3.0.201512011535	org.scala-ide.sbt.source.feature.feature.gr...	scala-ide.org
› Scala IDE for Eclipse	4.3.0.v-2_11-201512...	org.scala-ide.sdt.feature.feature.group	scala-ide.org
Scala IDE for Eclipse dev support	4.3.0.v-2_11-201512...	org.scala-ide.sdt.dev.feature.feature.group	scala-ide.org
Scala IDE for Eclipse Source	4.3.0.v-2_11-201512...	org.scala-ide.sdt.source.feature.feature.gr...	scala-ide.org
› Scala Search	0.2.5.v-2_11-201512...	org.scala.tools.eclipse.search.feature.featu...	org.scala.tools.eclipse
Scala Search sources	0.2.5.v-2_11-201512...	org.scala.tools.eclipse.search.source.featu...	example.org
› Scala Worksheet	0.4.0.v-2_11-201512...	org.scalaide.worksheet.feature.feature.gro...	Scala IDE
Scala Worksheet sources	0.4.0.v-2_11-201512...	org.scalaide.worksheet.source.feature.feat...	Scala IDE
› ScalaTest for Scala IDE	2.9.3.v-4-2_11-2015...	org.scala-ide.sdt.scalatest.feature.feature....	scala-ide.org
ScalaTest Scala IDE Source	2.9.3.v-4-2_11-2015...	org.scala-ide.sdt.scalatest.source.feature.f...	scala-ide.org

- JDT (Java Development Tools) Weaving for Scala are enabling Apect for J functionality.

Install Python IDE

- We can similarly install Python IDE. Select `Help > Eclipse Marketplace`, enter `Python` and then select `Install`.
- The result of this installation is that Eclipse will acquire a `PyDev` perspective.



Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

Find: Python All Markets All Categories Go

PyDev - Python IDE for Eclipse 4.5.4

PyDev is a plugin that enables Eclipse to be used as a Python IDE (supporting also Jython and IronPython). It uses advanced type inference techniques to provide... **more info**

by Brainwy Software, EPL

IDE Python Aptana Pydev Django

312 Installs: **502K** (14,739 last month) Install

Nodeclipse EditBox - background colors themes to highlight code blocks for C++, Java, JavaScript, Python, Ruby and others 0.70.0

Yes, it is the longest name on Marketplace. Beating "IBM® WebSphere® Application Server V8.5 Liberty Profile Developer Tools for Eclipse Helios & Indigo".... **more info**

by Nodeclipse/Enide, EPL

code block color themes highlight

10 Installs: **19.9K** (472 last month) Install

Enerate

JET2 transforms for generating: - Python code - Python code with Traits - Google App Ehgine Python code - Spring Roo scripts - Java code **more info**

by Richard Lincoln, EPL

Marketplaces

Delete your Maven project and Import it again

- After all these changes to Eclipse, delete your Maven project and import it again.
- Hopefully all errors will go away.

- On the following pages we present an alternative way of installing ScalaIDE to Eclipse, using Eclipse Marketplace

Installing Spark Packages

- Normally when you run Spark cluster one machine is the spark-master and other machines are spark-workers. You need code for those roles.
- There are five Spark packages:
  - `spark-core`: delivers core functionality of Spark
  - `spark-worker`: init scripts for spark-worker
  - `spark-master`: init scripts for spark-master
  - `spark-python`: Python client for Spark
  - `spark-history-server`
- To install all those packages on CentOS system, you would typically do:

```
$ sudo yum install spark-core spark-master spark-worker spark-
history-server spark-python
```

- Cloudera VM appears to have spark-core and spark-history-server installed, already. Whether it does or not you can still safely run the above command.
- After the above installation, in directory `/etc/init.d` you will see:

```
$ ls spark*
spark-history-server  spark-master  spark-worker
```

Configure and Run Spark (Standalone)

- Before you can run Spark in standalone mode, you must do the following on every host in the cluster:
- Edit `/etc/spark/conf/spark-env.sh` to point to the host where the spark-master runs:

```
### Change the following to specify a real cluster's Master host
###
export STANDALONE_SPARK_MASTER_HOST=`hostname` # Note Linux command `…`
```

You can change other elements of the default configuration by modifying `/etc/spark/conf/spark-env.sh`. You can change:

`SPARK_MASTER_PORT / SPARK_MASTER_WEBUI_PORT`, to use non-default ports

`SPARK_WORKER_CORES`, to set the number of cores to use on this machine

`SPARK_WORKER_MEMORY`, to set how much memory to use (for example 1000MB, 2GB)

`SPARK_WORKER_PORT / SPARK_WORKER_WEBUI_PORT`

`SPARK_WORKER_INSTANCE`, to set the number of worker processes per node

`SPARK_WORKER_DIR`, to set the working directory of worker processes

- We will try running with default values.

Configuring the Spark History Server

- Before you can run the Spark history server, you must create the `/user/spark/applicationHistory/` directory in HDFS and set ownership and permissions as follows:

```
$ sudo -u hdfs hadoop fs -mkdir /user/spark
$ sudo -u hdfs hadoop fs -mkdir /user/spark/applicationHistory
$ sudo -u hdfs hadoop fs -chown -R spark:spark /user/spark
$ sudo -u hdfs hadoop fs -chmod 1777 /user/spark/applicationHistory
```

- On Cloudera VM, that directory appears to be present. We just need to run the last command to set permissions properly.
- To start spark-history-server go to `/etc/init.d` and type:

```
$ sudo service spark-history-server start
```

Spark Client Systems

- If you are running Spark client programs (Spark jobs) on machine(s) different from Spark Master, perform the following on that client machine. We try to run everything on one VM.

1. Create `/etc/spark/conf/spark-defaults.conf` on the Spark client:

```
cp /etc/spark/conf/spark-defaults.conf.template /etc/spark/conf/spark-defaults.conf
```

2. Add the following to `/etc/spark/conf/spark-defaults.conf:`

```
spark.eventLog.dir=/user/spark/applicationHistory
spark.eventLog.enabled=true
```

- This causes Spark applications running on this client to write their history to the directory that the history server reads.
- In addition, if you want the YARN ResourceManager to link directly to the Spark History Server, you can set the `spark.yarn.historyServer.address` property in `/etc/spark/conf/spark-defaults.conf:`

```
spark.yarn.historyServer.address=http://HISTORY_HOST:18088
```

Starting and Stopping Standalone Cluster

To start Spark Standalone clusters:

1. On one host in the cluster, start the Spark Master by going to `/etc/init.d` directory and typing

```
$ sudo service spark-master start
```

- You can access the Spark Master UI at spark_master:18080.

2. On all the other hosts, start the workers:

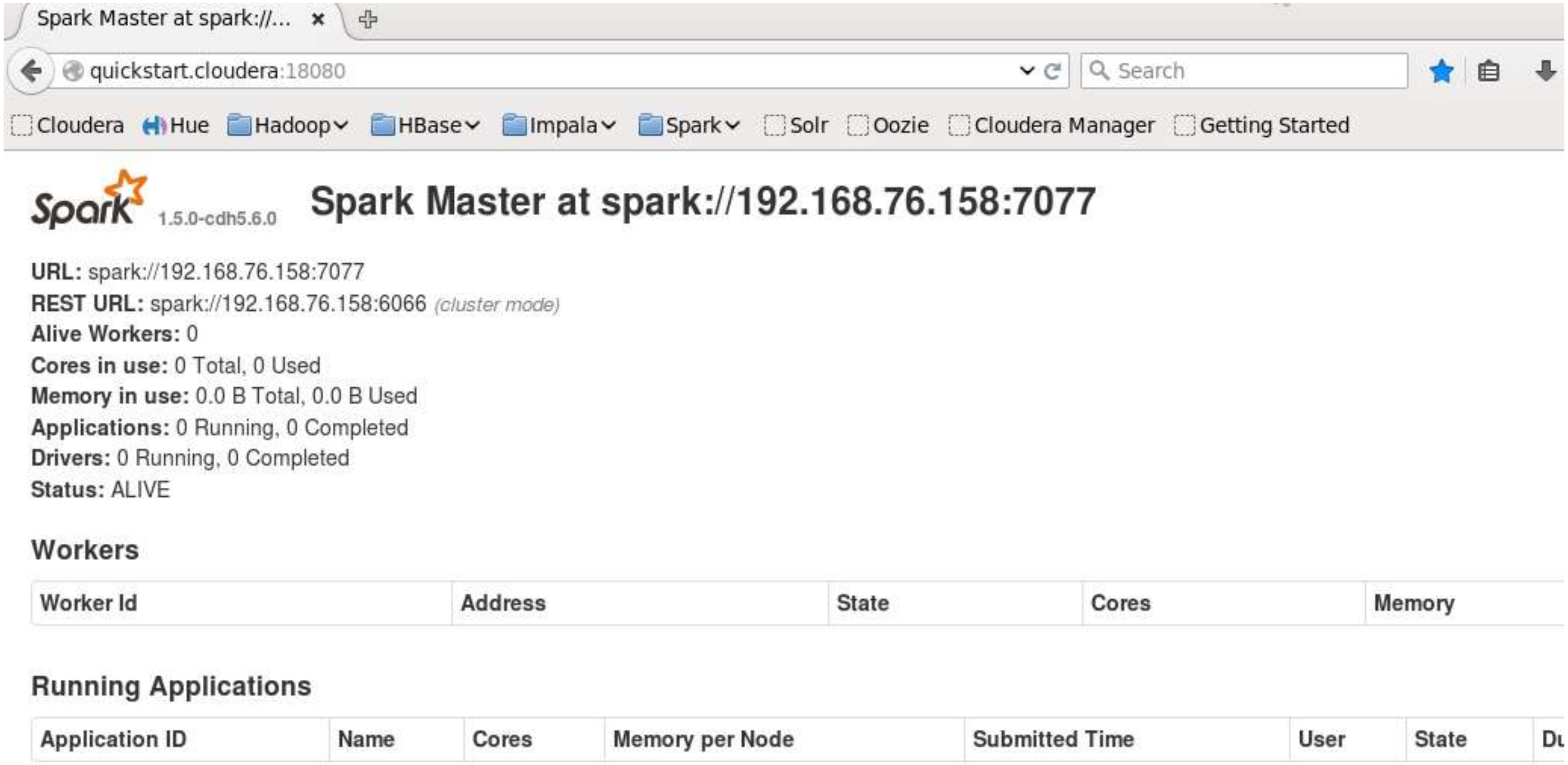
```
$ sudo service spark-worker start
```

- To stop Spark, use the following commands on the appropriate hosts:

```
$ sudo service spark-worker stop
$ sudo service spark-master stop
```

- Service logs are stored in `/var/log/spark.`

Spark Master on Port 7077



Spark Master at spark://192.168.76.158:7077

URL: spark://192.168.76.158:7077
REST URL: spark://192.168.76.158:6066 *(cluster mode)*
Alive Workers: 0
Cores in use: 0 Total, 0 Used
Memory in use: 0.0 B Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

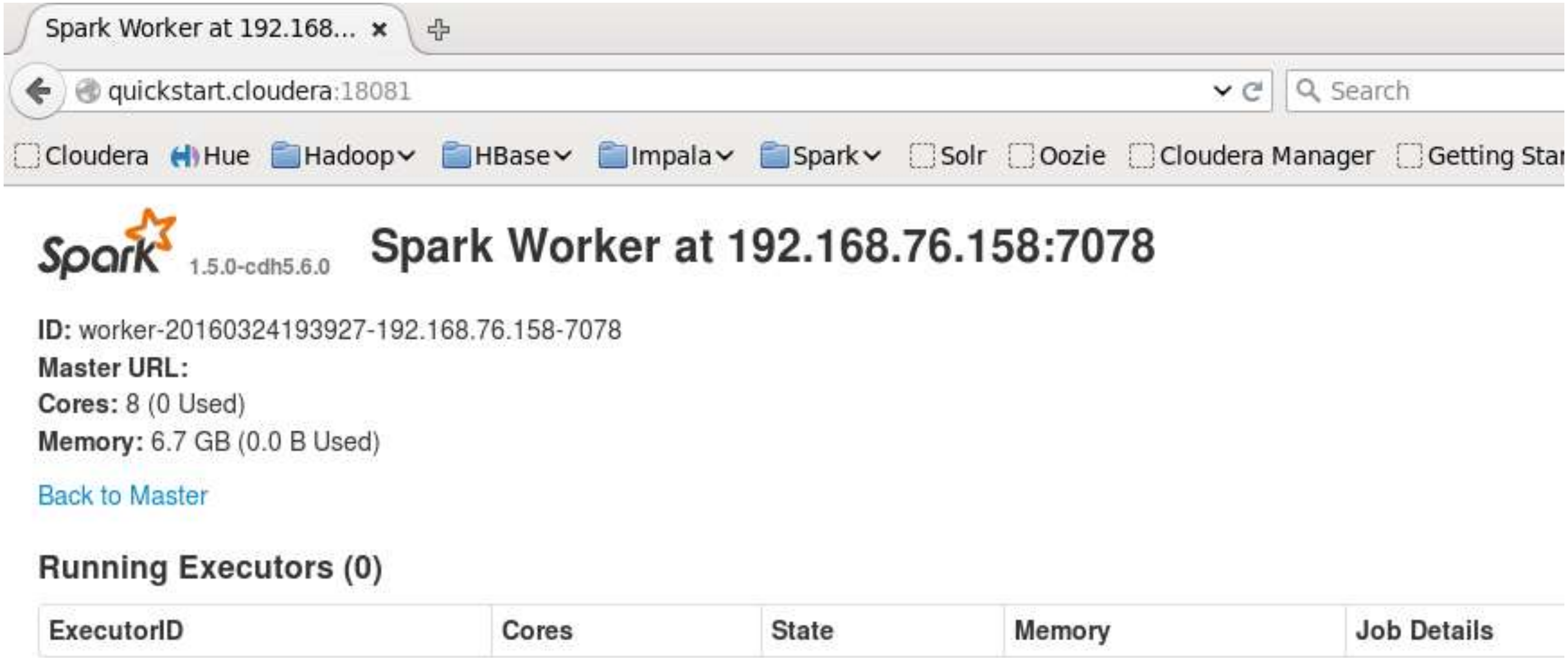
Workers

Worker Id	Address	State	Cores	Memory

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Du

Spark Worker at port 7078



Spark Worker at 192.168.76.158:7078

ID: worker-20160324193927-192.168.76.158-7078
Master URL:
Cores: 8 (0 Used)
Memory: 6.7 GB (0.0 B Used)

Back to Master

Running Executors (0)

ExecutorID	Cores	State	Memory	Job Details

Run Examples

- No that we went through all these troubles we are confident that our Java code compiles and (hopefully) works.
- To run our `JavaNetworkWordCount` java class we can use Eclipse to create Java configuration, export it as a jar file and then use `spark-submit` command to run that jar.
- For now, we can also use `/usr/lib/spark/bin/run-example` command. It is a just simpler. We should type, on the single line:

```
$  /usr/lib/spark/bin/run-example
org.apache.spark.streaming.examples.JavaNetworkWordCount local[2]
localhost 9999
```

- `local[2]` means that we will run 2 threads on the local machine and fetch whatever is pushed by TCPIP protocol to port `9999`.
- To push data (words, text) to port 9999 we will use Linux utility NetCat: `nc`

```
$ man nc
nc [-46DdhklnrStUuvzC] [-i interval] [-p source_port]
              [-s source_ip_address] [-T ToS]
             [-w timeout] [-X proxy_protocol] [-x proxy_address[:port]]
[hostname] [port[s]]

DESCRIPTION
The nc (or netcat) utility is used for just about anything under the sun involving TCP or UDP.
It can open TCP connections, send UDP packets, listen on arbitrary TCP and UDP ports, do port
scanning, and deal with both IPv4 and IPv6.  Unlike telnet(1), nc scripts nicely, and sepa-
rates error messages onto standard error instead of sending them to standard output, as
telnet(1) does with some.
```

Run `spark-submit`

- I exported my configuration as file `networkcount.jar` and transferred from Windows to Cloudera VM. In the directory where that file resides I type:

```
$ spark-submit --class
org.apache.spark.examples.streaming.JavaNetworkWordCount --master
local[1] networkcount.jar localhost 9999
```

- In another command prompt window I start NetCat and type a bunch of words

```
$ nc -lk 9999
hello
word
how are you
I am fine
hello again
We are so happy to see you
```

- You may try doing this as well:

```
$ nc -lk 9999 | cat all-bible
```

- When you kill spark-submit job, it will print a bunch of those words, in groups captured in 1 sec intervals.

Test Results

```
-------------------------------------------
Time: 1357008430000 ms
-------------------------------------------
(hello,1)
(word,1)
-------------------------------------------
Time: 1357008431000 ms
-------------------------------------------
(how,1)
(are,1)
(you,1)
-------------------------------------------
Time: 1357008432000 ms
-------------------------------------------
-------------------------------------------
Time: 1357008433000 ms
-------------------------------------------
(I, 1)
(am, 1)
```

Dependencies

- To write Spark Streaming programs you must add the following to SBT or Maven project:

```
groupId = org.apache.spark
artifactId = spark-streaming_2.10
version = 0.9.1
```

- For ingesting data from sources like Kafka and Flume that are not present in the Spark Streaming core API, you will have to add the corresponding artifact `spark-streaming-xyz_2.10` to the dependencies.
- For example, some of the common ones are as follows.

Source	**Artifact**
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

Stream Initialization

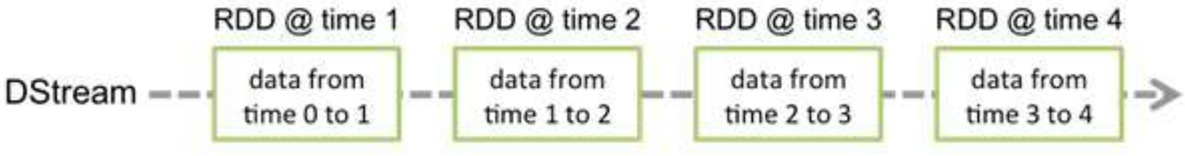
- To initialize a Spark Streaming program in Java, a `JavaStreamingContext` object has to be created, which is the main entry point of all Spark Streaming functionality. A `JavaStreamingContext` object can be created by using

```
new JavaStreamingContext(master, appName, batchInterval,
[sparkHome], [jars])
```

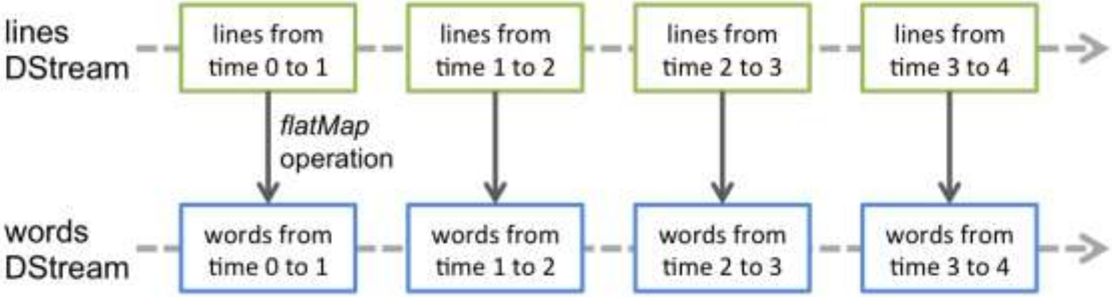
- The master parameter is a standard Spark cluster URL and can be “local” for local testing. The appName is a name of your program, which will be shown on your cluster’s web UI. The `batchInterval` is the size of the batches.
- Finally, the last two parameters are needed to deploy your code to a cluster if running in distributed mode.
- The underlying SparkContext can be accessed as `streamingContext.sparkContext`.
- The batch interval must be set based on the latency requirements of your application and available cluster resources.

DStream

- Discretized Stream or `DStream` represents a continuous stream of data, the input data stream received from source, or the processed data stream generated by transforming the input stream.
- Internally, it is represented by a continuous sequence of RDDs.
- Each RDD in a DStream contains data from a certain interval:



- Any operation applied on a DStream translates to operations on the underlying RDDs. For example, in the earlier example of converting a stream of lines to words, the flatmap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream.



- These underlying RDD transformations are computed by the Spark engine. The DStream operations hide most of these details and provides the developer with higher-level API for convenience. These operations are discussed in detail in later sections.

Input Sources

- We have seen `streamingContext.socketTextStream(...)` in the example with DStream from text data received over a TCP socket connection.
- Core Spark Streaming API provides methods for creating DStreams from files (and Akka actors as input sources). For files, the DStream can be created as

```
javaStreamingContext.fileStream(dataDirectory);
```

- Spark Streaming will monitor the directory `dataDirectory` for any Hadoop-compatible filesystem and process any files created in that directory. The files must have the same data format.
- The files must be created in the dataDirectory by atomically moving or renaming them into the data directory. Once moved the files must not be changed.

- Additional functionality for creating DStreams from sources such as Kafka, Flume, and Twitter can be imported by adding proper dependencies.
- For example, for Kafka, after adding the artifact `spark-streaming-kafka_2.10` to the project dependencies, we can create a DStream from Kafka as

```
import org.apache.spark.streaming.kafka.*
KafkaUtils.createStream(javaStreamingContext, kafkaParams, ...);
```

Operations

- There are two kinds of DStream operations - *transformations* and *output operations*.
- Similar to RDD transformations, DStream transformations operate on one or more DStreams to create new DStreams with transformed data.
- After applying a sequence of transformations to the input streams, output operations are called, which write data out to an external data sink, such as a filesystem or a database.
- DStreams support many of the transformations available on normal Spark RDD's

Transformations

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(*otherStream*, [*numTasks*])	When called on DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

`UpdateStateByKey` Operation

- The updateStateByKey operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, you will have to do two steps.
- Define the state - The state can be of arbitrary data type.
- Define the state update function - Specify with a function how to update the state using the previous state and the new values from input stream.
- For example we want to maintain a running count of each word seen in a text data stream. Here, the running count is the state and it is an integer. We define the update function as

```
Function2<List<Integer>, Optional<Integer>, Optional<Integer>> updateFunction =
  new Function2<List<Integer>, Optional<Integer>, Optional<Integer>>() {
    @Override public Optional<Integer> call(List<Integer> values,
Optional<Integer> state) {
      Integer newSum = ...  // add the new values with the previous running count
to get the new count
      return Optional.of(newSum)
    }
  }
```

- This is applied on a DStream containing words (say, the pairs DStream containing (word, 1) pairs in the example JavaNetworkWordCount).

```
JavaPairDStream<String, Integer> runningCounts =
pairs.updateStateByKey(updateFunction);
```

- The update function will be called for each word, with new Values having a sequence of 1's (from the (word, 1) pairs) and the runningCount having the previous count.

Transform Operation

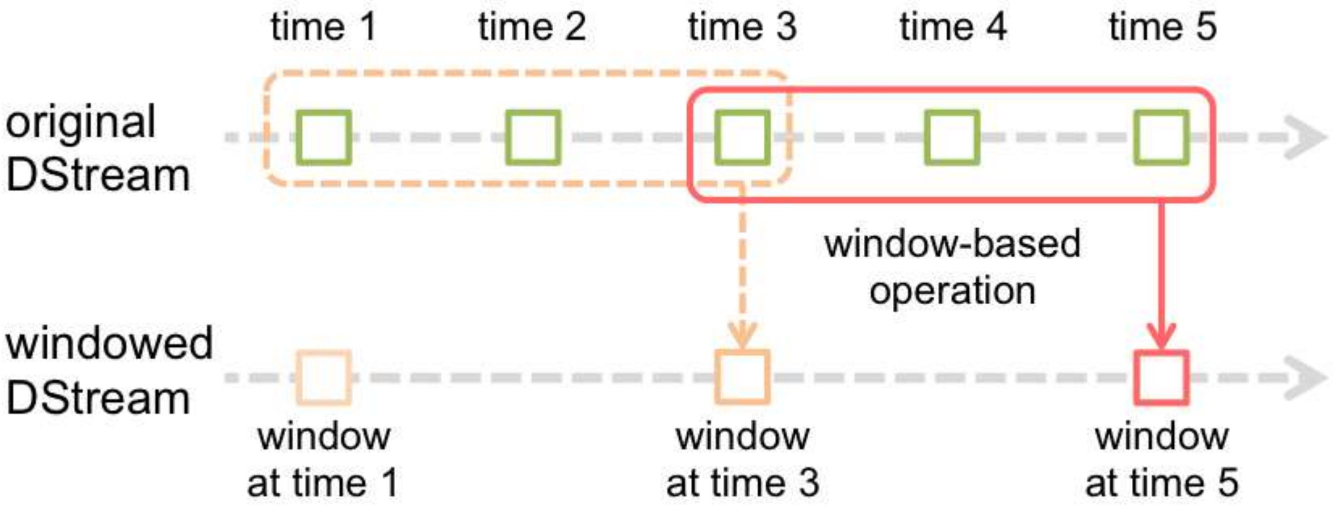
- The `transform` operation (along with its variations like `transformWith`) allows arbitrary RDD-to-RDD functions to be applied on a DStream. It can be used to apply any RDD operation that is not exposed in the DStream API. For example, the functionality of joining every batch in a data stream with another dataset is not directly exposed in the DStream API.
- For example, if you want to do real-time data cleaning by joining the input data stream with precomputed spam information and then filtering based on it.

```
// RDD containing spam information
JavaPairRDD<String, Double> spamInfoRDD = javaSparkContext.hadoopFile(...);

JavaPairDStream<String, Integer> cleanedDStream = inputDStream.transform(
  new Function<JavaPairRDD<String, Integer>, JavaPairRDD<String, Integer>>() {
    @Override public JavaPairRDD<String, Integer> call(JavaPairRDD<String,
Integer> rdd) throws Exception {
      rdd.join(spamInfoRDD).filter(...) // join data stream with spam information
to do data cleaning
      ...
    }
  });
```

- We can also use machine learning and graph computation algorithms in the transform method.

Window Operations

- Spark Streaming also provides window computations, which allow you to apply transformations over a sliding window of data.



- Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream. In this figure, the operation is applied over last 3 time units of data, and slides by 2 time units.
- Window-based operation needs to specify two parameters.
  - window length - The duration of the window (3 in the figure)
  - slide interval - The interval at which the window-based operation is performed (2 in the figure).
- These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).

An Example for `window` Operation

- Let’s say we want to extend the earlier example by generating word counts over last 30 seconds of data, every 10 seconds. To do this, we have to apply the `reduceByKey` operation on the pairs DStream of (word, 1) pairs over the last 30 seconds of data. This is done using the operation `reduceByKeyAndWindow`.

```
// Reduce function adding two integers, defined separately for
clarity
Function2<Integer, Integer, Integer> reduceFunc = new
Function2<Integer, Integer, Integer>() {
  @Override public Integer call(Integer i1, Integer i2) throws
Exception {
    return i1 + i2;
  }
};
```

- // Reduce last 30 seconds of data, every 10 seconds

```
JavaPairDStream<String, Integer> windowedWordCounts =
pair.reduceByKeyAndWindow(reduceFunc, new Duration(30000), new
Duration(10000));
```

- All of window operations take the said two parameters – `windowLength` and `slideInterval`.

Common `window` functions

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*,*slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative so that it can be computed correctly in parallel.
reduceByKeyAndWindow(*func*,*windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local machine, 8 for a cluster) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKeyAndWindow(*func*, *invFunc*,*windowLength*, *slideInterval*, [*numTasks*])	A more efficient version of the above `reduceByKeyAndWindow()` where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enter the sliding window, and "inverse reducing" the old data that leave the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable to only "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter *invFunc*. Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument.
countByValueAndWindow(*windowLength*,*slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument.

Output Operations

- When an output operator is called, it triggers the computation of a stream. Currently the following output operators are defined:

Output Operation	Meaning
print()	Prints first ten elements of every batch of data in a DStream on the driver.
foreachRDD(*func*)	The fundamental output operator. Applies a function, *func*, to each RDD generated from the stream. This function should have side effects, such as printing output, saving the RDD to external files, or writing it over the network to an external system.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as a `SequenceFile` of serialized objects. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as a text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as a Hadoop file. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.

Persistence

- Similar to RDDs, DStreams also allow developers to persist the stream's data in memory. `persist()` method applied on a DStream would automatically persist every RDD of that DStream in memory.
- Persisting a Dstream is useful if the data in the DStream will be computed multiple times (e.g., multiple operations on the same data). For window-based operations like `reduceByWindow` and `reduceByKeyAndWindow` and state-based operations like `updateStateByKey`, this is implicitly true.
- DStreams generated by window-based operations are automatically persisted in memory, without the developer calling persist().
- For input streams that receive data over the network (such as, Kafka, Flume, sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-tolerance.
- Unlike RDDs, the default persistence level of DStreams keeps the data serialized in memory.

RDD Checkpointing

- A stateful operation is one which operates over multiple batches of data. This includes all window-based operations and the `updateStateByKey` operation. Since stateful operations have a dependency on previous batches of data, they continuously accumulate metadata over time. To clear this metadata, streaming supports periodic checkpointing by saving intermediate data to HDFS.
- The checkpointing incurs the cost of saving to HDFS which may cause the corresponding batch to take longer to process. The checkpointing interval needs to be set carefully.
- At small batch sizes (1 second), checkpointing every batch may significantly reduce operation throughput. Conversely, checkpointing too slowly causes the lineage and task sizes to grow which may have detrimental effects.
- Typically, a checkpoint interval of 5 - 10 times of sliding interval of a DStream is good setting to try.
- To enable checkpointing, the developer has to provide the HDFS path to which RDD will be saved. This is done by using

```
ssc.checkpoint(hdfsPath)
```

- ssc is the StreamingContext or JavaStreamingContext
- The interval of checkpointing of a DStream can be set by using

```
dstream.checkpoint(checkpointInterval)
```

- For DStreams that must be checkpointed (that is, DStreams created by updateStateByKey and reduceByKeyAndWindow with inverse function), the checkpoint interval of the DStream is by default set to a multiple of the DStream’s sliding interval such that its at least 10 seconds.

Setting the Right Batch Size

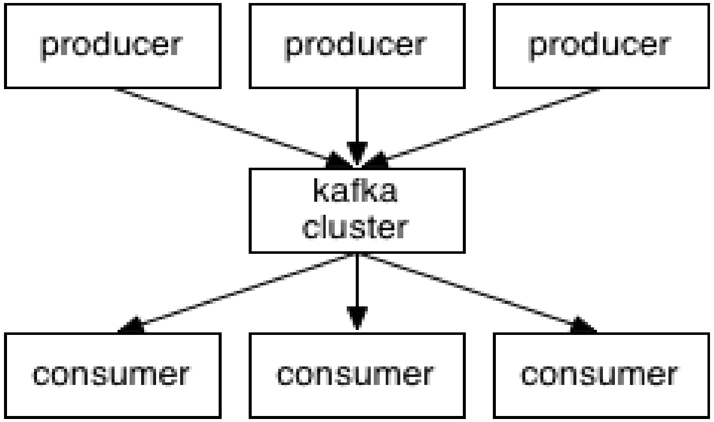
- For a Spark Streaming application running on a cluster to be stable, the processing of the data streams must keep up with the rate of ingestion of the data streams. Depending on the type of computation, the batch size used may have significant impact on the rate of ingestion that can be sustained by the Spark Streaming application on a fixed cluster resources.
- For example, let us consider the earlier JavaWordCountNetwork example. For a particular data rate, the system may be able to keep up with reporting word counts every 2 seconds (i.e., batch size of 2 seconds), but not every 500 milliseconds.
- A good approach to figure out the right batch size for your application is to test it with a conservative batch size (say, 5-10 seconds) and a low data rate. To verify whether the system is able to keep up with data rate, you can check the value of the end-to-end delay experienced by each processed batch (either look for “Total delay” in Spark driver log4j logs, or use the StreamingListener interface).
- If the delay is maintained to be comparable to the batch size, then system is stable. Otherwise, if the delay is continuously increasing, it means that the system is unable to keep up and it therefore unstable. Once you have an idea of a stable configuration, you can try increasing the data rate and/or reducing the batch size.
- The momentary increase in the delay due to temporary data rate increases maybe fine as long as the delay reduces back to a low value (i.e., less than batch size).

24/7 Operations

- By default, Spark does not forget any of the metadata (RDDs generated, stages processed, etc.).
- For Spark Streaming application to operate 24/7, it is necessary for Spark to do periodic cleanup of it metadata.
- This can be enabled by setting the configuration property `spark.cleaner.ttl` to the number of seconds you want any metadata to persist.
- For example, setting spark.cleaner.ttl to 600 would cause Spark periodically cleanup all metadata and persisted RDDs that are older than 10 minutes.
- This property needs to be set before the SparkContext is created.
- This value is closely tied with any window operation that is being used. Any window operation would require the input data to be persisted in memory for at least the duration of the window. Hence it is necessary to set the delay to at least the value of the largest window operation used in the Spark Streaming application. If this delay is set too low, the application will throw an exception saying so.

Call in Kafka

- Spark Streaming might have difficulties keeping up with incoming messages. We need some kind of buffer, to keep messages in place which Spark catches its breath. Apache Kafka provides that service.
- Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.
- Kafka maintains feeds of messages in categories called *topics.*
- Processes that publish messages to a Kafka topic are called *producers.*
- Processes that subscribe to topics and process the feed of published messages are called *consumers.*
- Kafka is run as a cluster comprised of one or more servers each of which is called a *broker*.
- For several Kafka machines (processes, servers) to run in coordination we need a "Cluster coordination software" called Zookeeper.

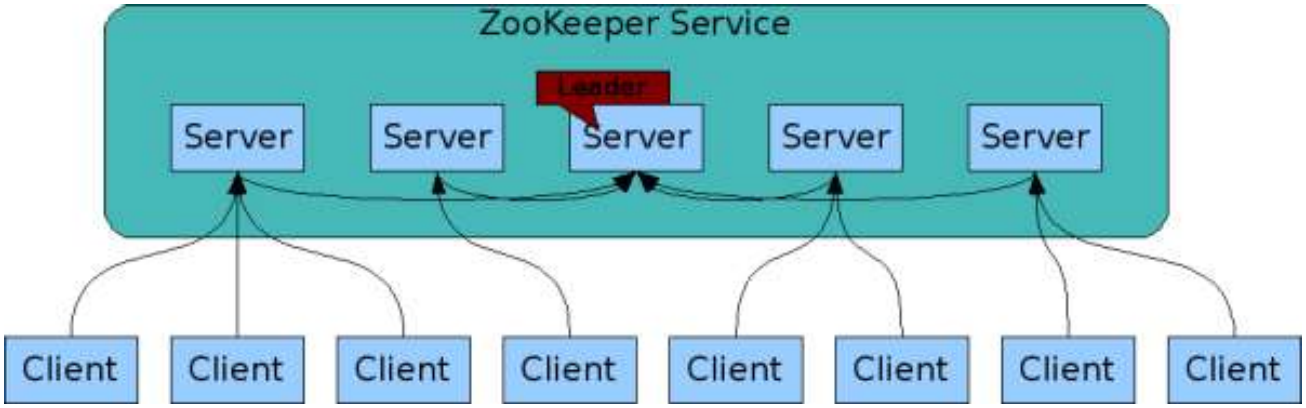


Zookeper

- ZooKeeper is a high-performance coordination service for distributed applications. It exposes common services - such as naming, configuration management, synchronization, and group services - in a simple interface so you don't have to write them from scratch.
- We can use Zookeeper off-the-shelf to implement consensus, group management, leader election, and presence protocols.
- Many frameworks, like Kafka, are built to use Zookeeper.
- ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchal namespace which is organized similarly to a standard file system. The name space consists of data registers - called znodes, in ZooKeeper parlance - and these are similar to files and directories. Unlike a typical file system, which is designed for storage, ZooKeeper data is kept in-memory, which means ZooKeeper can achieve high throughput and low latency numbers.
- The ZooKeeper implementation puts a premium on high performance, highly available, strictly ordered access. The performance aspects of ZooKeeper means it can be used in large, distributed systems. The reliability aspects keep it from being a single point of failure. The strict ordering means that sophisticated synchronization primitives can be implemented at the client.
- **ZooKeeper is replicated.** Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an ensemble.

Zookeeper

- The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.
- Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.
- **ZooKeeper is fast.** It is especially fast in "read-dominant" workloads. ZooKeeper applications run on thousands of machines, and it performs best where reads are more common than writes, at ratios of around 10:1.



Install Kafka, Start the Server

- Step 1. Download the 0.9.0.0 release and un-tar it.

```
> tar -xzf kafka_2.11-0.9.0.0.tgz
> cd kafka_2.11-0.9.0.0
```

- Step 2: Start the server
- Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with kafka to get a quick-and-dirty single-node ZooKeeper instance.

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
```

```
• [2016-03-22 15:01:37,495] INFO Reading configuration from:
  config/zookeeper.properties
  (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
• ...
```

- Now start the Kafka server:

```
> bin/kafka-server-start.sh config/server.properties
```

```
• [2016-0-22 15:01:47,028] INFO Verifying properties
  (kafka.utils.VerifiableProperties)
• [2016-03-22 15:01:47,051] INFO Property
  socket.send.buffer.bytes is overridden to 1048576
  (kafka.utils.VerifiableProperties)
```

- …

Create Topic, Start Producer, Consumer

- **Step 3**: Create a topic
- Let's create a topic named "test" with a single partition and only one replica:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

- We can now see that topic if we run the list topic command:

```
bin/kafka-topics.sh --list --zookeeper localhost:2181
test
```

- Alternatively, instead of manually creating topics you can also configure your brokers to auto-create topics when a non-existent topic is published to.
- **Step 4**: Send some messages
- Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default each line will be sent as a separate message.
- Run the producer and then type a few messages into the console to send to the server.

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
This is a message
This is another message
```

- **Step 5**: Start a consumer
- Kafka also has a command line consumer that will dump out messages to standard output.

```
➢ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning
This is a message
This is another message
```

- If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

Spark Streaming & Kafka Integration

- We want to configure Spark Streaming to receive data from Kafka.
- There are two approaches to this - the old approach using Receivers and Kafka’s high-level API, and a new approach (introduced in Spark 1.3) without using Receivers.
- They have different programming models, performance characteristics, and semantics guarantees.

Receiver Approach, Dependencies

- The Receiver is implemented using the Kafka high-level consumer API.
- The data received from Kafka through a Receiver is stored in Spark executors, and then jobs launched by Spark Streaming processes the data.
- Under the default configuration, this approach can lose data under failures. To ensure zero-data loss, you have to additionally enable Write Ahead Logs in Spark Streaming. These logs synchronously save all the received Kafka data into write ahead logs on a distributed file system (e.g HDFS), so that all the data can be recovered on failure.

- For Scala/Java applications using SBT/Maven project definitions, link your streaming application with the following artifact.

```
groupId = org.apache.spark
artifactId = spark-streaming-kafka_2.10
version = 1.6.1
```

- For Python applications, you will also have to add the above library and its dependencies when deploying the application.

Receiver Approach, Programming

- In the streaming application code, import `KafkaUtils` and create an input DStream :

```
import org.apache.spark.streaming.kafka.*;
 JavaPairReceiverInputDStream<String, String> kafkaStream =
     KafkaUtils.createStream(streamingContext,
     [ZK quorum], [consumer group id], [per-topic number of Kafka
partitions to consume]);
```

- You can also specify the key and value classes and their corresponding decoder classes using variations of `createStream`.
- Topic partitions in Kafka do not correlate to partitions of RDDs generated in Spark Streaming. Increasing the number of topic-specific partitions in the `KafkaUtils.createStream()` only increases the number of threads using which topics that are consumed within a single receiver. It does not increase the parallelism of Spark in processing the data.
- Multiple Kafka input DStreams can be created with different groups and topics for parallel receiving of data using multiple receivers.
- If you have enabled Write Ahead Logs with a replicated file system like HDFS, the received data is already being replicated in the log. Hence, the storage level in storage level for the input stream to `StorageLevel.MEMORY_AND_DISK_SER` (that is, use `KafkaUtils.createStream(..., StorageLevel.MEMORY_AND_DISK_SER))`.

Deploying

- As with any Spark applications, `spark-submit` is used to launch your application. However, the details are slightly different for Scala/Java applications and Python applications.
- For Scala and Java applications, if you are using SBT or Maven for project management, then package spark-streaming-kafka_2.10 and its dependencies into the application JAR.
- For Python applications which lack SBT/Maven project management, `spark-streaming-kafka_2.10` and its dependencies can be directly added to spark-submit using –packages.

```
 ./bin/spark-submit --packages org.apache.spark:spark-streaming-
kafka_2.10:1.6.1 ...
```

- Alternatively, you can also download the JAR of the Maven `artifact` `spark-streaming-kafka-assembly` from the Maven repository and add it to `spark-submit with --jars`

Direct Approach

- This new receiver-less "direct" approach periodically queries Kafka for the latest offsets in each topic+partition, and accordingly defines the offset ranges to process in each batch. When the jobs to process the data are launched, Kafka's simple consumer API is used to read the defined ranges of offsets from Kafka (similar to read files from a file system).

This approach has the following advantages over the receiver-based approach:

- **Simplified Parallelism**: No need to create multiple input Kafka streams and union them. With directStream, Spark Streaming will create as many RDD partitions as there are Kafka partitions to consume, which will all read data from Kafka in parallel. So there is a one-to-one mapping between Kafka and RDD partitions, which is easier to understand and tune.
- **Efficiency:** Achieving zero-data loss in the first approach required the data to be stored in a Write Ahead Log, which further replicated the data. This is actually inefficient as the data effectively gets replicated twice - once by Kafka, and a second time by the Write Ahead Log. This second approach eliminates the problem as there is no receiver, and hence no need for Write Ahead Logs. As long as you have sufficient Kafka retention, messages can be recovered from Kafka.
- **Exactly-once semantics:** The first approach uses Kafka's high level API to store consumed offsets in Zookeeper. This is traditionally the way to consume data from Kafka. While this approach (in combination with write ahead logs) can ensure zero data loss (i.e. at-least once semantics), there is a small chance some records may get consumed twice under some failures. This occurs because of inconsistencies between data reliably received by Spark Streaming and offsets tracked by Zookeeper. Hence, in this second approach, we use simple Kafka API that does not use Zookeeper. Offsets are tracked by Spark Streaming within its checkpoints. This eliminates inconsistencies between Spark Streaming and Zookeeper/Kafka, and so each record is received by Spark Streaming effectively exactly once despite failures.
- **One disadvantage** of this approach is that it does not update offsets in Zookeeper, hence Zookeeper-based Kafka monitoring tools will not show progress.

Programming

- This approach is supported only in Scala/Java application. Link your SBT/Maven project with the following artifact

```
groupId = org.apache.spark
artifactId = spark-streaming-kafka_2.10
version = 1.6.1
```

- In the streaming application code, import `KafkaUtils` and create an input `DStream` as:

```
 import org.apache.spark.streaming.kafka.*;

  JavaPairReceiverInputDStream<String, String> directKafkaStream =
      KafkaUtils.createDirectStream(streamingContext,
          [key class], [value class], [key decoder class], [value decoder 
class],
          [map of Kafka parameters], [set of topics to consume]);
```

- You can also pass a `messageHandler` to `createDirectStream` to access `MessageAndMetadata` that contains metadata about the current message and transform it to any desired type. See the API docs and the example.
- In the Kafka parameters, you must specify either `metadata.broker.list` or `bootstrap.servers`. By default, it will start consuming from the latest offset of each Kafka partition. If you set configuration `auto.offset.reset` in Kafka parameters to smallest, then it will start consuming from the smallest offset.
- You can also start consuming from any arbitrary offset using other variations of KafkaUtils.createDirectStream. Furthermore, if you want to access the Kafka offsets consumed in each batch, you can do the following

Code Snippet

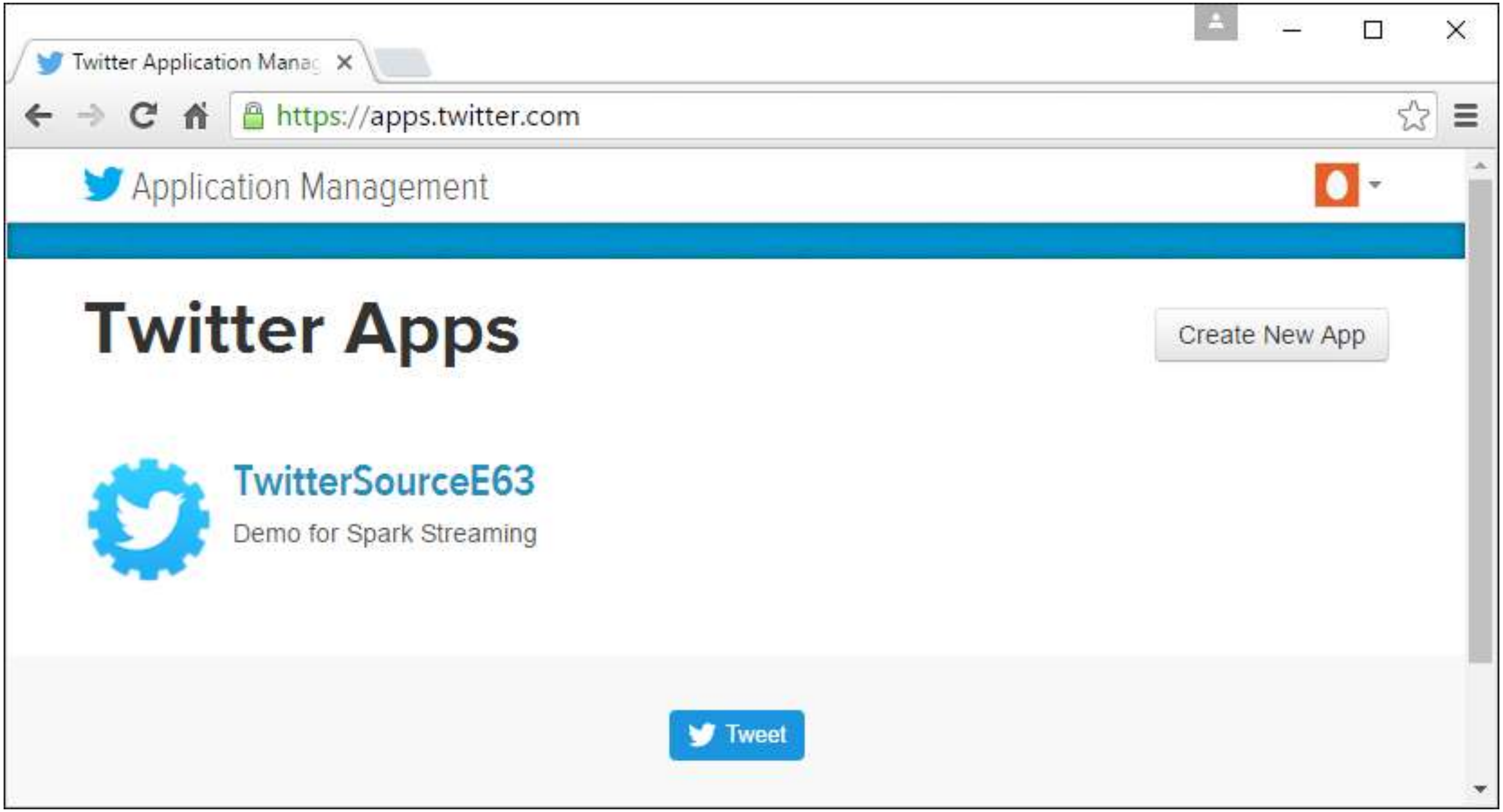
```
 // Hold a reference to the current offset ranges, so it can be used downstream
 final AtomicReference<OffsetRange[]> offsetRanges = new AtomicReference<>();

 directKafkaStream.transformToPair(
   new Function<JavaPairRDD<String, String>, JavaPairRDD<String, String>>() {
     @Override
     public JavaPairRDD<String, String> call(JavaPairRDD<String, String> rdd) throws
Exception {
       OffsetRange[] offsets = ((HasOffsetRanges) rdd.rdd()).offsetRanges();
       offsetRanges.set(offsets);
       return rdd;
     }
   }
 ).map(
   ...
 ).foreachRDD(
   new Function<JavaPairRDD<String, String>, Void>() {
     @Override
     public Void call(JavaPairRDD<String, String> rdd) throws IOException {
       for (OffsetRange o : offsetRanges.get()) {
         System.out.println(
           o.topic() + " " + o.partition() + " " + o.fromOffset() + " " +
o.untilOffset()
         );
       }
       ...
       return null;
     }
   }
 );
```

Near Real-Time Stack

- For data scientists and developers working with real-time data pipelines, Spark Streaming and Kafka are not all that is needed. Often a large volume of data needs to be stored or retrieved, either before or after processing with Spark Stream. Spark Streaming-Kafka-Cassandra has recently emerged as the stack of choice for such applications,.
- This stack satisfies the key requirements for real-time data analytics:
  - **Spark Streaming** is an extension of the core Spark API; it allows integration of near real-time data from disparate event streams.
  - **Kafka:** a messaging system to capture and publish streams of data. With Spark you can ingest data from Kafka, filter that stream down to a smaller data set, augment the data, and then push that refined data set to a persistent data store.
  - **Cassandra:** appears to be an excellent choice when data needs to be written to a scalable and resilient operational database for persistence, easy application development, and real-time analytics.
- These open source frameworks are powerful and well-suited for the requirements of building real-time data pipelines.

Create new Twitter App

- A convenient source of streaming data is Twitter. Please make an account and create a Twitter Application.



Twitter Application Manag

https://apps.twitter.com

Application Management

Twitter Apps

Create New App

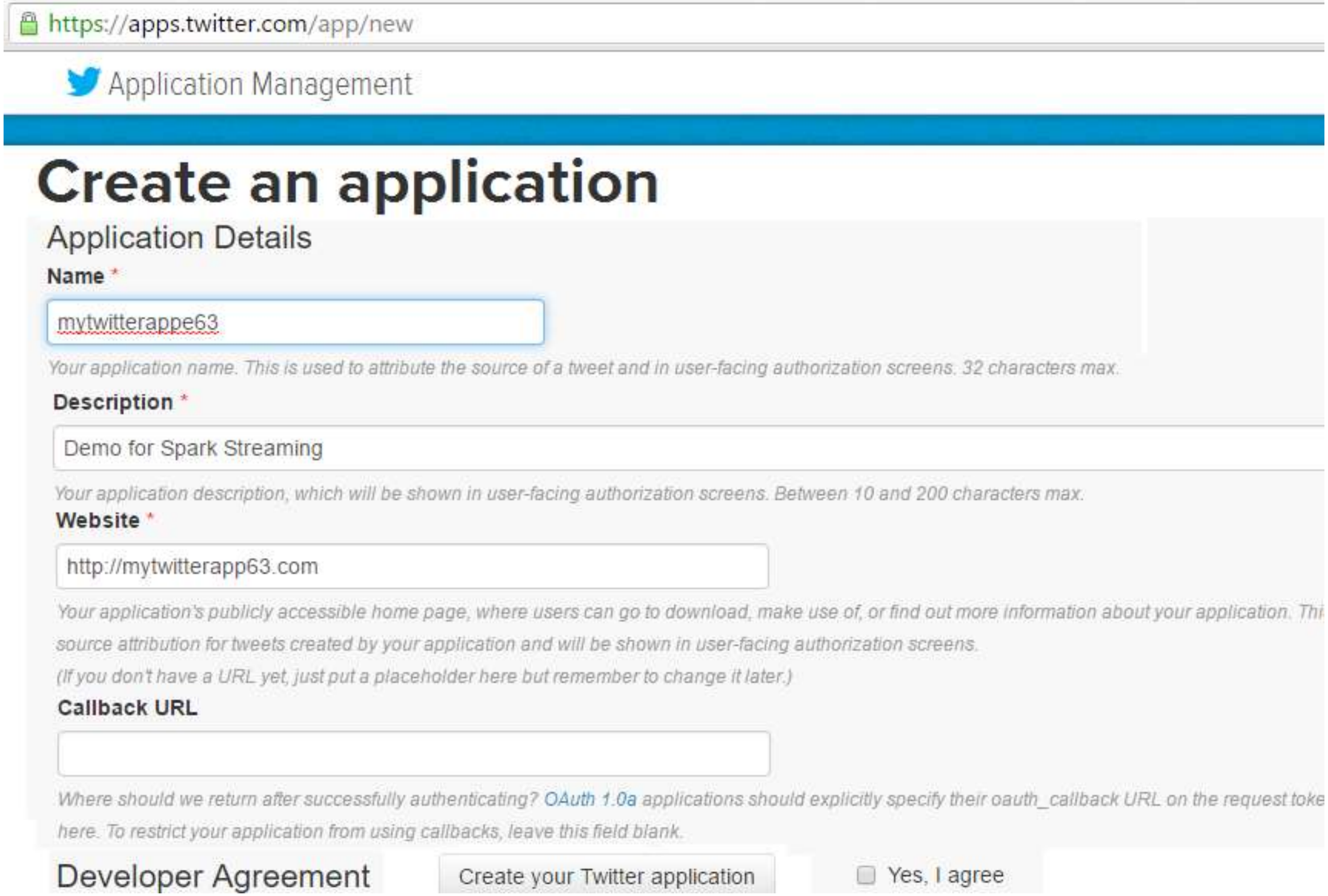
TwitterSourceE63

Demo for Spark Streaming

Tweet

Create Twitter Application

- Website and Callback URL are “dummy” entries.
- Accept the Developer Agreement



https://apps.twitter.com/app/new

Application Management

Create an application

Application Details

Name *

mytwitterappe63

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Demo for Spark Streaming

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

http://mytwitterapp63.com

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. Thi source attribution for tweets created by your application and will be shown in user-facing authorization screens.

(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

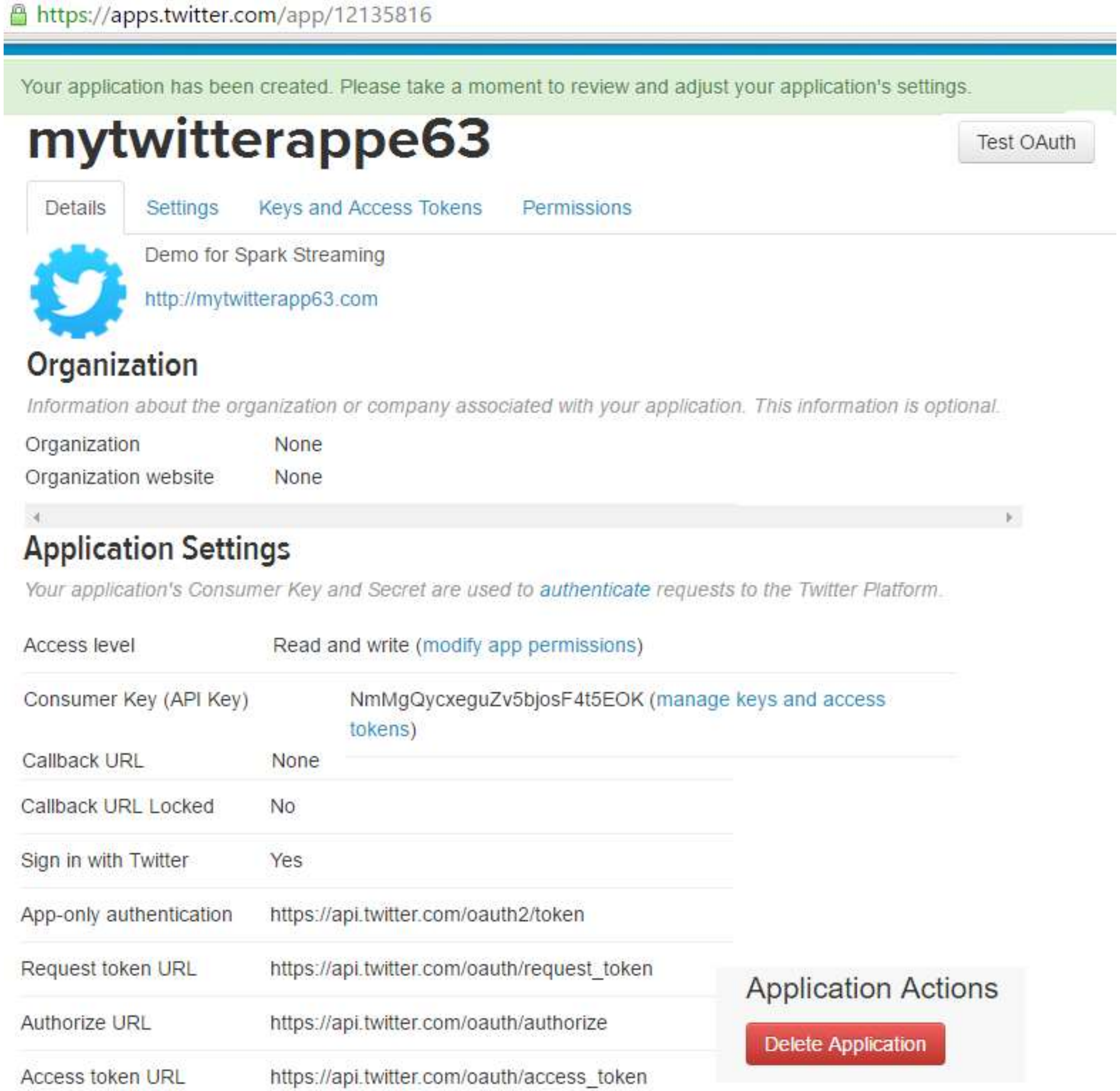
Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request toke here. To restrict your application from using callbacks, leave this field blank.

Developer Agreement

☐ Yes, I agree

Application Created

- Select Keys and Access Tokens



https://apps.twitter.com/app/12135816

Your application has been created. Please take a moment to review and adjust your application's settings.

mytwitterappe63

Demo for Spark Streaming

http://mytwitterapp63.com

Organization

Information about the organization or company associated with your application. This information is optional.

Organization	None
Organization website	None

Application Settings

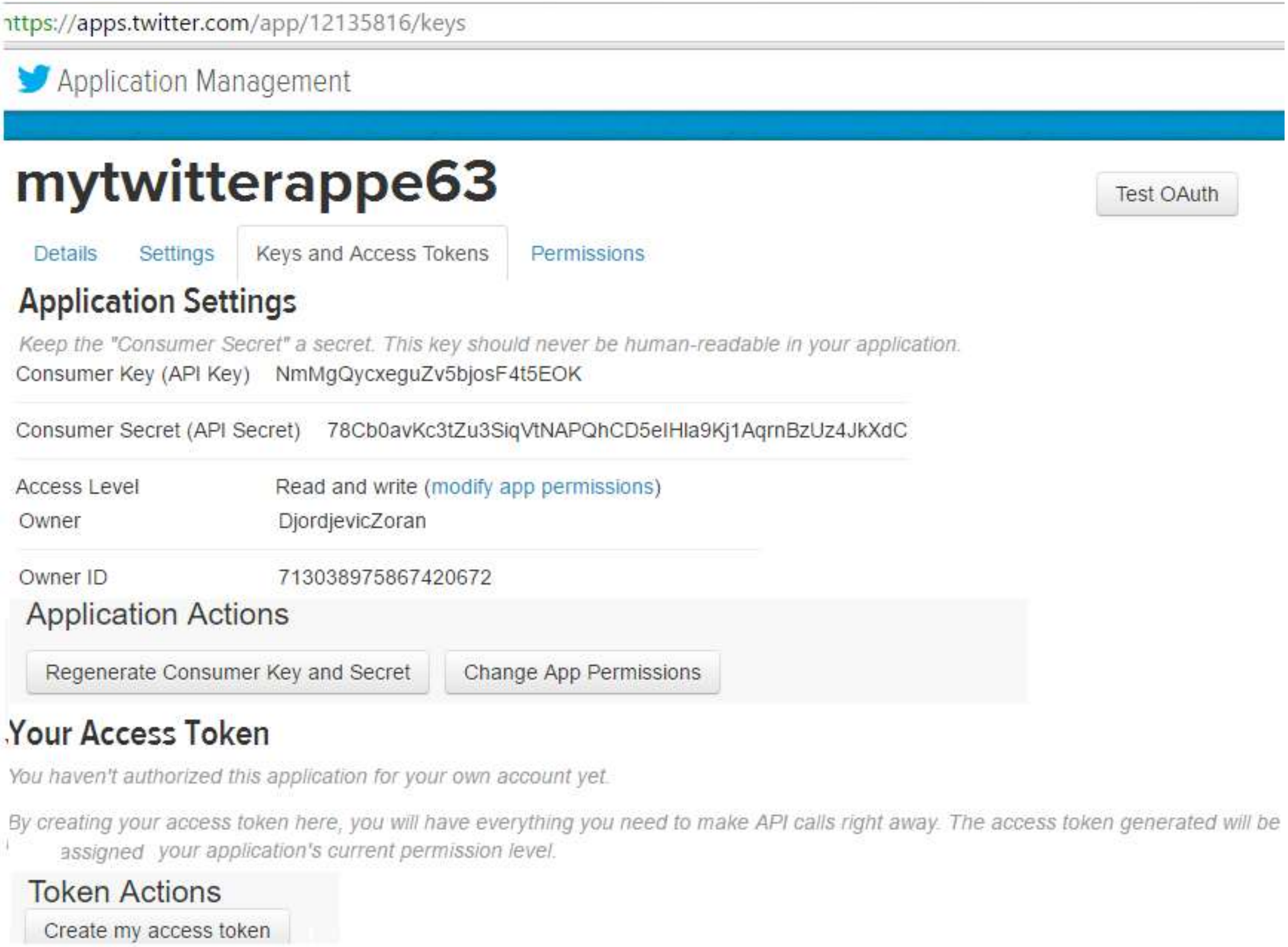
Your application's Consumer Key and Secret are used to authenticate requests to the Twitter Platform.

Access level	Read and write (modify app permissions)
Consumer Key (API Key)	NmMgQycxeguZv5bjosF4t5EOK (manage keys and access tokens)
Callback URL	None
Callback URL Locked	No
Sign in with Twitter	Yes
App-only authentication	https://api.twitter.com/oauth2/token
Request token URL	https://api.twitter.com/oauth/request_token
Authorize URL	https://api.twitter.com/oauth/authorize
Access token URL	https://api.twitter.com/oauth/access_token

Application Actions

Keys and Tokens

- Select “Create my access token”



mytwitterappe63

Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key) NmMgQycxeguZv5bjosF4t5EOK

Consumer Secret (API Secret) 78Cb0avKc3tZu3SiqVtNAPQhCD5eIHla9Kj1AqrnBzUz4JkXdC

Access Level Read and write (modify app permissions)

Owner DjordjevicZoran

Owner ID 713038975867420672

Application Actions

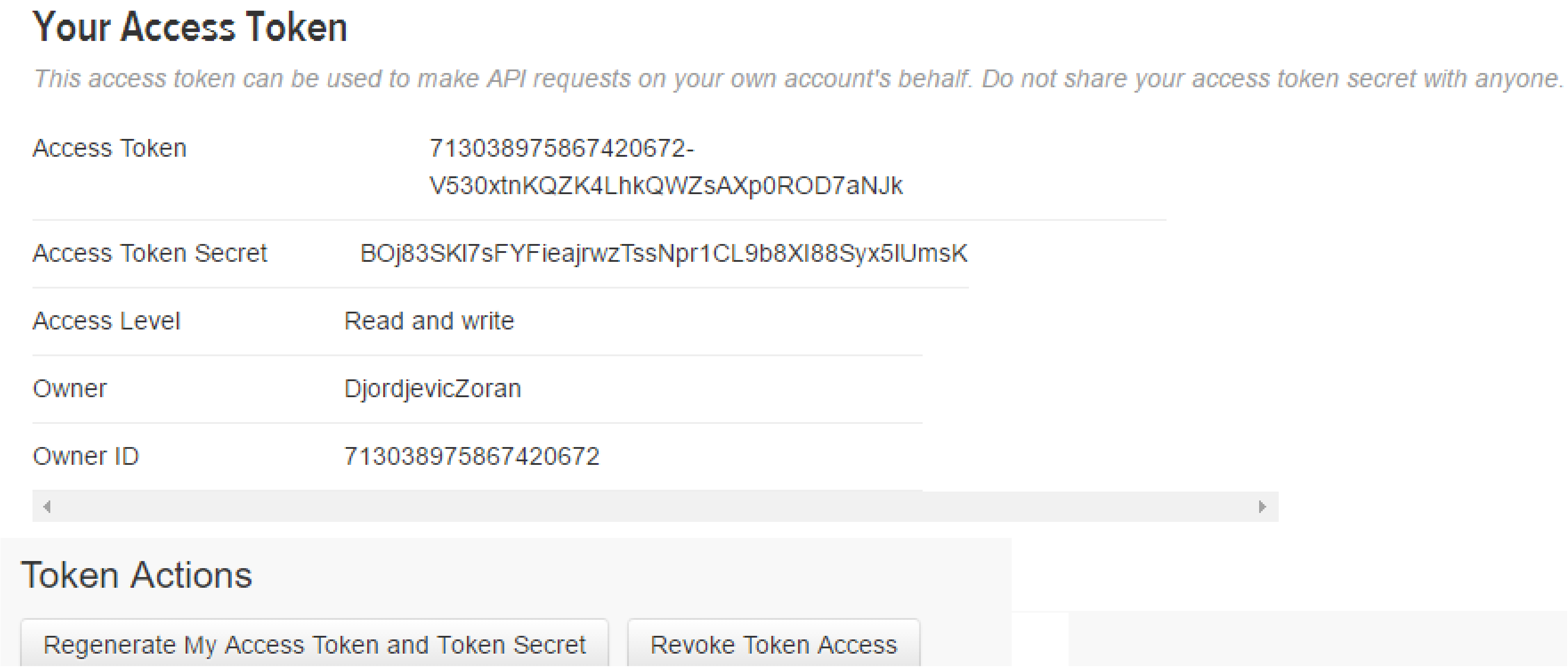
Your Access Token

You haven't authorized this application for your own account yet.

By creating your access token here, you will have everything you need to make API calls right away. The access token generated will be assigned your application's current permission level.

Token Actions

Your Access Token



Your Access Token

This access token can be used to make API requests on your own account's behalf. Do not share your access token secret with anyone.

Access Token	713038975867420672- V530xtnKQZK4LhkQWZsAXp0ROD7aNJk
Access Token Secret	BOj83SKI7sFYFieajrwzTssNpr1CL9b8XI88Syx5lUmsK
Access Level	Read and write
Owner	DjordjevicZoran
Owner ID	713038975867420672

Token Actions

- Record all these values somewhere