Apache Flink

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

Apache Flink is an open source stream processing framework, which has both batch and stream processing capabilities. Apache Flink is very similar to Apache Spark, but it follows stream-first approach. It is also a part of Big Data tools list. This tutorial explains the basics of Flink Architecture Ecosystem and its APIs.

Audience

This tutorial is for beginners who are aspiring to become experts with stream processing in Big Data Domain. It is also ideal for Big Data professionals who know Apache Hadoop and Apache Spark.

Prerequisites

Before proceeding with this tutorial, you should have basic knowledge of Scala programming and any Linux operating system.

Copyright & Disclaimer

Table of Contents

1. Apache Flink — Big Data Platform

The advancement of data in the last 10 years has been enormous; this gave rise to a term 'Big Data'. There is no fixed size of data, which you can call as big data; any data that your traditional system (RDBMS) is not able to handle is Big Data. This Big Data can be in structured, semi-structured or un-structured format. Initially, there were three dimensions to data – Volume, Velocity, Variety. The dimensions have now gone beyond just the three Vs. We have now added other Vs – Veracity, Validity, Vulnerability, Value, Variability, etc.

Big Data led to the emergence of multiple tools and frameworks that help in the storage and processing of data. There are a few popular big data frameworks such as Hadoop, Spark, Hive, Pig, Storm and Zookeeper. It also gave opportunity to create Next Gen products in multiple domains like Healthcare, Finance, Retail, E-Commerce and more.

Whether it is an MNC or a start-up, everyone is leveraging Big Data to store and process it and take smarter decisions.

2. Apache Flink — Batch vs Real-time Processing

In terms of Big Data, there are two types of processing:

- Batch Processing
- Real-time Processing

Processing based on the data collected over time is called Batch Processing. For example, a bank manager wants to process past one-month data (collected over time) to know the number of cheques that got cancelled in the past 1 month.

Processing based on immediate data for instant result is called Real-time Processing. For example, a bank manager getting a fraud alert immediately after a fraud transaction (instant result) has occurred.

The table given below lists down the differences between Batch and Real-Time Processing:

Batch Processing	Real-Time Processing
Static Files	Event Streams
Processed Periodically in minute, hour, day etc.	Processed immediately (nanoseconds)
Past data on disk storage	In Memory Storage
Example: Bill Generation	Example: ATM Transaction Alert

These days, real-time processing is being used a lot in every organization. Use cases like fraud detection, real-time alerts in healthcare and network attack alert require real-time processing of instant data; a delay of even few milliseconds can have a huge impact.

An ideal tool for such real time use cases would be the one, which can input data as stream and not batch. Apache Flink is that real-time processing tool.

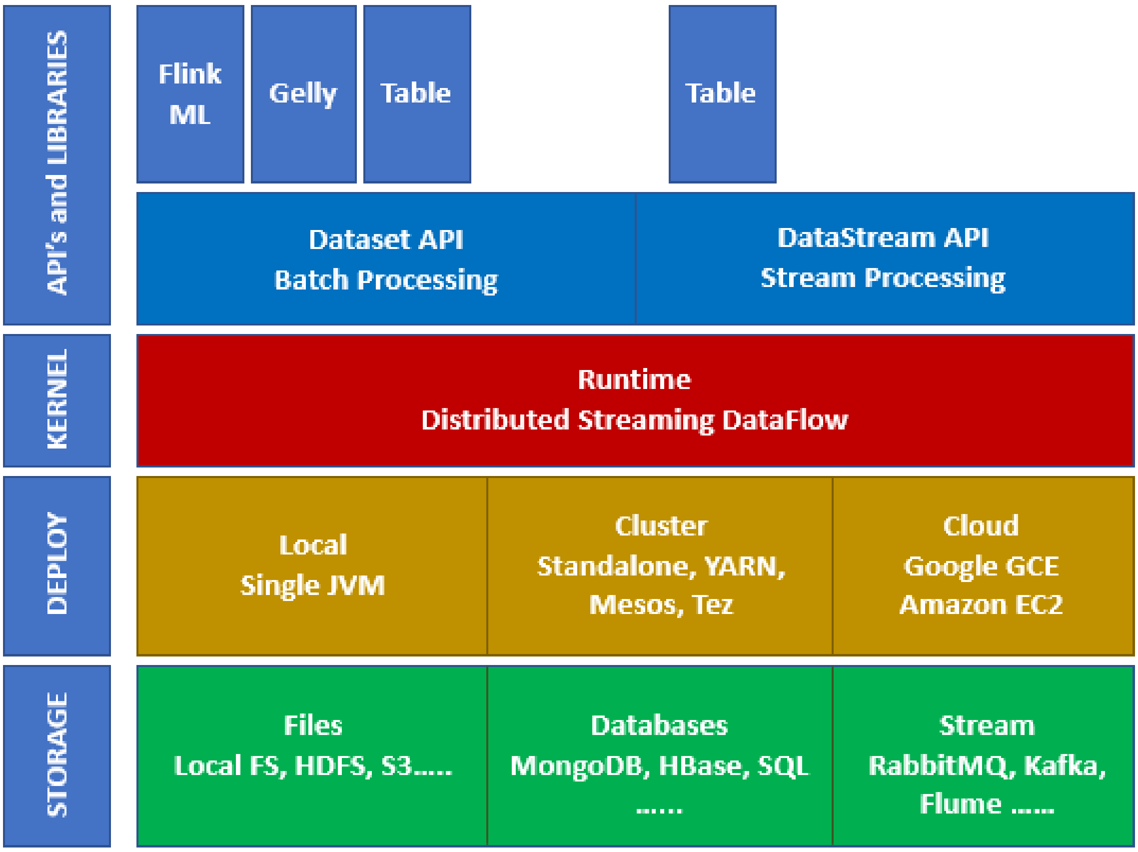
3. Apache Flink — Introduction

Apache Flink is a real-time processing framework which can process streaming data. It is an open source stream processing framework for high-performance, scalable, and accurate real-time applications. It has true streaming model and does not take input data as batch or micro-batches.

Apache Flink was founded by Data Artisans company and is now developed under Apache License by Apache Flink Community. This community has over 479 contributors and 15500 + commits so far.

Ecosystem on Apache Flink

The diagram given below shows the different layers of Apache Flink Ecosystem:



Storage

Apache Flink has multiple options from where it can Read/Write data. Below is a basic storage list:

- HDFS (Hadoop Distributed File System)
- Local File System

- S3
- RDBMS (MySQL, Oracle, MS SQL etc.)
- MongoDB
- HBase
- Apache Kafka
- Apache Flume

Deploy

You can deploy Apache Fink in local mode, cluster mode or on cloud. Cluster mode can be standalone, YARN, MESOS.

On cloud, Flink can be deployed on AWS or GCP.

Kernel

This is the runtime layer, which provides distributed processing, fault tolerance, reliability, native iterative processing capability and more.

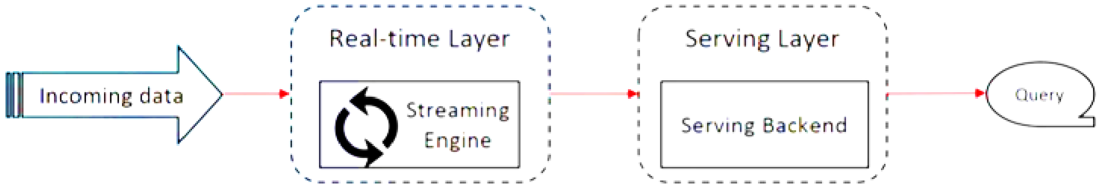
APIs & Libraries

This is the top layer and most important layer of Apache Flink. It has Dataset API, which takes care of batch processing, and Datastream API, which takes care of stream processing. There are other libraries like Flink ML (for machine learning), Gelly (for graph processing), Tables for SQL. This layer provides diverse capabilities to Apache Flink.

4. Apache Flink — Architecture

Apache Flink works on Kappa architecture. Kappa architecture has a single processor - stream, which treats all input as stream and the streaming engine processes the data in real-time. Batch data in kappa architecture is a special case of streaming.

The following diagram shows the **Apache Flink Architecture**.

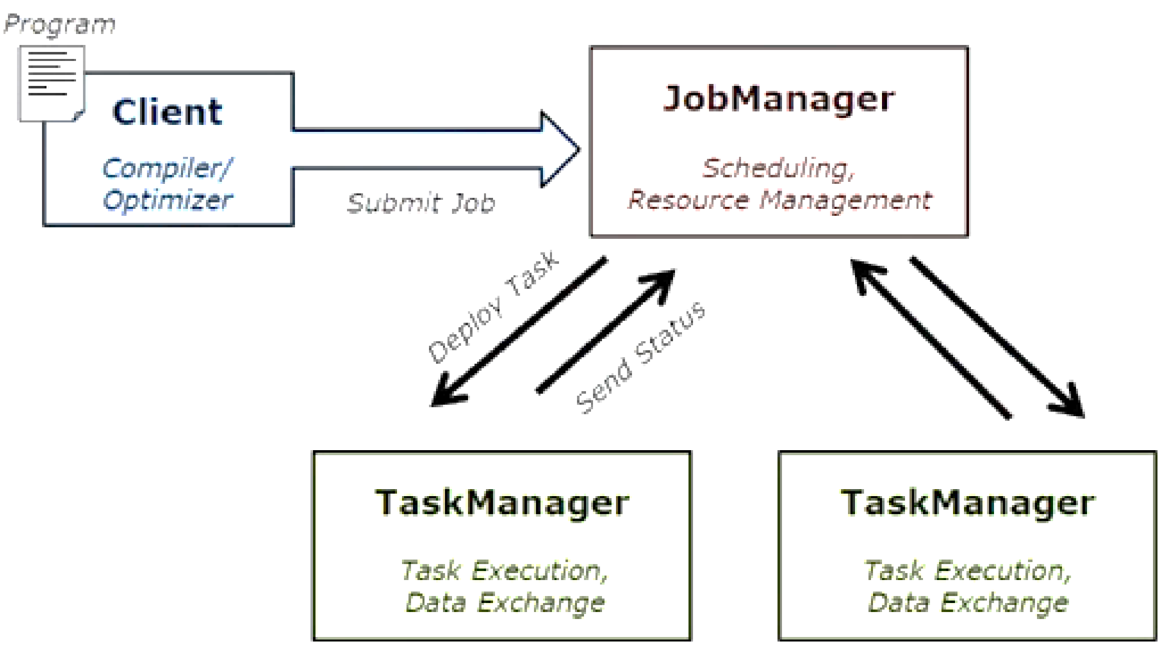


The key idea in Kappa architecture is to handle both batch and real-time data through a single stream processing engine.

Most big data framework works on Lambda architecture, which has separate processors for batch and streaming data. In Lambda architecture, you have separate codebases for batch and stream views. For querying and getting the result, the codebases need to be merged. Not maintaining separate codebases/views and merging them is a pain, but Kappa architecture solves this issue as it has only one view – real-time, hence merging of codebase is not required.

That does not mean Kappa architecture replaces Lambda architecture, it completely depends on the use-case and the application that decides which architecture would be preferable.

The following diagram shows Apache Flink job execution architecture.



Program

It is a piece of code, which you run on the Flink Cluster.

Client

It is responsible for taking code (program) and constructing job dataflow graph, then passing it to JobManager. It also retrieves the Job results.

JobManager

After receiving the Job Dataflow Graph from Client, it is responsible for creating the execution graph. It assigns the job to TaskManagers in the cluster and supervises the execution of the job.

TaskManager

It is responsible for executing all the tasks that have been assigned by JobManager. All the TaskManagers run the tasks in their separate slots in specified parallelism. It is responsible to send the status of the tasks to JobManager.

Features of Apache Flink

The features of Apache Flink are as follows:

- It has a streaming processor, which can run both batch and stream programs.
- It can process data at lightning fast speed.
- APIs available in Java, Scala and Python.
- Provides APIs for all the common operations, which is very easy for programmers to use.
- Processes data in low latency (nanoseconds) and high throughput.
- Its fault tolerant. If a node, application or a hardware fails, it does not affect the cluster.
- Can easily integrate with Apache Hadoop, Apache MapReduce, Apache Spark, HBase and other big data tools.
- In-memory management can be customized for better computation.
- It is highly scalable and can scale upto thousands of node in a cluster.
- Windowing is very flexible in Apache Flink.
- Provides Graph Processing, Machine Learning, Complex Event Processing libraries.

5. Apache Flink — System Requirements

The following are the system requirements to download and work on Apache Flink –

Recommended Operating System

- Microsoft Windows 10
- Ubuntu 16.04 LTS
- Apple macOS 10.13/High Sierra

Memory Requirement

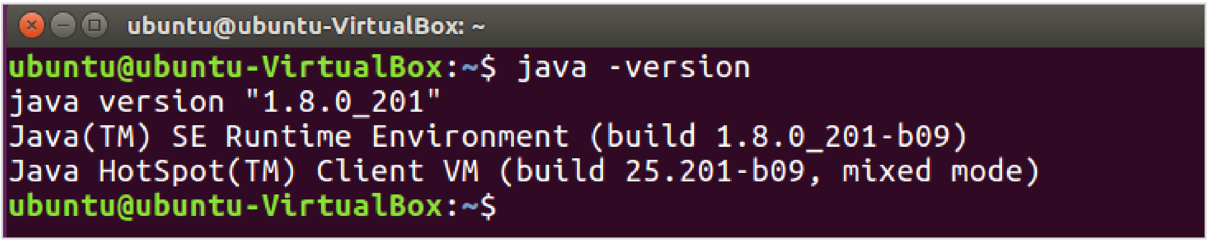
- Memory - Minimum 4 GB, Recommended 8 GB
- Storage Space - 30 GB

Note – Java 8 must be available with environment variables already set.

6. Apache Flink — Setup/Installation

Before the start with the setup/ installation of Apache Flink, let us check whether we have Java 8 installed in our system.

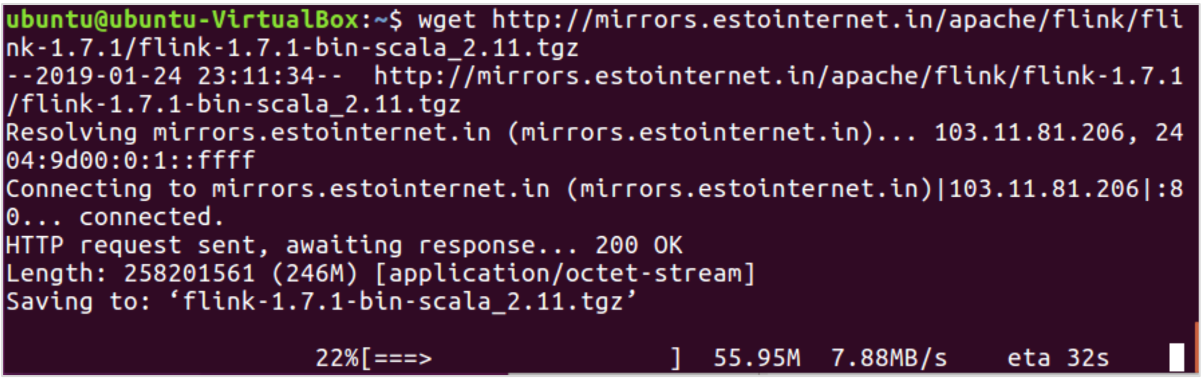
Java - version



```
ubuntu@ubuntu-VirtualBox: ~
ubuntu@ubuntu-VirtualBox:~$ java -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) Client VM (build 25.201-b09, mixed mode)
ubuntu@ubuntu-VirtualBox:~$
```

We will now proceed by downloading Apache Flink.

```
wget http://mirrors.estointernet.in/apache/flink/flink-1.7.1/flink-1.7.1-bin-scala_2.11.tgz
```



```
ubuntu@ubuntu-VirtualBox:~$ wget http://mirrors.estointernet.in/apache/flink/flink-1.7.1/flink-1.7.1-bin-scala_2.11.tgz
--2019-01-24 23:11:34--  http://mirrors.estointernet.in/apache/flink/flink-1.7.1/flink-1.7.1-bin-scala_2.11.tgz
Resolving mirrors.estointernet.in (mirrors.estointernet.in)... 103.11.81.206, 2404:9d00:0:1::ffff
Connecting to mirrors.estointernet.in (mirrors.estointernet.in)|103.11.81.206|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 258201561 (246M) [application/octet-stream]
Saving to: ‘flink-1.7.1-bin-scala_2.11.tgz’

                    22%[===>               ]  55.95M  7.88MB/s    eta 32s
```

Now, uncompress the tar file.

```
tar -xzf flink-1.7.1-bin-scala_2.11.tgz
```

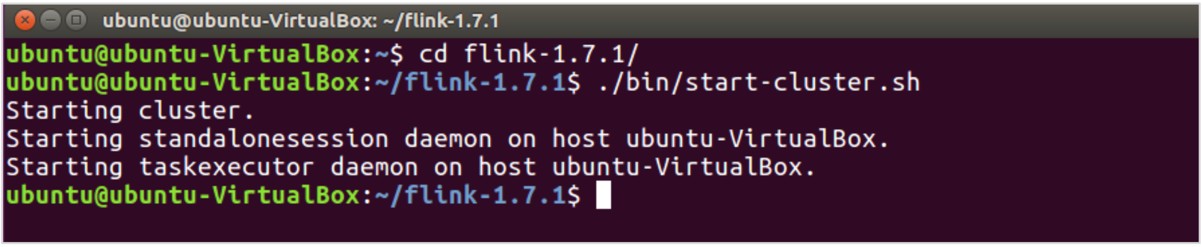
```
ubuntu@ubuntu-VirtualBox:~$ tar -xzf flink-1.7.1-bin-scala_2.11.tgz
ubuntu@ubuntu-VirtualBox:~$ ls
Desktop     examples.desktop                Music     Templates
Documents   flink-1.7.1                     Pictures  Videos
Downloads   flink-1.7.1-bin-scala_2.11.tgz  Public
ubuntu@ubuntu-VirtualBox:~$
```

Go to Flink's home directory.

```
cd flink-1.7.1/
```

Start the Flink Cluster.

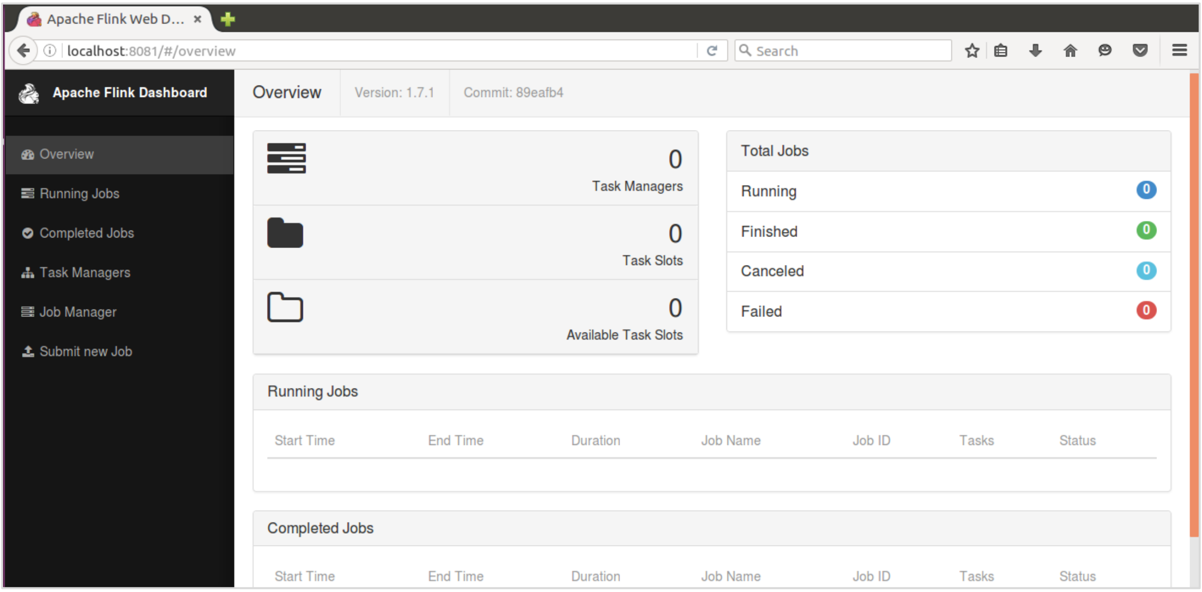
```
./bin/start-cluster.sh
```



Open the Mozilla browser and go to the below URL, it will open the Flink Web Dashboard.

http://localhost:8081

This is how the User Interface of Apache Flink Dashboard looks like.



Now the Flink cluster is up and running.

7. Apache Flink — API Concepts

Flink has a rich set of APIs using which developers can perform transformations on both batch and real-time data. A variety of transformations includes mapping, filtering, sorting, joining, grouping and aggregating. These transformations by Apache Flink are performed on distributed data. Let us discuss the different APIs Apache Flink offers.

Dataset API

Dataset API in Apache Flink is used to perform batch operations on the data over a period. This API can be used in Java, Scala and Python. It can apply different kinds of transformations on the datasets like filtering, mapping, aggregating, joining and grouping.

Datasets are created from sources like local files or by reading a file from a particular sourse and the result data can be written on different sinks like distributed files or command line terminal. This API is supported by both Java and Scala programming languages.

Here is a Wordcount program of Dataset API:

```
public class WordCountProg {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

        DataSet<String> text = env.fromElements(
            "Hello",
            "My Dataset API Flink Program");

        DataSet<Tuple2<String, Integer>> wordCounts = text
            .flatMap(new LineSplitter())
            .groupBy(0)
            .sum(1);

        wordCounts.print();
    }

    public static class LineSplitter implements FlatMapFunction<String,
Tuple2<String, Integer>> {
        @Override
        public void flatMap(String line, Collector<Tuple2<String, Integer>>
out) {
```

```
            for (String word : line.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }
}
```

DataStream API

This API is used for handling data in continuous stream. You can perform various operations like filtering, mapping, windowing, aggregating on the stream data. There are various sources on this data stream like message queues, files, socket streams and the result data can be written on different sinks like command line terminal. Both Java and Scala programming languages support this API.

Here is a streaming Wordcount program of DataStream API, where you have continuous stream of word counts and the data is grouped in the second window.

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class WindowWordCountProg {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Tuple2<String, Integer>> dataStream = env
                .socketTextStream("localhost", 9999)
                .flatMap(new Splitter())
                .keyBy(0)
                .timeWindow(Time.seconds(5))
                .sum(1);

        dataStream.print();
```

```
        env.execute("Streaming WordCount Example");
    }

    public static class Splitter implements FlatMapFunction<String,
Tuple2<String, Integer>> {
        @Override
        public void flatMap(String sentence, Collector<Tuple2<String, Integer>>
out) throws Exception {
            for (String word: sentence.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }

}
```

8. Apache Flink — Table API and SQL

Table API is a relational API with SQL like expression language. This API can do both batch and stream processing. It can be embedded with Java and Scala Dataset and Datastream APIs. You can create tables from existing Datasets and Datastreams or from external data sources. Through this relational API, you can perform operations like join, aggregate, select and filter. Whether the input is batch or stream, the semantics of the query remains the same.

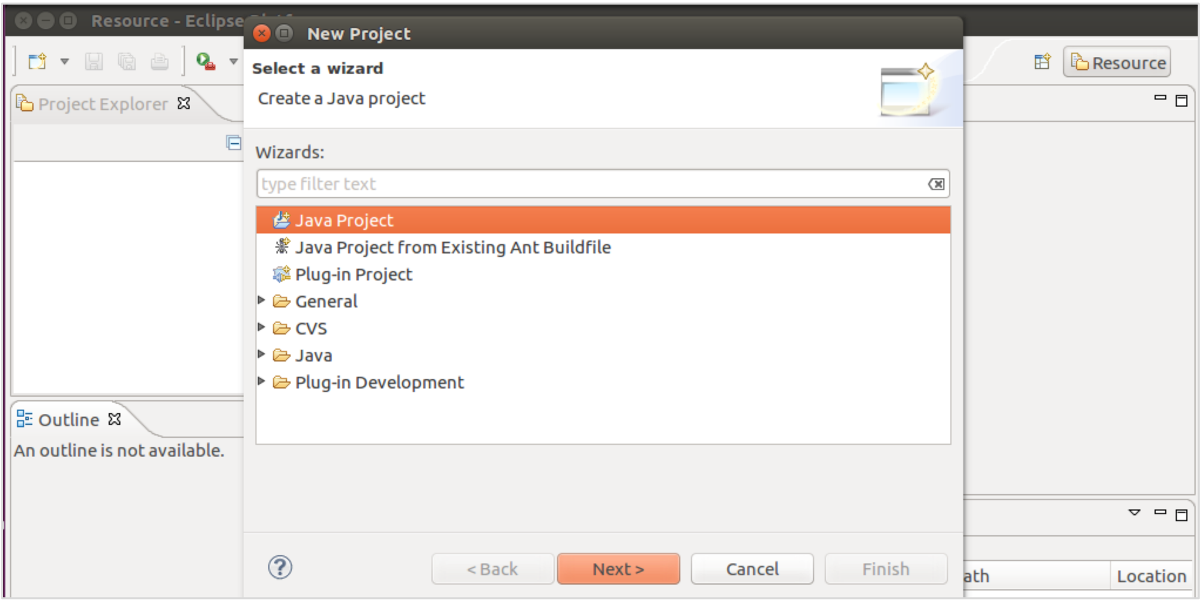
Here is a sample Table API program:

```
// for batch programs use ExecutionEnvironment instead of
StreamExecutionEnvironment
val env = StreamExecutionEnvironment.getExecutionEnvironment


// create a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)


// register a Table
tableEnv.registerTable("table1", ...)           // or
tableEnv.registerTableSource("table2", ...)     // or
tableEnv.registerExternalCatalog("extCat", ...)
// register an output Table
tableEnv.registerTableSink("outputTable", ...);


// create a Table from a Table API query
val tapiResult = tableEnv.scan("table1").select(...)
// Create a Table from a SQL query
val sqlResult  = tableEnv.sqlQuery("SELECT ... FROM table2 ...")


// emit a Table API result Table to a TableSink, same for SQL result
tapiResult.insertInto("outputTable")


// execute
env.execute()
```

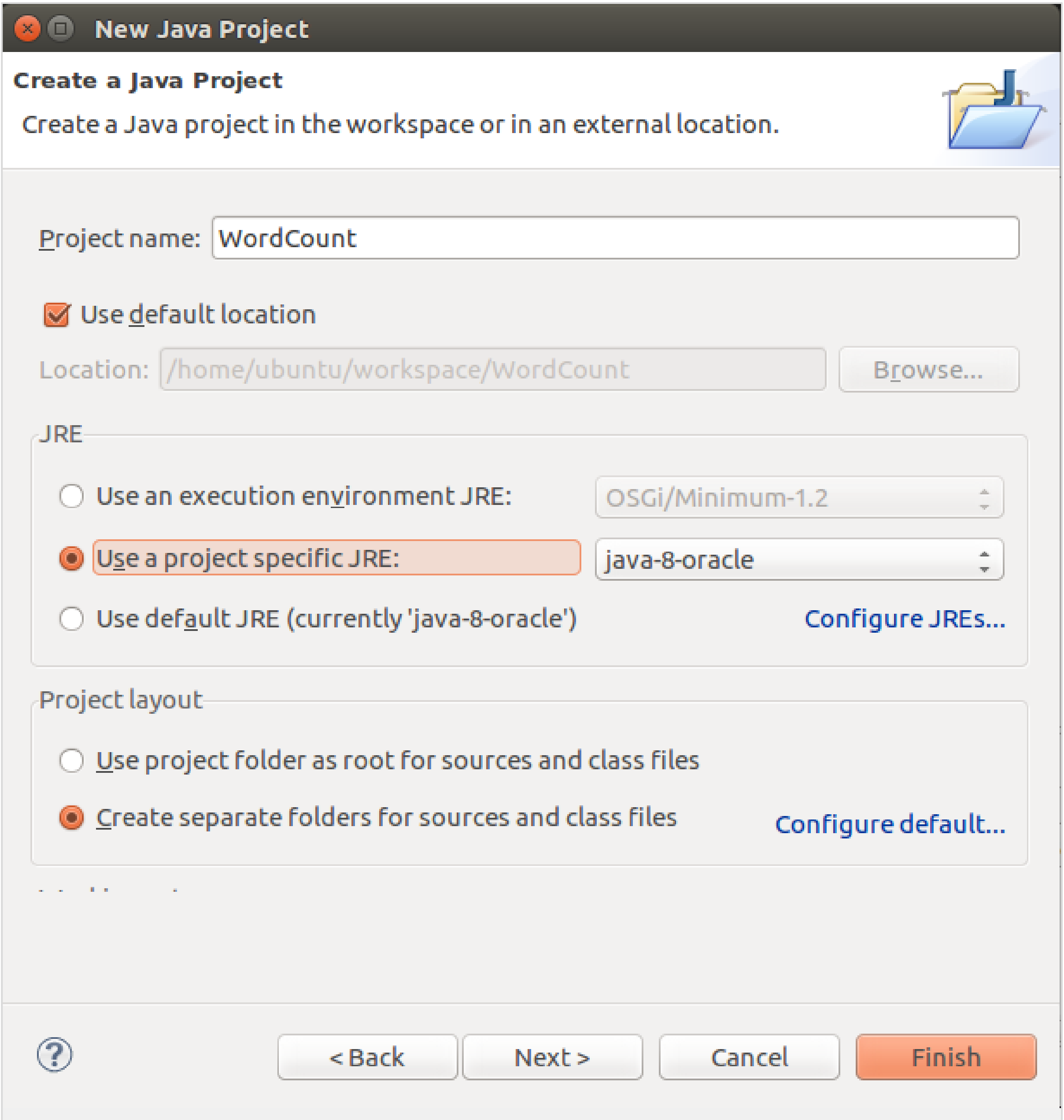
9. Apache Flink — Creating a Flink Application

In this chapter, we will learn how to create a Flink application.

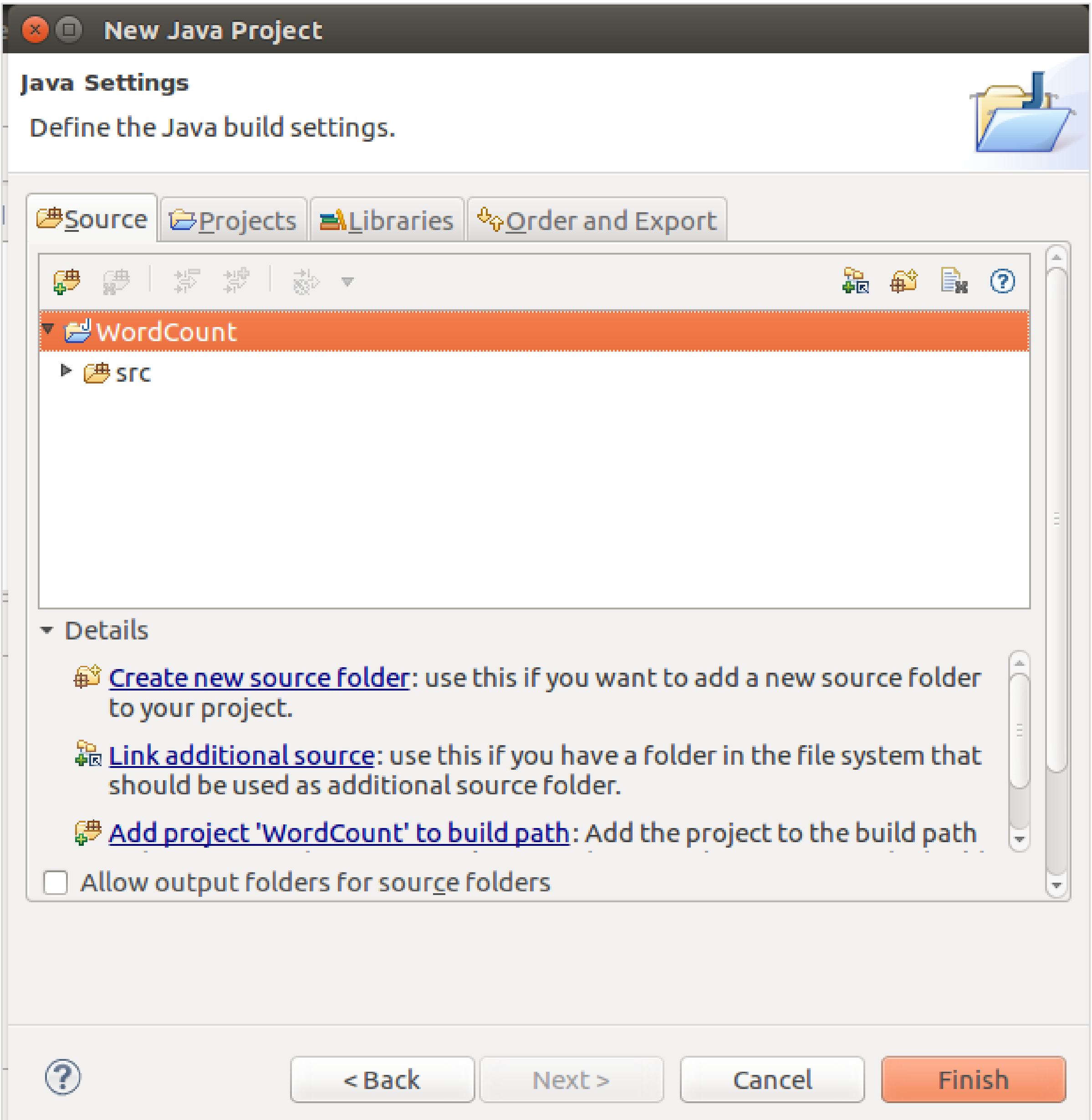
Open Eclipse IDE, click on New Project and Select Java Project.



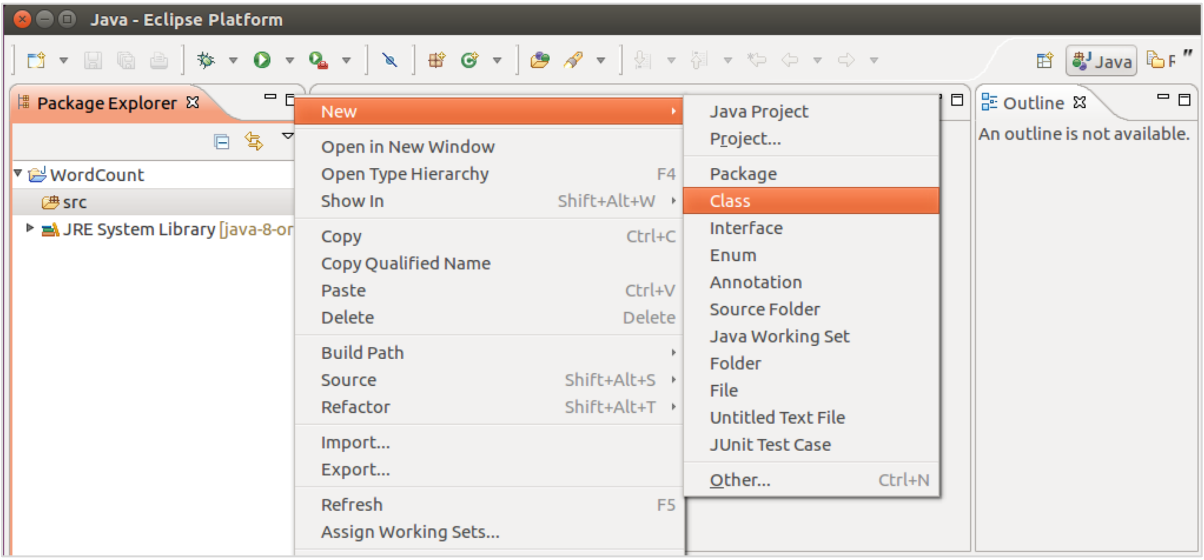
Give Project Name and click on Finish.



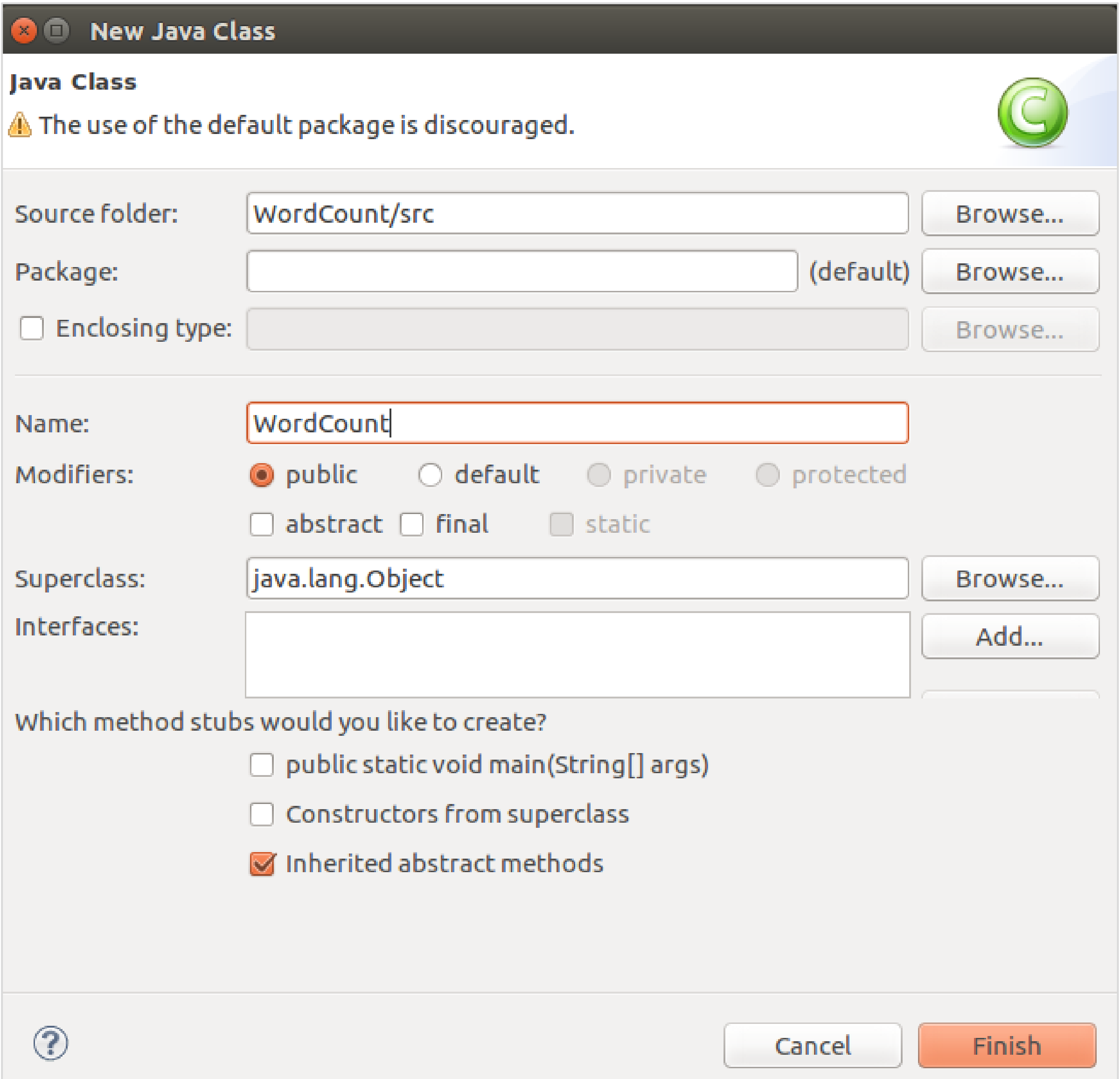
Now, click on Finish as shown in the following screenshot.



Now, right-click on **src** and go to New >> Class.



Give a class name and click on Finish.



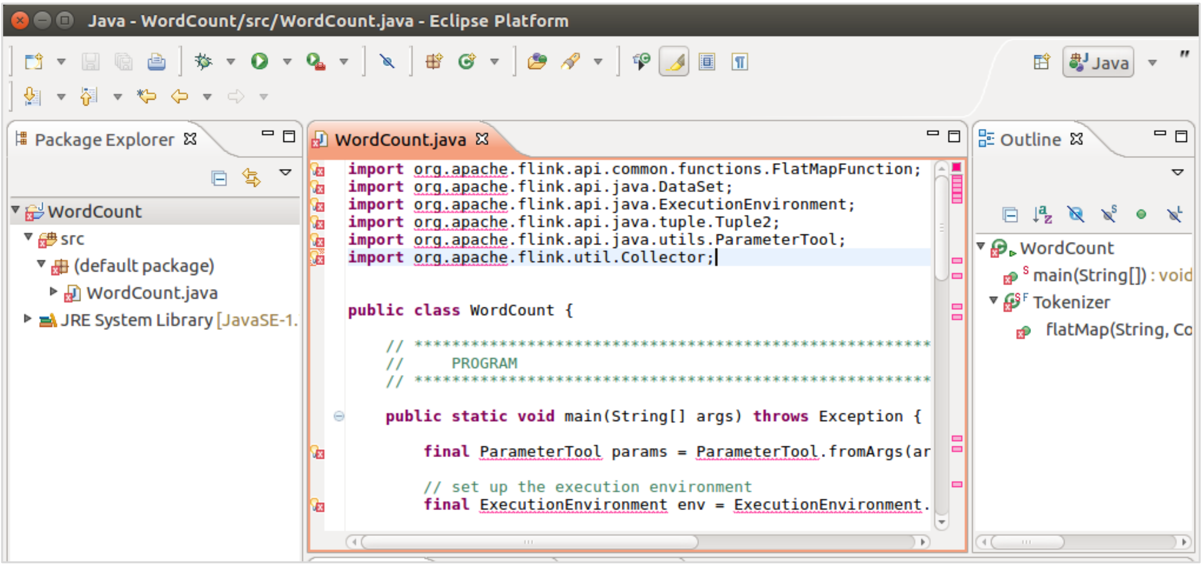
Copy and paste the below code in the Editor.

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.util.Collector;


public class WordCount {
```

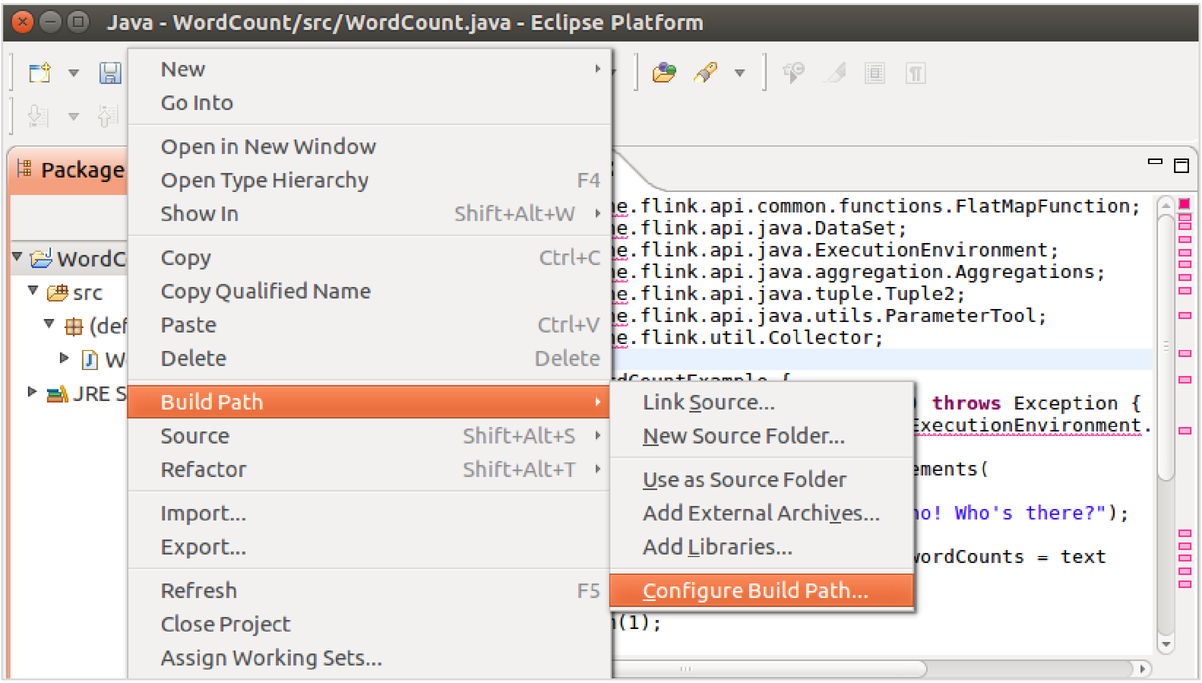
```
    //
*************************************************************************
    //     PROGRAM
    //
*************************************************************************

    public static void main(String[] args) throws Exception {

            final ParameterTool params = ParameterTool.fromArgs(args);

            // set up the execution environment
            final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

            // make parameters available in the web interface
            env.getConfig().setGlobalJobParameters(params);

            // get input data
            DataSet<String> text = env.readTextFile(params.get("input"));

            DataSet<Tuple2<String, Integer>> counts =
                        // split up the lines in pairs (2-tuples) containing:
(word,1)
                        text.flatMap(new Tokenizer())
                        // group by the tuple field "0" and sum up tuple field
"1"
                        .groupBy(0)
                        .sum(1);

            // emit result
            if (params.has("output")) {
                counts.writeAsCsv(params.get("output"), "\n", " ");
                // execute program
                env.execute("WordCount Example");
            } else {
                System.out.println("Printing result to stdout. Use --output
to specify output path.");
                counts.print();
```

```
            }

      }

      //
*************************************************************************
      //     USER FUNCTIONS
      //
*************************************************************************

      public static final class Tokenizer implements FlatMapFunction<String,
Tuple2<String, Integer>> {

            public void flatMap(String value, Collector<Tuple2<String,
Integer>> out) {
                  // normalize and split the line
                  String[] tokens = value.toLowerCase().split("\\W+");

                  // emit the pairs
                  for (String token : tokens) {
                        if (token.length() > 0) {
                              out.collect(new Tuple2<>(token, 1));
                        }
                  }
            }
      }
}
```

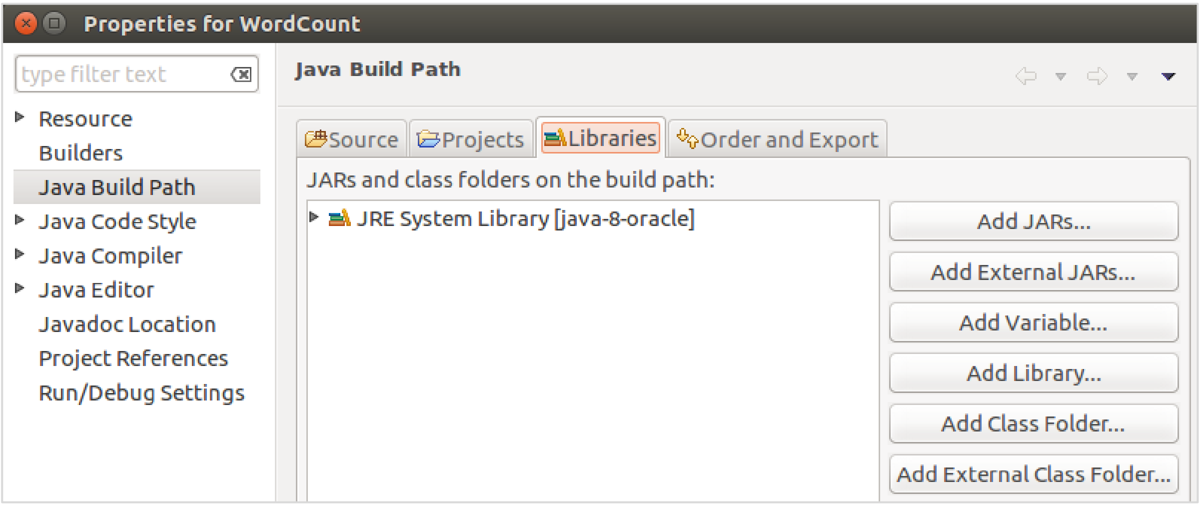
You will get many errors in the editor, because Flink libraries need to be added to this project.



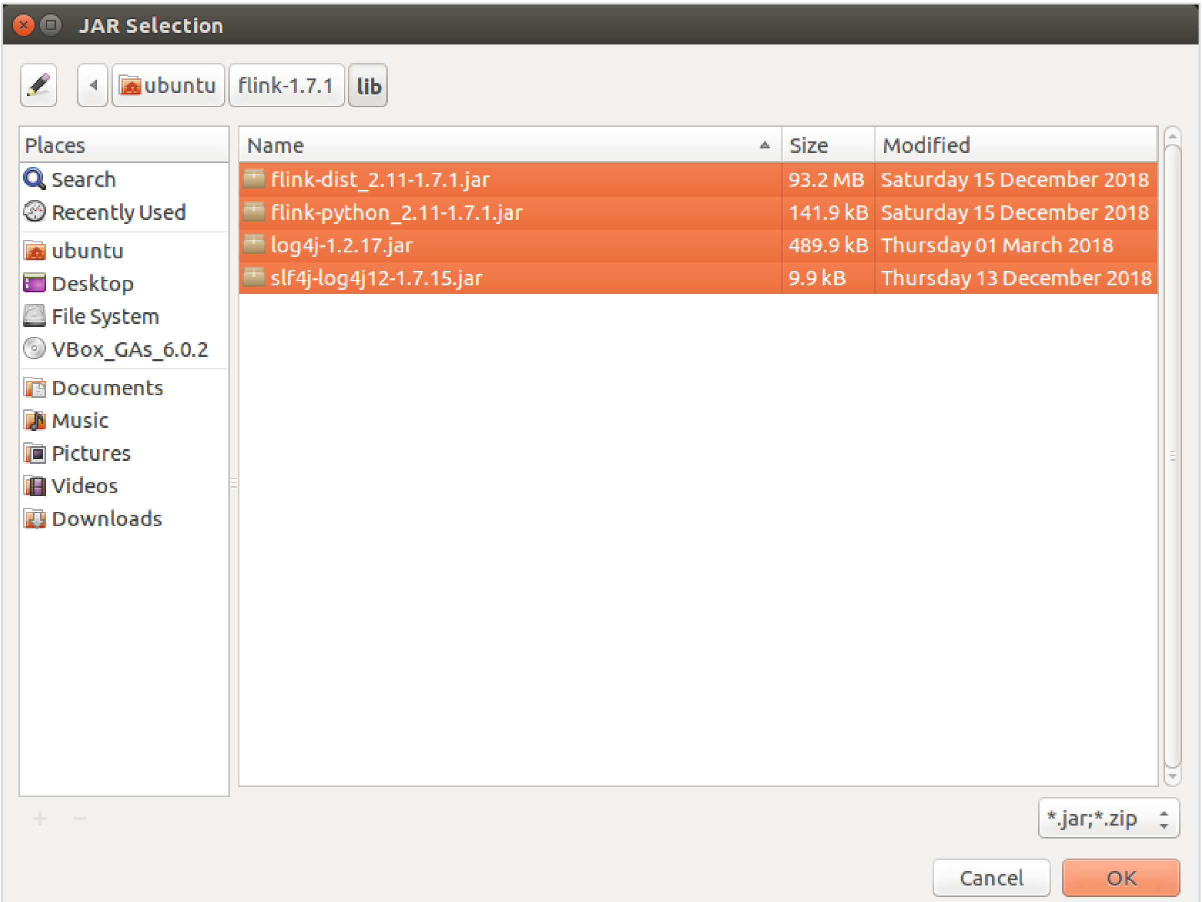
Right-click on the project >> Build Path >> Configure Build Path.



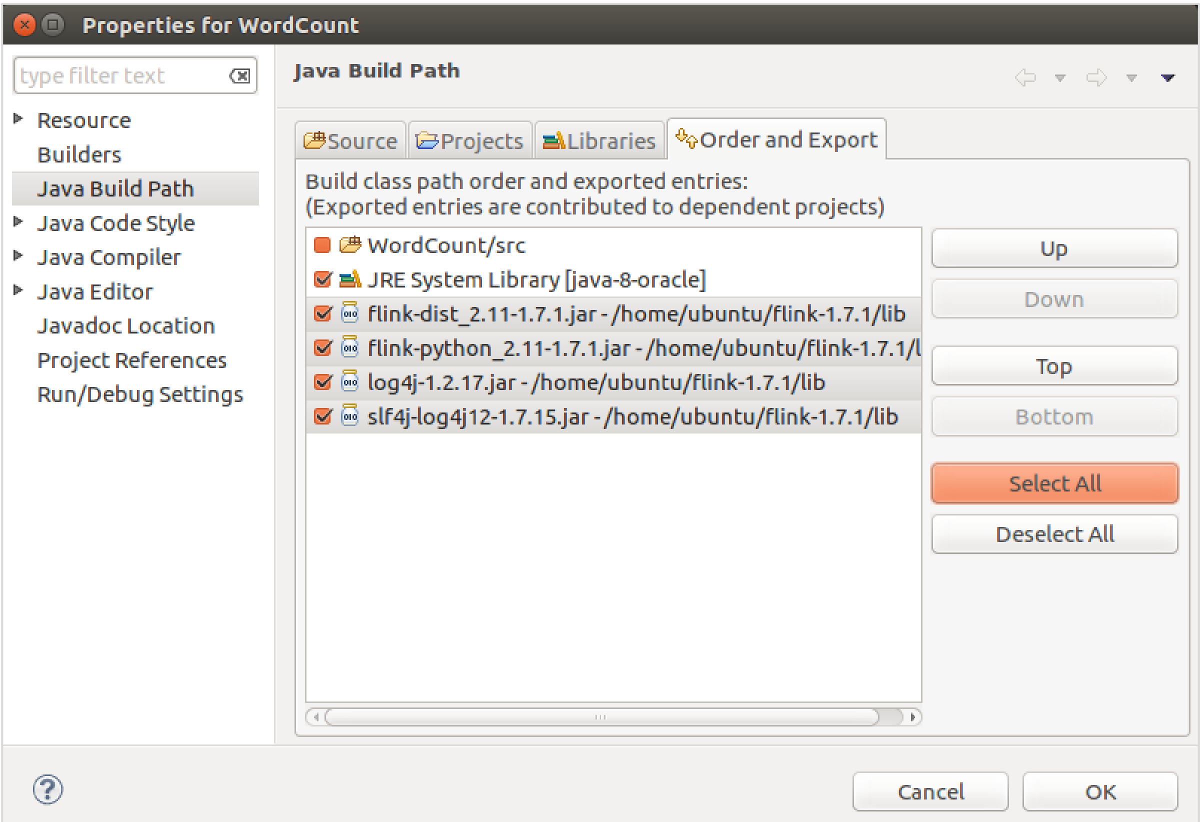
Select the Libraries tab and click on Add External JARs.



Go to Flink's lib directory, select all the 4 libraries and click on OK.

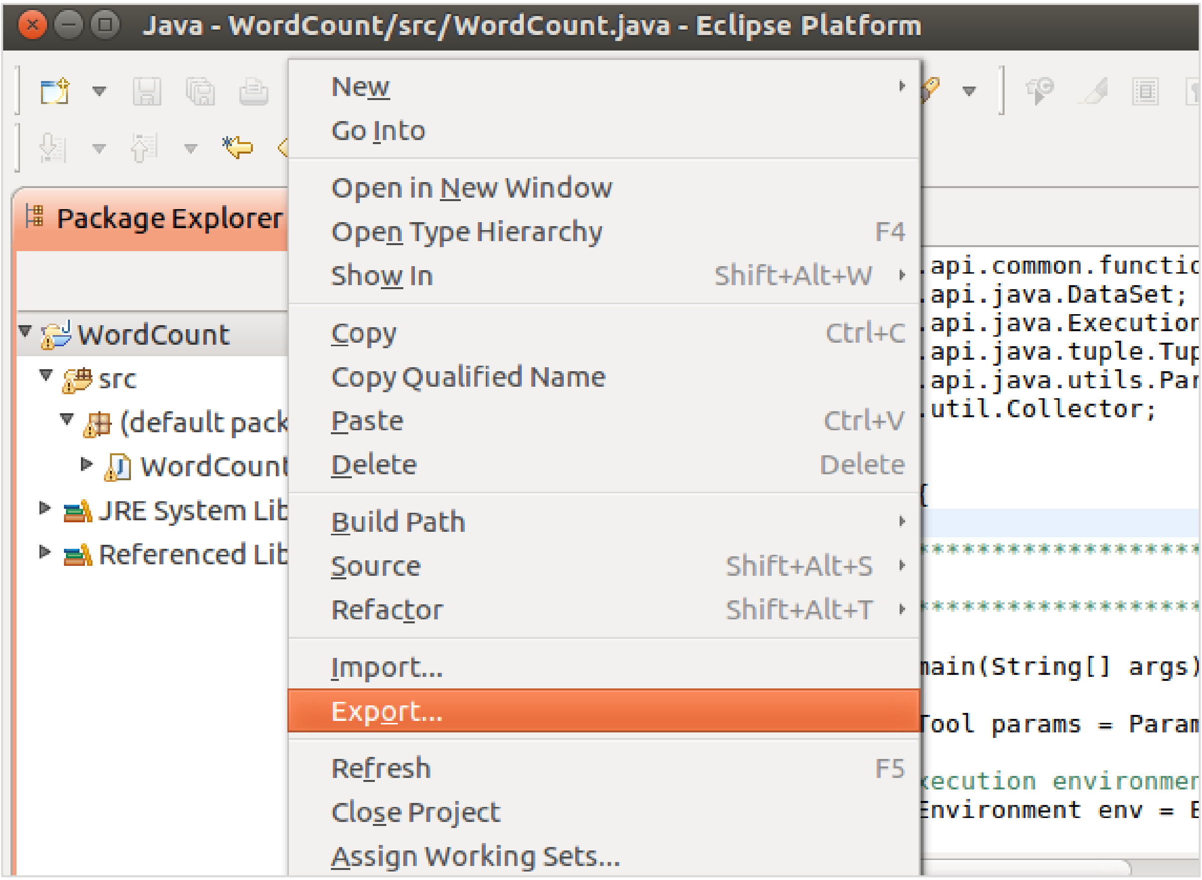


Go to the Order and Export tab, select all the libraries and click on OK.

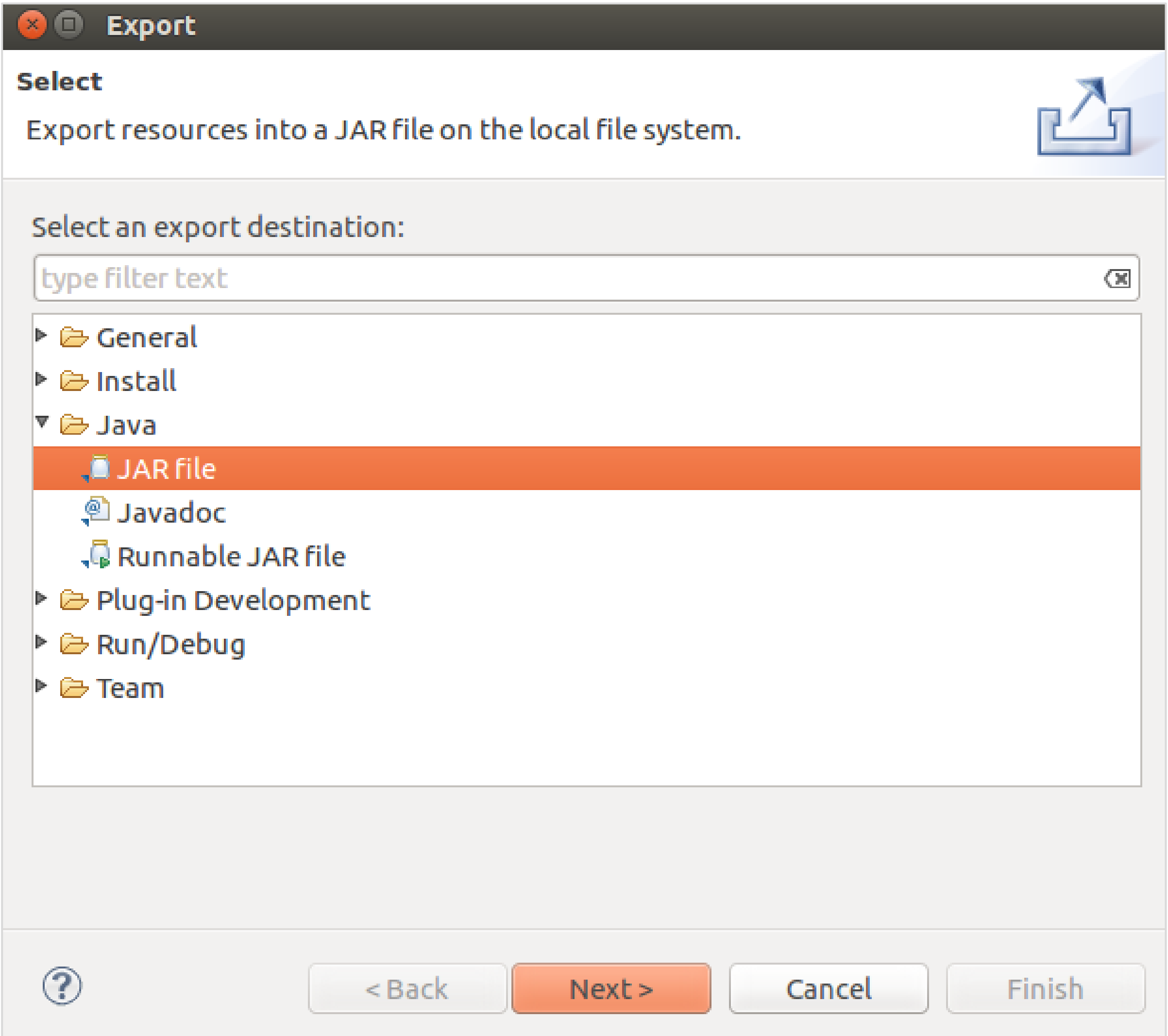


You will see that the errors are no more there.

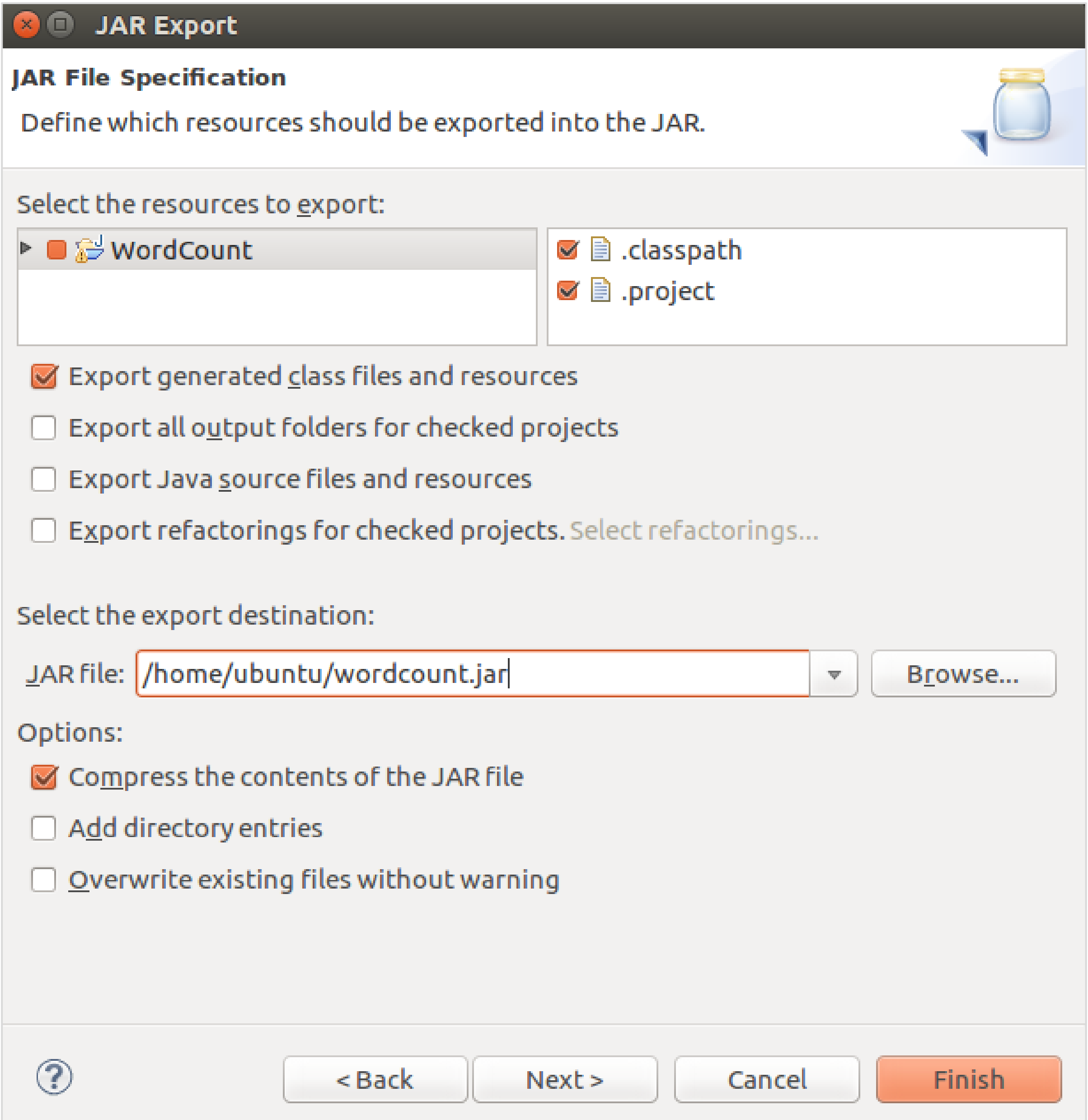
Now, let us export this application. Right-click on the project and click on Export.



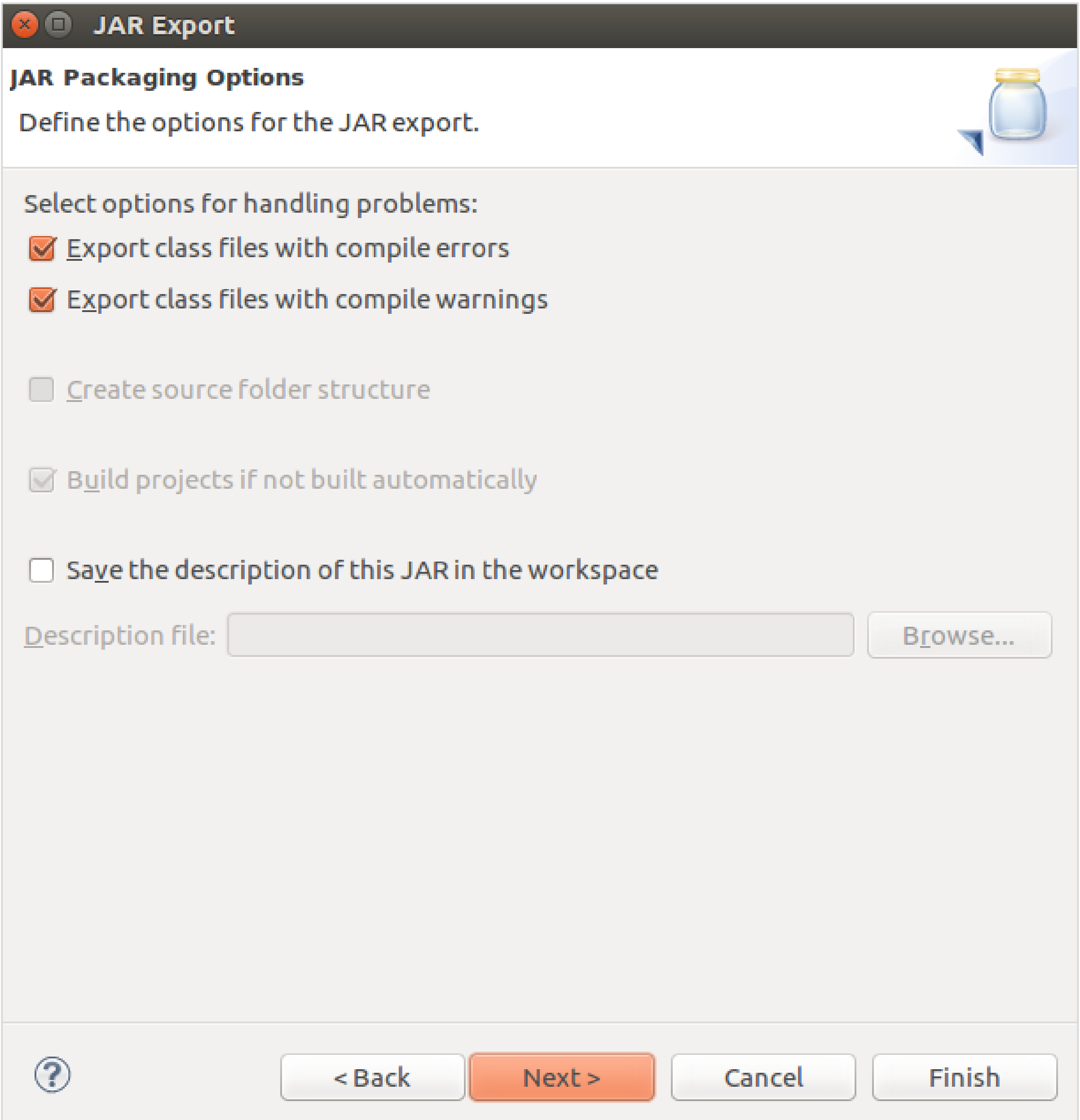
Select JAR file and click Next>



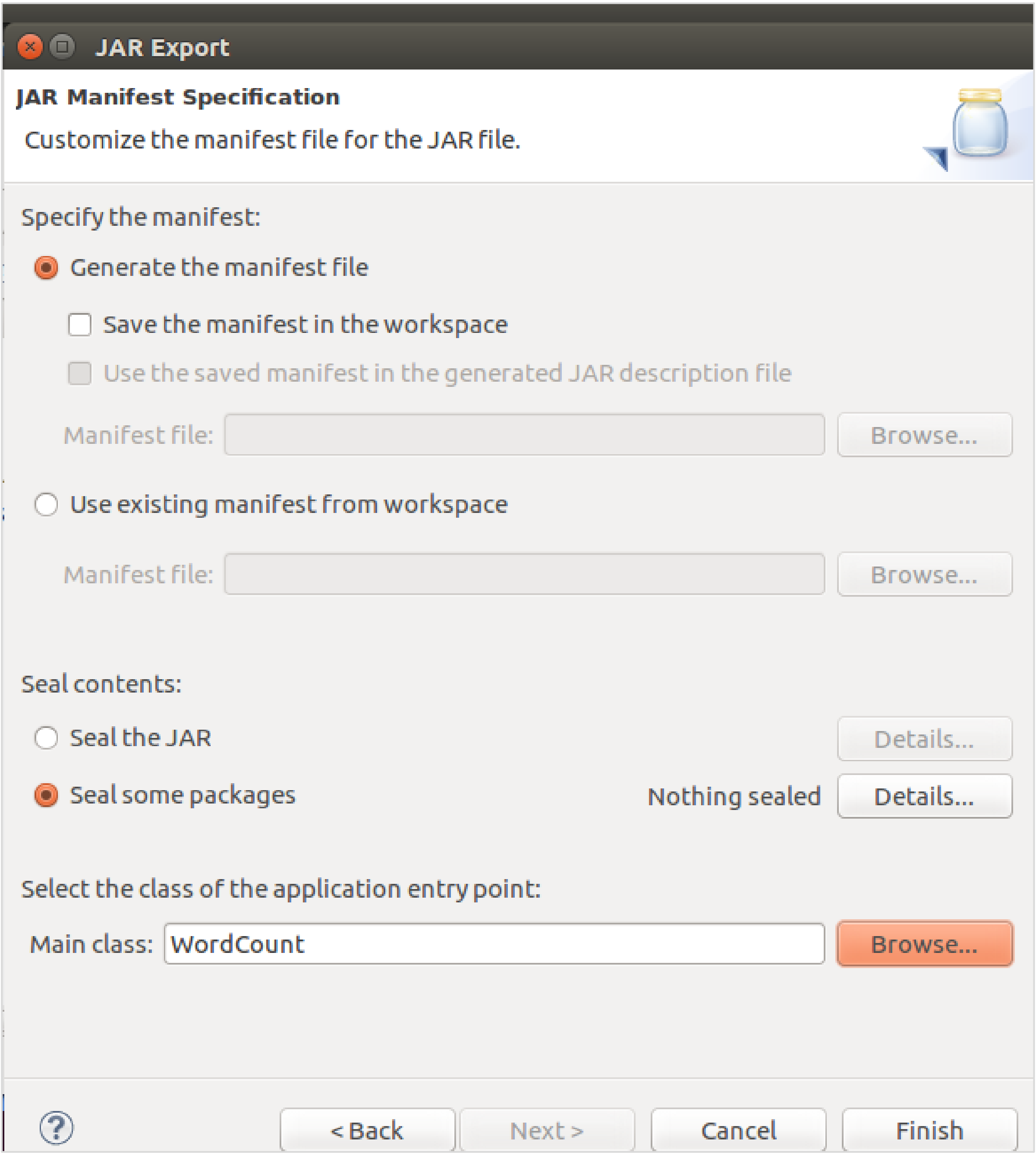
Give a destination path and click on Next>



Click on Next>



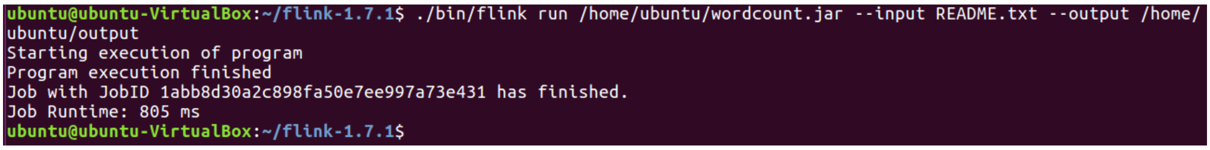
Click on Browse, select the main class (WordCount) and click Finish.



Note: Click OK, in case you get any warning.

Run the below command. It will further run the Flink application you just created.

```
./bin/flink run /home/ubuntu/wordcount.jar --input README.txt --output
/home/ubuntu/output
```



```
ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$ ./bin/flink run /home/ubuntu/wordcount.jar --input README.txt --output /home/
ubuntu/output
Starting execution of program
Program execution finished
Job with JobID 1abb8d30a2c898fa50e7ee997a73e431 has finished.
Job Runtime: 805 ms
ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$
```

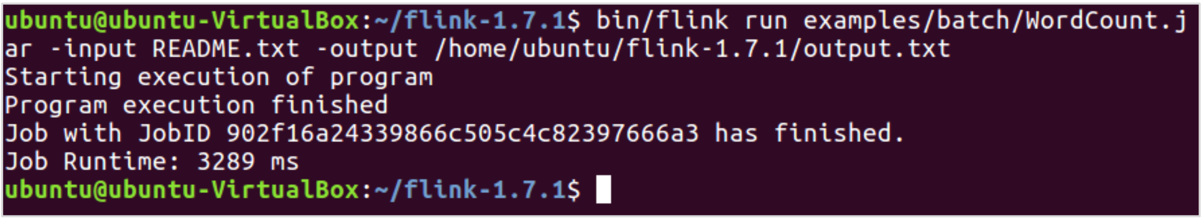
10. Apache Flink — Running a Flink Program

In this chapter, we will learn how to run a Flink program.

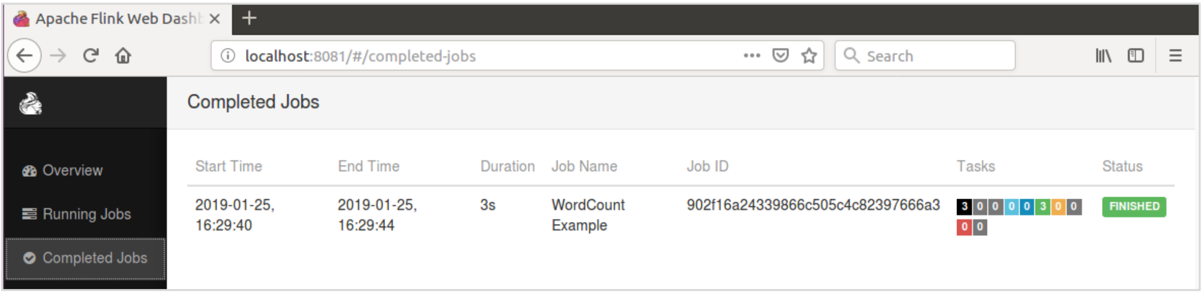
Let us run the Flink wordcount example on a Flink cluster.

Go to Flink's home directory and run the below command in the terminal.

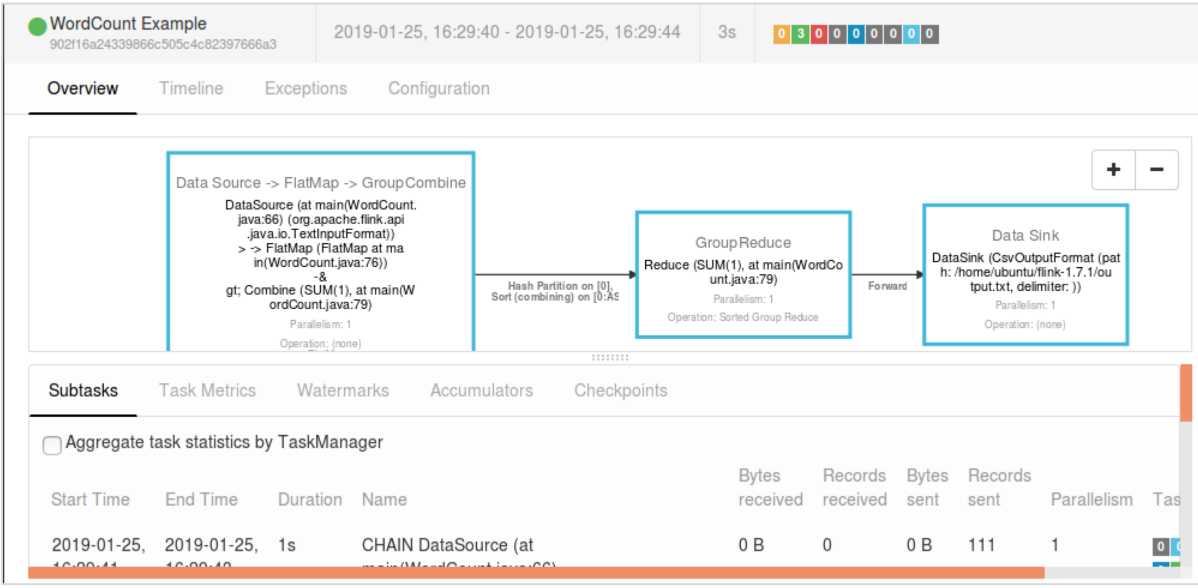
```
bin/flink run examples/batch/WordCount.jar -input README.txt -output
/home/ubuntu/flink-1.7.1/output.txt
```



Go to Flink dashboard, you will be able to see a completed job with its details.

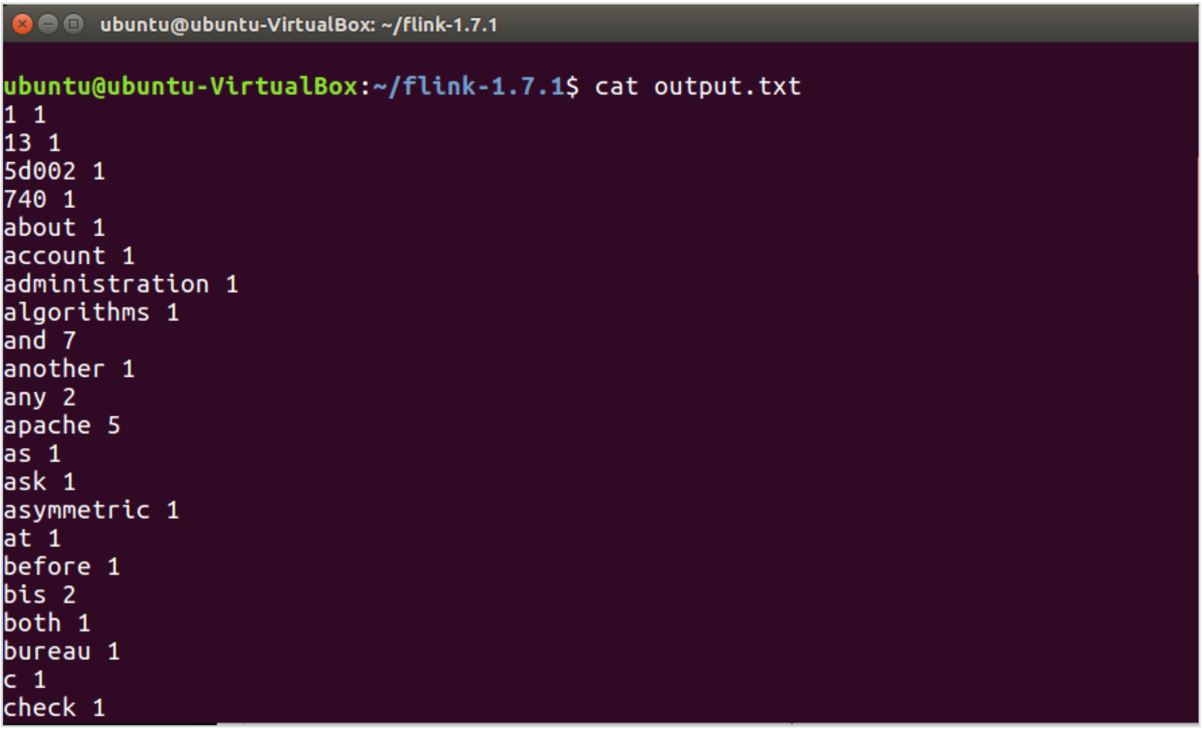


If you click on Completed Jobs, you will get detailed overview of the jobs.



To check the output of wordcount program, run the below command in the terminal.

```
cat output.txt
```



11. Apache Flink — Libraries

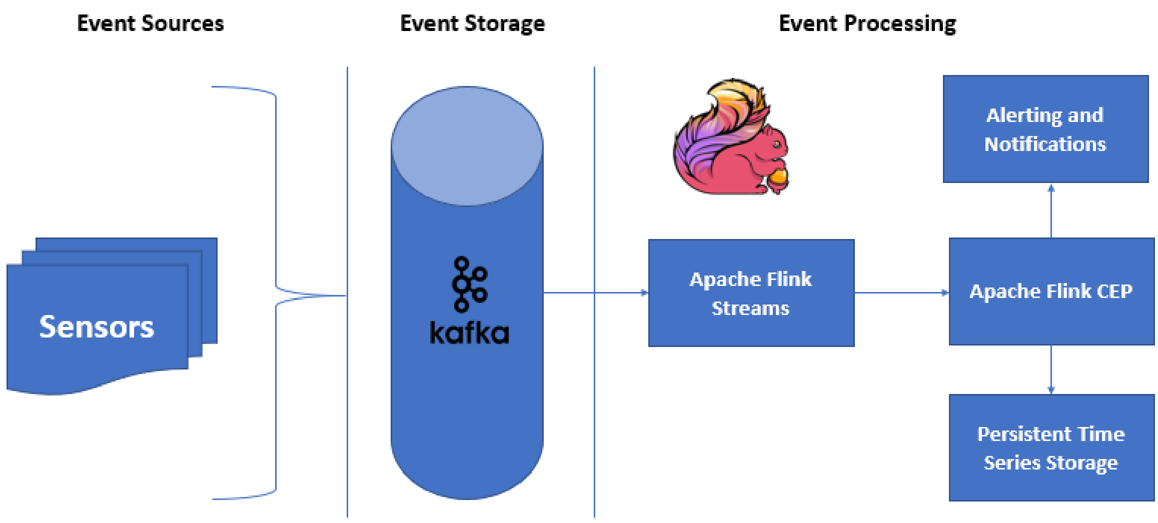
In this chapter, we will learn about the different libraries of Apache Flink.

Complex Event Processing (CEP)

FlinkCEP is an API in Apache Flink, which analyses event patterns on continuous streaming data. These events are near real time, which have high throughput and low latency. This API is used mostly on Sensor data, which come in real-time and are very complex to process.

CEP analyses the pattern of the input stream and gives the result very soon. It has the ability to provide real-time notifications and alerts in case the event pattern is complex. FlinkCEP can connect to different kind of input sources and analyse patterns in them.

This how a sample architecture with CEP looks like:



Sensor data will be coming in from different sources, Kafka will act as a distributed messaging framework, which will distribute the streams to Apache Flink, and FlinkCEP will analyse the complex event patterns.

You can write programs in Apache Flink for complex event processing using Pattern API. It allows you to decide the event patterns to detect from the continuous stream data. Below are some of the most commonly used CEP patterns:

Begin

It is used to define the starting state. The following program shows how it is defined in a Flink program:

```
Pattern<Event, ?> next = start.next("next");
```

Where

It is used to define a filter condition in the current state.

```
patternState.where(new FilterFunction <Event>() {
@Override
public boolean filter(Event value) throws Exception {
return ... // some condition
}
});
```

Next

It is used to append a new pattern state and the matching event needed to pass the previous pattern.

```
Pattern<Event, ?> next = start.next("next");
```

FollowedBy

It is used to append a new pattern state but here other events can occur b/w two matching events.

```
Pattern<Event, ?> followedBy = start.followedBy("next");
```

Gelly

Apache Flink's Graph API is Gelly. Gelly is used to perform graph analysis on Flink applications using a set of methods and utilities. You can analyse huge graphs using Apache Flink API in a distributed fashion with Gelly. There are other graph libraries also like Apache Giraph for the same purpose, but since Gelly is used on top of Apache Flink, it uses single API. This is very helpful from development and operation point of view.

Let us run an example using Apache Flink API – Gelly.

Firstly, you need to copy 2 Gelly jar files from opt directory of Apache Flink to its lib directory. Then run flink-gelly-examples jar.

```
cp opt/flink-gelly* lib/
./bin/flink run examples/gelly/flink-gelly-examples_*.jar
```

```
ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$ cp opt/flink-gelly* lib/
ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$ ./bin/flink run examples/gelly/flink-gelly-examples_*.jar
Starting execution of program

Select an algorithm to view usage: flink run examples/flink-gelly-examples_<version>.jar --algorithm <algorithm>

Available algorithms:
  AdamicAdar                  similarity score weighted by centerpoint degree
  ClusteringCoefficient       measure the connectedness of vertex neighborhoods
  ConnectedComponents         ConnectedComponents
  EdgeList                    the edge list
  GraphMetrics                compute vertex and edge metrics
  HITS                        score vertices as hubs and authorities
  JaccardIndex                similarity score as fraction of common neighbors
  PageRank                    score vertices by the number and quality of incoming links
  TriangleListing             list triangles

ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$
```

Let us now run the PageRank example.

PageRank computes a per-vertex score, which is the sum of PageRank scores transmitted over in-edges. Each vertex's score is divided evenly among out-edges. High-scoring vertices are linked to by other high-scoring vertices.

The result contains the vertex ID and the PageRank score.

```
usage: flink run examples/flink-gelly-examples_<version>.jar --algorithm
PageRank [algorithm options] --input <input> [input options] --output <output>
[output options]

./bin/flink run examples/gelly/flink-gelly-examples_*.jar --algorithm PageRank
--input CycleGraph --vertex_count 2 --output Print
```

```
ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$ ./bin/flink run examples/gelly/flink-gelly-examples_*.jar --algorithm PageRan
k --input CycleGraph --vertex_count 2 --output Print
Starting execution of program

Vertex ID: 0, PageRank score: 0.5
Vertex ID: 1, PageRank score: 0.5

Program execution finished
Job with JobID 44eb49deb42b81ab597a67ac1ef1149e has finished.
Job Runtime: 5283 ms
Accumulator Results:
- 8e99a3af13ef75eaf17e3c969d799e73-collect (java.util.ArrayList) [2 elements]

ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$
```

12. Apache Flink — Machine Learning

Apache Flink's Machine Learning library is called FlinkML. Since usage of machine learning has been increasing exponentially over the last 5 years, Flink community decided to add this machine learning APO also in its ecosystem. The list of contributors and algorithms are increasing in FlinkML. This API is not a part of binary distribution yet.

Here is an example of linear regression using FlinkML:

```
// LabeledVector is a feature vector with a label (class or real value)
val trainingData: DataSet[LabeledVector] = ...
val testingData: DataSet[Vector] = ...


// Alternatively, a Splitter is used to break up a DataSet into training and
testing data.
val dataSet: DataSet[LabeledVector] = ...
val trainTestData: DataSet[TrainTestDataSet] = Splitter.trainTestSplit(dataSet)
val trainingData: DataSet[LabeledVector] = trainTestData.training
val testingData: DataSet[Vector] = trainTestData.testing.map(lv => lv.vector)


val mlr = MultipleLinearRegression()
  .setStepsize(1.0)
  .setIterations(100)
  .setConvergenceThreshold(0.001)


mlr.fit(trainingData)


// The fitted model can now be used to make predictions
val predictions: DataSet[LabeledVector] = mlr.predict(testingData)
```

Inside **flink-1.7.1/examples/batch/** path, you will find KMeans.jar file. Let us run this sample FlinkML example.

This example program is run using the default point and the centroid data set.

```
./bin/flink run examples/batch/KMeans.jar --output Print
```

```
ubuntu@ubuntu-VirtualBox:~/flink-1.7.1$ ./bin/flink run examples/batch/KMeans.jar --output Print
Starting execution of program
Executing K-Means example with default point data set.
Use --points to specify file input.
Executing K-Means example with default centroid data set.
Use --centroids to specify file input.
Program execution finished
Job with JobID 6619346b85610635c21e96b16d4f7dfc has finished.
Job Runtime: 2373 ms
```

13. Apache Flink — Use Cases

In this chapter, we will understand a few test cases in Apache Flink.

Apache Flink – Bouygues Telecom

Bouygues Telecom is one of the largest telecom organization in France. It has 11+ million mobile subscribers and 2.5+ million fixed customers. Bouygues heard about Apache Flink for the first time in a Hadoop Group Meeting held at Paris. Since then they have been using Flink for multiple use-cases. They have been processing billions of messages in a day in real-time through Apache Flink.

This is what Bouygues has to say about Apache Flink: "W*e ended up with Flink because the system supports true streaming - both at the API and at the runtime level, giving us the programmability and low latency that we were looking for. In addition, we were able to get our system up and running with Flink in a fraction of the time compared to other solutions, which resulted in more available developer resources for expanding the business logic in the system."*

At Bouygues, customer experience is the highest priority. They analyse data in real-time so that they can give below insights to their engineers:

- Real-Time Customer Experience over their network
- What is happening globally on the network
- Network evaluations and operations

They created a system called LUX (Logged User Experience) which processed massive log data from network equipment with internal data reference to give quality of experience indicators which will log their customer experience and build an alarming functionality to detect any failure in consumption of data within 60 seconds.

To achieve this, they needed a framework which can take massive data in real-time, is easy to set up and provides rich set of APIs for processing the streamed data. Apache Flink was a perfect fit for Bouygues Telecom.

Apache Flink – Alibaba

Alibaba is the largest ecommerce retail company in the world with 394 billion $ revenue in 2015. Alibaba search is the entry point to all the customers, which shows all the search and recommends accordingly.

Alibaba uses Apache Flink in its search engine to show results in real-time with highest accuracy and relevancy for each user.

Alibaba was looking for a framework, which was:

- Very Agile in maintaining one codebase for their entire search infrastructure process.
- Provides low latency for the availability changes in the products on the website.

- Consistent and cost effective.

Apache Flink qualified for all the above requirements. They need a framework, which has a single processing engine and can process both batch and stream data with same engine and that is what Apache Flink does.

They also use Blink, a forked version for Flink to meet some unique requirements for their search. They are also using Apache Flink's Table API with few improvements for their search.

This is what Alibaba had to say about apache Flink: "*Looking back, it was no doubt a huge year for Blink and Flink at Alibaba. No one thought that we would make this much progress in a year, and we are very grateful to all the people who helped us in the community. Flink is proven to work at the very large scale. We are more committed than ever to continue our work with the community to move Flink forward!*"

14. Apache Flink — Flink vs Spark vs Hadoop

Here is a comprehensive table, which shows the comparison between three most popular big data frameworks: Apache Flink, Apache Spark and Apache Hadoop.

	Apache Hadoop	**Apache Spark**	**Apache Flink**
Year of Origin	2005	2009	2009
Place of Origin	MapReduce (Google) Hadoop (Yahoo)	University of California, Berkeley	Technical University of Berlin
Data Processing Engine	Batch	Batch	Stream
Processing Speed	Slower than Spark and Flink	100x Faster than Hadoop	Faster than spark
Programming Languages	Java, C, C++, Ruby, Groovy, Perl, Python	Java, Scala, python and R	Java and Scala
Programming Model	MapReduce	Resilient distributed Datasets (RDD)	Cyclic dataflows
Data Transfer	Batch	Batch	Pipelined and Batch
Memory Management	Disk Based	JVM Managed	Active Managed
Latency	Low	Medium	Low
Throughput	Medium	High	High
Optimization	Manual	Manual	Automatic
API	Low-level	High-level	High-level
Streaming Support	NA	Spark Streaming	Flink Streaming
SQL Support	Hive, Impala	SparkSQL	Table API and SQL
Graph Support	NA	GraphX	Gelly
Machine Learning Support	NA	SparkML	FlinkML

15. Apache Flink — Conclusion

The comparison table that we saw in the previous chapter concludes the pointers pretty much. Apache Flink is the most suited framework for real-time processing and use cases. Its single engine system is unique which can process both batch and streaming data with different APIs like Dataset and DataStream.

It does not mean Hadoop and Spark are out of the game, the selection of the most suited big data framework always depends and vary from use case to use case. There can be several use cases where a combination of Hadoop and Flink or Spark and Flink might be suited.

Nevertheless, Flink is the best framework for real time processing currently. The growth of Apache Flink has been amazing and the number of contributors to its community is growing day by day.

Happy Flinking!