

Lab 5. Building Spark Streaming Applications with Kafka



In this lab, we will cover Apache Spark, which is distributed in memory processing engines and then we will walk through Spark Streaming concepts and how we can integrate Apache Kafka with Spark.

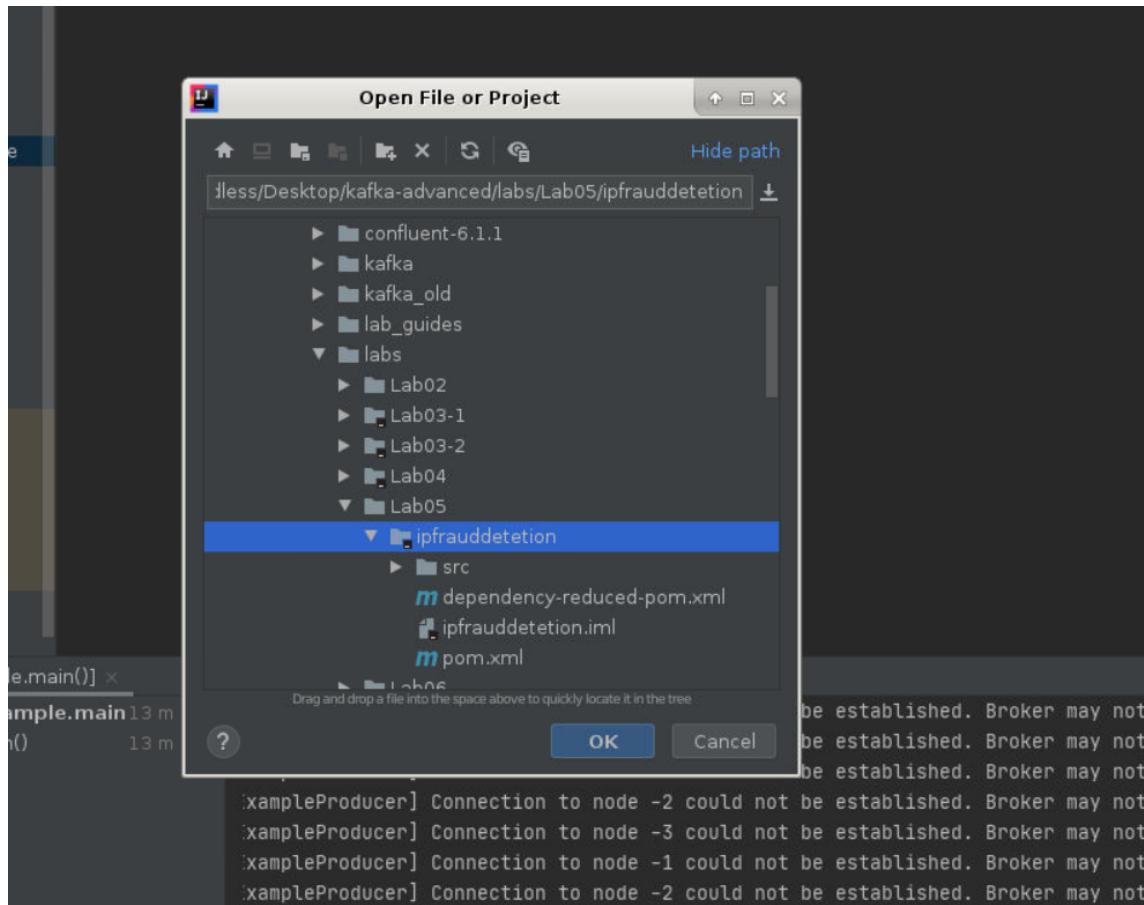
In short, we will cover the following topics:

- Introduction to Spark
 - Receiver-based approach (Spark-Kafka integration)
 - Direct approach (Spark-Kafka integration)
 - Use case (Log processing)

Lab Solution

Complete solution for this lab is available in the following directory:

~/kafka-advanced/labs/Lab05/ipfrauddetection



Spark Streaming

We have two approaches to integrate Kafka with Spark and we will go into detail on each:

- Receiver-based approach
 - Direct approach

The receiver-based approach is the older way of doing integration. Direct API integration provides lots of advantages over the receiver-based approach.

Apache Spark Installation

Make sure you have compatible java installed on your machine. You can verify it by typing command:

```
java -version
```

Apache spark setup has been downloaded already on the following path and added to \$PATH variable:

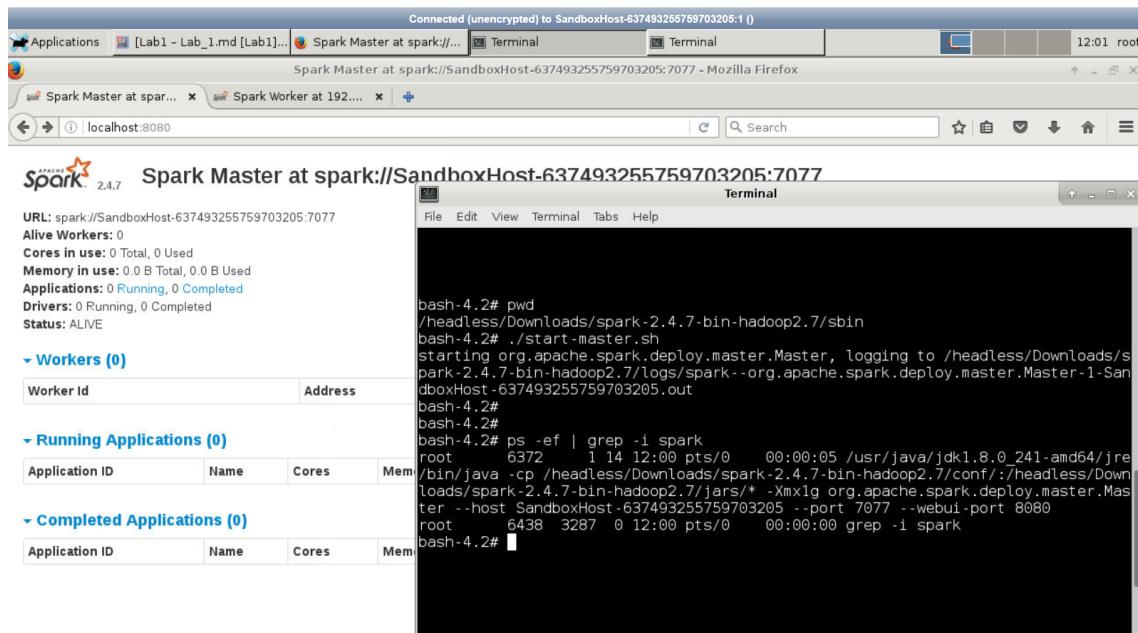
```
/headless/Downloads/spark-2.4.7-bin-hadoop2.7/
```

Start Master

```
cd /headless/Downloads/spark-2.4.7-bin-hadoop2.7/  
  
.sbin/start-master.sh
```

Inspect the response:

```
starting org.apache.spark.deploy.master.Master, logging to /headless/Downloads/spark-  
2.4.7-bin-hadoop2.7/logs/spark--org.apache.spark.deploy.master.Master-1-SandboxHost-  
637493255759703205.out
```



Once started, the master will print out a spark://HOST:PORT URL for itself, which you can use to connect workers to it, or pass as the “master” argument to SparkContext. You can also find this URL on the master’s web UI, which is <http://localhost:8080> by default.

Similarly, you can start one or more workers and connect them to the master via:

Start Worker

Start Worker and register the worker with master

Open <http://localhost:8080/> in browser and copy the master url

The screenshot shows the Spark Master UI at <http://localhost:8080>. The main content area includes:

- URL:** spark://SandboxHost-637493255759703205:7077
- 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

Below these are sections for **Workers (0)**, **Running Applications (0)**, and **Completed Applications (0)**, each with a table header.

now start the worker and register it with master using following command (Update `spark://hostname:7077` with the master hostname first):

```
./sbin/start-slave.sh spark://hostname:7077
```

The screenshot shows the Spark Worker UI at <http://localhost:8080>. The main content area includes:

- ID:** worker-20210219120249-192.168.0.209-33781
- Master URL:** spark://SandboxHost-637493255759703205:7077
- Cores:** 3 (0 Used)
- Memory:** 6.9 GB (0.0 B Used)

A link [Back to Master](#) is present. Below these are sections for **Running Executors (0)**, each with a table header, and a **Terminal** window showing the command execution:

```
bash-4.2# pwd  
/headless/Downloads/spark-2.4.7-bin-hadoop2.7/sbin  
bash-4.2# ./start-slave.sh spark://SandboxHost-637493255759703205:7077  
starting org.apache.spark.deploy.worker.Worker, logging to /headless/Downloads/spark-2.4.7-bin-hadoop2.7/logs/spark--org.apache.spark.deploy.worker.Worker-1-SandboxHost-637493255759703205.out  
bash-4.2#
```

Worker webUI: <http://localhost:8081>

The screenshot shows a Linux desktop environment with a terminal window open. In the terminal, the command `spark-shell --master spark://localhost:7077` is run. The output shows the master URL and the number of cores and memory available.

```
[user@host ~]$ spark-shell --master spark://localhost:7077
Connected (unencrypted) to SandboxHost-637493255759703205:1 ()
Apache Spark 2.4.7
URL: spark://SandboxHost-637493255759703205:7077
Alive Workers: 1
Cores in use: 3 Total, 0 Used
Memory in use: 6.9 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE
```

The Firefox browser window displays the Apache Spark Master UI. It includes:

- A summary section with basic cluster statistics.
- A table titled "Workers (1)" showing one worker entry:

Worker Id	Address	State	Cores	Memory
worker-20210219120249-192.168.0.209-33781	192.168.0.209:33781	ALIVE	3 (0 Used)	6.9 GB (0.0 B Used)

- A table titled "Running Applications (0)" showing no applications.
- A table titled "Completed Applications (0)" showing no completed applications.

Once you have started a worker, look at the master's web UI (<http://localhost:8080> by default). You should see the new node listed there, along with its number of CPUs and memory (minus one gigabyte left for the OS).

Note: `spark-shell` and `spark-submit` have been added to `PATH` already and also present in following directory:

```
/headless/Downloads/spark-2.4.7-bin-hadoop2.7/
```

Spark Shell

Your cluster on single node is ready now, you can test it using running command **spark-shell**

```
[user@hostname ~]$ spark-shell --master spark://hostname:7077
```

Reload the master webui. You will get one running application:

The screenshot shows a Linux desktop environment with a terminal window and a Firefox browser. The terminal window displays Spark logs and a Scala shell session. The Firefox browser shows the Spark Master UI at `spark://SandboxHost-637493255759703205:7077`. The UI provides information about workers, applications, and completed applications.

```

Connected (unencrypted) to SandboxHost-637493255759703205:1
Spark Master at spark://... Terminal [Terminal] 12:06 root
Mozilla Firefox
Spark Master at spar... Spark Worker at 192.... localhost:8080
File Edit View Terminal Tabs Help
21/02/19 12:05:51 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
21/02/19 12:05:52 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.0.209:4040
Spark context available as 'sc' (master = spark://SandboxHost-637493255759703205:7077, app id = app-20210219120603-0000).
Spark session available as 'spark'.
Welcome to
    \_\_/\_\_/\_\_/\_\_/\_\_
    \_\_/\_\_/\_\_/\_\_/\_\_
version 2.4.7
Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_241)
Type in expressions to have them evaluated.
Type :help for more information.
scala>

```

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20210219120603-0000	(kill)	Spark shell					

You can exit the spark shell using typing `:q` then enter

Spark Submit

Now, submit example application using spark-submit. Replace `spark://hostname:7077` with hostname of master node first.

```
spark-submit --class org.apache.spark.examples.SparkPi --master spark://hostname:7077
/headless/Downloads/spark-2.4.7-bin-hadoop2.7/examples/jars/spark-examples_2.11-
2.4.7.jar
```

If this example execute successfully, your spark installation is fine. You can see the results in console log:

```
2019-09-12 13:53:27 INFO DAGScheduler:54 - Job 0 finished: reduce at SparkPi.scala:38,
took 0.615754 sPi is roughly 3.141557082785412 2019-09-12 13:53:27 INFO
AbstractConnector:318 - Stopped Spark@6914bc2c{HTTP/1.1, [http/1.1]}{0.0.0.0:4040}
```

The screenshot shows a terminal window with the following log output:

```

Connected (unencrypted) to SandboxHost-637493255759703205.1 ()
[Lab1 - Lab_1.md [Lab1]...]
Spark Master at spark://... Terminal
Spark Master at spark://SandboxHost-637493255759703205:7077 - Mozilla Firefox
Spark Master at spark... x Spark Worker at 192.... x + Terminal
localhost:8081 File Edit View Terminal Tabs Help
Apache Spark 2.4.7
Spark Master at spark://SandboxHost-637493255759703205:7077 /headless/Downloads/spark-2.4.7-bin-hadoop2.7/examples/jars/spark-examples_2.11-2.4.7.jar
USER_ID: 0, GROUP_ID: 0
bash-4.2# spark-submit --class org.apache.spark.examples.SparkPi --master spark://SandboxHost-637493255759703205:7077
21/02/19 12:08:26 WARN Utils: Your hostname, SandboxHost-637493255759703205 resolves to a loopback address: 127.0.0.1; using 192.168.0.209 instead (on interface eth0)
21/02/19 12:08:26 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
21/02/19 12:08:27 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
21/02/19 12:08:28 INFO SparkContext: Running Spark version 2.4.7
21/02/19 12:08:28 INFO SparkContext: Submitted application: Spark Pi
21/02/19 12:08:28 INFO SecurityManager: Changing view acls to: root
21/02/19 12:08:28 INFO SecurityManager: Changing modify acls to: root
21/02/19 12:08:28 INFO SecurityManager: Changing view acls groups to:
21/02/19 12:08:28 INFO SecurityManager: authentication disabled
; ui acls disabled; users with view permissions: Set(root); groups with view permissions: Set()
; users with modify permissions: Set(root); groups with modify permissions: Set()
21/02/19 12:08:28 INFO Utils: Successfully started service 'sparkDriver' on port 42875.

```

Workers (1)

Worker ID
worker-20210219120249-192.168.0.209-33781

Running Applications (0)

Application ID	Name	Cores	Memory	Submitted Time	User	State	Duration
app-20210219120829-0001	Spark Pi	3	1024.0 MB	2021/02/19 12:08:29	root	FINISHED	6 s
app-20210219120603-0000	Spark shell	3	1024.0 MB	2021/02/19 12:06:03	root	FINISHED	41 s

Completed Applications (2)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20210219120829-0001	Spark Pi	3	1024.0 MB	2021/02/19 12:08:29	root	FINISHED	6 s
app-20210219120603-0000	Spark shell	3	1024.0 MB	2021/02/19 12:06:03	root	FINISHED	41 s

Task

- Run above spark-submit command but assign '2g' memory. You will get output after running spark-submit command as shown in the screenshot below:

The screenshot shows a terminal window with the following log output:

```

Connected (unencrypted) to SandboxHost-637493255759703205.1 ()
[Lab1 - Lab_1.md [Lab1]...]
Spark Worker at 192.168.0.209:33781 - Mozilla Firefox
Spark Master at spar... x Spark Worker at 192.... x + Terminal
localhost:8081 File Edit View Terminal Tabs Help
Apache Spark 2.4.7
Spark Worker at 192.168.0.209:33781
ID: worker-20210219120249-192.168.0.209-33781
Master URL: spark://SandboxHost-637493255759703205:7077
Cores: 3 (0 Used)
Memory: 6.9 GB (0.0 B Used)
Back to Master

```

Running Executors (0)

ExecutorID	Cores	State	Memory	Job Details	Logs

Finished Executors (3)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	3	KILLED	1024.0 MB	ID: app-20210219120603-0000 Name: Spark shell User: root	stdout stderr
0	3	KILLED	1024.0 MB	ID: app-20210219120829-0001 Name: Spark Pi User: root	stdout stderr
0	3	KILLED	2.0 GB	ID: app-20210219121014-0002 Name: Spark Pi User: root	stdout stderr

Hint: Add following parameter while running spark-submit `--executor-memory 2g`.

Java example for receiver-based integration

Let us take an example to be sure:

```

import org.apache.Spark.SparkConf;
import org.apache.Spark.api.java.function.FlatMapFunction;
import org.apache.Spark.api.java.function.Function;
import org.apache.Spark.api.java.function.Function2;
import org.apache.Spark.api.java.function.PairFunction;
import org.apache.Spark.streaming.Duration;
import org.apache.Spark.streaming.api.java.JavaDStream;
import org.apache.Spark.streaming.api.java.JavaPairDStream;
import org.apache.Spark.streaming.api.java.JavaPairReceiverInputDStream;
import org.apache.Spark.streaming.api.java.JavaStreamingContext;
import org.apache.Spark.streaming.kafka.KafkaUtils;
import scala.Tuple2;

import java.util.Arrays;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.regex.Pattern;

public class KafkaWordCountJava {
    private static final Pattern WORD_DELIMETER = Pattern.compile(" ");

    public static void main(String[] args) throws Exception {
        String zkQuorum = "localhost:2181";
        String groupName = "stream";
        int numThreads = 3;
        String topicsName = "test";
        SparkConf SparkConf = new
SparkConf().setAppName("WordCountKafkaStream").setMaster("local[2]");

        JavaStreamingContext javaStreamingContext = new
JavaStreamingContext(SparkConf, new Duration(5000));

        Map<String, Integer> topicToBeUsedBySpark = new HashMap<>();
        String[] topics = topicsName.split(",");
        for (String topic : topics) {
            topicToBeUsedBySpark.put(topic, numThreads);
        }

        JavaPairReceiverInputDStream<String, String> streamMessages =
            KafkaUtils.createStream(javaStreamingContext, zkQuorum, groupName,
topicToBeUsedBySpark);

        JavaDStream<String> lines = streamMessages.map(new Function<Tuple2<String,
String>, String>() {
            @Override
            public String call(Tuple2<String, String> tuple2) {
                return tuple2._2();
            }
        });
    }
}

```

```

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

JavaPairDStream<String, Integer> wordCounts = words.mapToPair(
    new PairFunction<String, String, Integer>() {
        @Override
        public Tuple2<String, Integer> call(String s) {
            return new Tuple2<>(s, 1);
        }
    }).reduceByKey(new Function2<Integer, Integer, Integer>() {
        @Override
        public Integer call(Integer i1, Integer i2) {
            return i1 + i2;
        }
    });
wordCounts.print();
javaStreamingContext.start();
javaStreamingContext.awaitTermination();
}
}

```

Direct approach

In receiver-based approach, we saw issues of data loss, costing less throughput using write-ahead logs and difficulty in achieving exactly one semantic of data processing. To overcome all these problems, Spark introduced the direct stream approach of integrating Spark with Kafka.

Java example for direct approach

Again, let us take a Java example:

```

import java.util.HashMap;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Map;
import java.util.Set;
import java.util.regex.Pattern;

import scala.Tuple2;

import kafka.serializer.StringDecoder;

import org.apache.SparkConf;
import org.apache.spark.streaming.api.java.*;
import org.apache.spark.streaming.kafka.KafkaUtils;
import org.apache.spark.streaming.Durations;

```

```

public class JavaDirectKafkaWordCount {
    private static final Pattern SPACE = Pattern.compile(" ");

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

        String brokers = "localhost:9092";
        String topics = "test";

        SparkConf sparkConf = new
sparkConf().setAppName("DirectKafkaWordCount").setMaster("local[2]");
        JavaStreamingContext javaStreamingContext = new
JavaStreamingContext(sparkConf, Durations.seconds(2));

        Set<String> topicsSet = new HashSet<>(Arrays.asList(topics.split(",")));
        Map<String, String> kafkaConfiguration = new HashMap<>();
        kafkaConfiguration.put("metadata.broker.list", brokers);

        JavaPairInputDStream<String, String> messages = KafkaUtils.createDirectStream(
            javaStreamingContext,
            String.class,
            String.class,
            StringDecoder.class,
            StringDecoder.class,
            kafkaConfiguration,
            topicsSet
        );

        JavaDStream<String> lines = messages.map(Tuple2::_2);

        JavaDStream<String> words = lines.flatMap(x ->
Arrays.asList(SPACE.split(x)).iterator());

        JavaPairDStream<String, Integer> wordCounts = words.mapToPair(s -> new
Tuple2<>(s, 1))
            .reduceByKey((i1, i2) -> i1 + i2);

        wordCounts.print();

        javaStreamingContext.start();
        javaStreamingContext.awaitTermination();
    }
}

```

Use case log processing - fraud IP detection

This section will cover a small use case which uses Kafka and Spark Streaming to detect a fraud IP, and the number of times the IP tried to hit the server. We will cover the use case in the following:

- **Producer:** We will use Kafka Producer API, which will read a log file and publish records to Kafka topic. However, in a real case, we may use Flume or producer application, which directly takes a log record on a real-time basis and publish to Kafka topic.

- **Fraud IPs list:** We will maintain a list of predefined fraud IP range which can be used to identify fraud IPs. For this application we are using in memory IP list which can be replaced by fast key based lookup, such as HBase.
- **Spark Streaming:** Spark Streaming application will read records from Kafka topic and will detect IPs and domains which are suspicious.

Maven

[Maven] is a build and project management tool and we will be building this project using Maven. I recommend using Eclipse or IntelliJ for creating projects. Add the following dependencies and plugins to your `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://Maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://Maven.apache.org/POM/4.0.0
http://Maven.apache.org/xsd/Maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.fenago</groupId>
    <artifactId>ip-fraud-detetion</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>kafka-producer</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.Spark/Spark-streaming-
kafka_2.10 -->
        <dependency>
            <groupId>org.apache.Spark</groupId>
            <artifactId>Spark-streaming-kafka_2.10</artifactId>
            <version>1.6.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.2</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.Spark/Spark-core_2.10 -->
        <dependency>
            <groupId>org.apache.Spark</groupId>
            <artifactId>Spark-core_2.10</artifactId>
            <version>2.0.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

```

```

        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.Spark/Spark-streaming_2.10
-->
<dependency>
    <groupId>org.apache.Spark</groupId>
    <artifactId>Spark-streaming_2.10</artifactId>
    <version>2.0.0</version>
    <scope>provided</scope>

</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>0.10.0.0</version>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.Maven.plugins</groupId>
            <artifactId>Maven-shade-plugin</artifactId>
            <version>2.4.2</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <filters>
                            <filter>
                                <artifact>junit:junit</artifact>
                                <includes>
                                    <include>junit/framework/**</include>
                                    <include>org/junit/**</include>
                                </includes>
                                <excludes>
                                    <exclude>org/junit/experimental/**</exclude>
                                    <exclude>org/junit/runners/**</exclude>
                                </excludes>
                            </filter>
                            <filter>
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

```

                </filter>
            </filters>
            <transformers>
                <transformer

implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>

                <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

<mainClass>com.fenago.streaming.FraudDetectionApp</mainClass>
                </transformer>
            </transformers>
        </configuration>
    </execution>
</executions>
</plugin>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-Maven-plugin</artifactId>
    <version>1.2.1</version>
    <executions>
        <execution>
            <goals>
                <goal>exec</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <includeProjectDependencies>true</includeProjectDependencies>
        <includePluginDependencies>false</includePluginDependencies>
        <executable>java</executable>
        <classpathScope>compile</classpathScope>
        <mainClass>com.fenago.streaming.FraudDetectionApp</mainClass>
    </configuration>
</plugin>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>Maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
</plugins>
</build>
</project>
```

Producer

You can use IntelliJ or Eclipse to build a producer application. This producer reads a log file taken from an Apache project which contains detailed records like:

```
64.242.88.10 - - [08/Mar/2004:07:54:30 -0800] "GET  
/twiki/bin/edit/Main/Unknown_local_recipient_reject_code?  
topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12846
```

You can have just one record in the test file and the producer will produce records by generating random IPs and replace it with existing. So, we will have millions of distinct records with unique IP addresses.

Record columns are separated by space delimiters, which we change to commas in producer. The first column represents the IP address or the domain name which will be used to detect whether the request was from a fraud client. The following is the Java Kafka producer which remembers logs.

Property reader

We preferred to use a property file for some important values such as topic, Kafka broker URL, and so on. If you want to read more values from the property file, then feel free to change it in the code. `streaming.properties` file:

```
topic=ipTest2  
broker.list=localhost:9092  
appname=IpFraud  
group.id=Stream
```

The following is an example of the property reader:

```
import java.io.FileNotFoundException;  
import java.io.IOException;  
import java.io.InputStream;  
import java.util.Properties;  
  
public class PropertyReader {  
  
    private Properties prop = null;  
  
    public PropertyReader() {  
  
        InputStream is = null;  
        try {  
            this.prop = new Properties();  
            is = this.getClass().getResourceAsStream("/streaming.properties");  
            prop.load(is);  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public String getPropertyValue(String key) {  
        return this.prop.getProperty(key);  
    }  
}
```

```
    }
}
```

Producer code

Do not run producer code yet, we will build and run Spark Streaming application first

A producer application is designed to be like a real-time log producer where the producer runs every three seconds and produces a new record with random IP addresses. You can add a few records in the `IP_LOG.log` file and then the producer will take care of producing millions of unique records from those three records.

We have also enabled auto creation of topics so you need not create topic before running your producer application. You can change the topic name in the `streaming.properties` file mentioned before:

```
import com.fenago.reader.PropertyReader;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;
import java.util.concurrent.Future;

public class IPLogProducer extends TimerTask {
    static String path = "";

    public BufferedReader readFile() {
        BufferedReader BufferedReader = new BufferedReader(new InputStreamReader(
            this.getClass().getResourceAsStream("/IP_LOG.log")));
        return BufferedReader;
    }

    public static void main(final String[] args) {
        Timer timer = new Timer();
        timer.schedule(new IPLogProducer(), 3000, 3000);
    }

    private String getNewRecordWithRandomIP(String line) {
        Random r = new Random();
        String ip = r.nextInt(256) + "." + r.nextInt(256) + "." + r.nextInt(256) + "."
+ r.nextInt(256);
        String[] columns = line.split(" ");
        columns[0] = ip;
        return Arrays.toString(columns);
    }

    @Override
    public void run() {
```

```

PropertyReader propertyReader = new PropertyReader();

Properties producerProps = new Properties();
producerProps.put("bootstrap.servers",
propertyReader.getPropertyValue("broker.list"));
producerProps.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("auto.create.topics.enable", "true");

KafkaProducer<String, String> ipProducer = new KafkaProducer<String, String>
(producerProps);

BufferedReader br = readFile();
String oldLine = "";
try {
    while ((oldLine = br.readLine()) != null) {
        String line = getNewRecordWithRandomIP(oldLine).replace("[",
""").replace("]", " ");
        ProducerRecord ipData = new ProducerRecord<String, String>
(propertyReader.getPropertyValue("topic"), line);
        Future<RecordMetadata> recordMetadata = ipProducer.send(ipData);
    }
} catch (IOException e) {
    e.printStackTrace();
}
ipProducer.close();
}
}

```

Fraud IP lookup

The following classes will help us as a lookup service which will help us to identify if request is coming from a fraud IP. We have used interface before implementing the class so that we can add more NoSQL databases or any fast lookup service. You can implement this service and add a lookup service by using HBase or any other fast key lookup service. We are using in-memory lookup and just added the fraud IP range in the cache. Add the following code to your project:

```

public interface IIPScanner {

    boolean isFraudIP(String ipAddresses);

}

```

`CacheIPLookup` is the implementation for the `IIPScanner` interface which does in memory lookup:

```

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

public class CacheIPLookup implements IIPScanner, Serializable {

```

```

private Set<String> fraudIPList = new HashSet<>();

public CacheIPLookup() {
    fraudIPList.add("212");
    fraudIPList.add("163");
    fraudIPList.add("15");
    fraudIPList.add("224");
    fraudIPList.add("126");
    fraudIPList.add("92");
    fraudIPList.add("91");
    fraudIPList.add("10");
    fraudIPList.add("112");
    fraudIPList.add("194");
    fraudIPList.add("198");
    fraudIPList.add("11");
    fraudIPList.add("12");
    fraudIPList.add("13");
    fraudIPList.add("14");
    fraudIPList.add("15");
    fraudIPList.add("16");
}

@Override
public boolean isFraudIP(String ipAddresses) {

    return fraudIPList.contains(ipAddresses);
}
}

```

Streaming code

We haven't focused much on modularization in our code. The IP fraud detection application scans each record and filters those records which qualify as a the fraud record based on fraud IP lookup service. The lookup service can be changed to use any fast lookup database. We are using in memory lookup service for this application:

```

import com.fenago.reader.PropertyReader;
import org.apache.Spark.SparkConf;
import org.apache.Spark.api.java.function.Function;
import org.apache.Spark.streaming.api.java.JavaStreamingContext;
import java.util.Set;
import java.util.regex.Pattern;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Map;
import scala.Tuple2;
import kafka.serializer.StringDecoder;
import org.apache.Spark.streaming.api.java.*;
import org.apache.Spark.streaming.kafka.KafkaUtils;
import org.apache.Spark.streaming.Durations;

```

```

public class FraudDetectionApp {
    private static final Pattern SPACE = Pattern.compile(" ");

    private static void main(String[] args) throws Exception {
        PropertyReader propertyReader = new PropertyReader();
        CacheIPLookup cacheIPLookup = new CacheIPLookup();
        SparkConf SparkConf = new
        SparkConf().setAppName("IP_FRAUD").setMaster("local[2]");
        JavaStreamingContext javaStreamingContext = new
        JavaStreamingContext(SparkConf, Durations.seconds(3));

        Set<String> topicsSet = new HashSet<>
        (Arrays.asList(propertyReader.getPropertyValue("topic").split(",")));
        Map<String, String> kafkaConfiguration = new HashMap<>();
        kafkaConfiguration.put("metadata.broker.list",
        propertyReader.getPropertyValue("broker.list"));
        kafkaConfiguration.put("group.id",
        propertyReader.getPropertyValue("group.id"));

        JavaPairInputDStream<String, String> messages = KafkaUtils.createDirectStream(
            javaStreamingContext,
            String.class,
            String.class,
            StringDecoder.class,
            StringDecoder.class,
            kafkaConfiguration,
            topicsSet
        );
        JavaDStream<String> ipRecords = messages.map(Tuple2::_2);

        JavaDStream<String> fraudIPs = ipRecords.filter(new Function<String, Boolean>
        () {
            @Override
            public Boolean call(String s) throws Exception {
                String IP = s.split(",")[0];
                String[] ranges = IP.split("\\.");
                String range = null;
                try {
                    range = ranges[0];
                } catch (ArrayIndexOutOfBoundsException ex) {
                    return false;
                }
                return cacheIPLookup.isFraudIP(range);
            }
        });
        DStream<String> fraudDstream = fraudIPs.dstream();
        fraudDstream.saveAsTextFiles("FraudRecord", "");

        javaStreamingContext.start();
        javaStreamingContext.awaitTermination();
    }
}

```

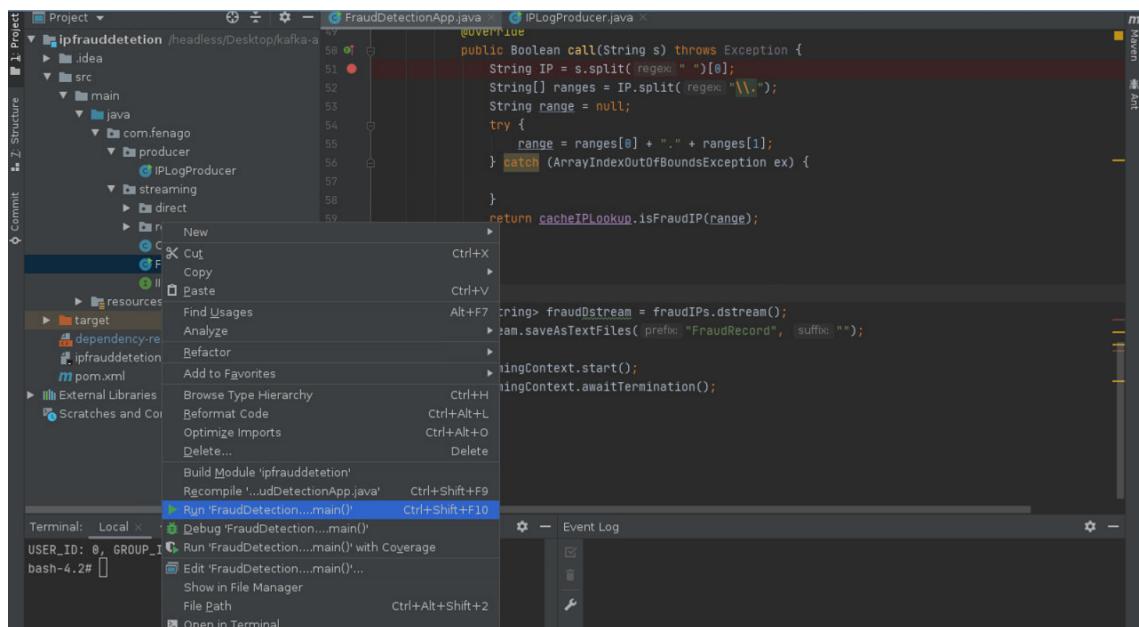
```

    }
}

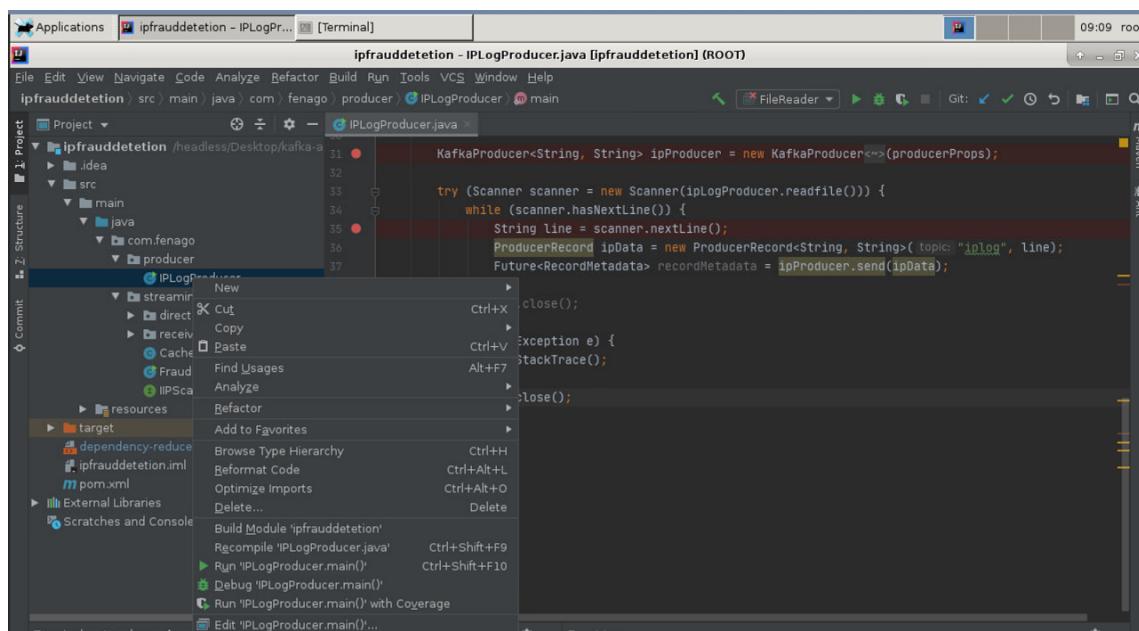
```

Once the Spark Streaming application starts, run Kafka producer and check the records.

Step 1: Run spark streaming code as shown below:



Step 2: Run producer code as shown below:



Summary

In this lab, we learned about Apache Spark. Our focus was on covering different ways we can integrate Kafka with Spark and their advantages and disadvantages. We also covered APIs for the receiver-based approach and direct approach. Finally, we covered a small use case about IP fraud detection through the log file and lookup service. You can now create your own Spark streaming application.