Final Project

Demo Implementation Guide

## Apache Flink Complex Event Processing

## Abstract:

* Complex Event Processing will match incoming events against a pattern and triggers actions/decisions/more complex events.
* CEP is used in IT security, finance (Trading/Fraud/Risk), in healthcare, traffic monitoring, etc.
* Some of these applications cannot tolerate a high latency.
* Real-time processing is the next generation of Big Data analytics; Data is flowing between systems without the need to store irrelevant/redundant data.
* Apache Mapreduce, Spark and Storm fail when it comes to real-time streaming performance due to their batch-based architecture.
* Apache Flink is the raising star that was built specifically for real time stream processing. Yet it can support batch processing.
* We will showcase the use of Flink CEP for patient monitoring

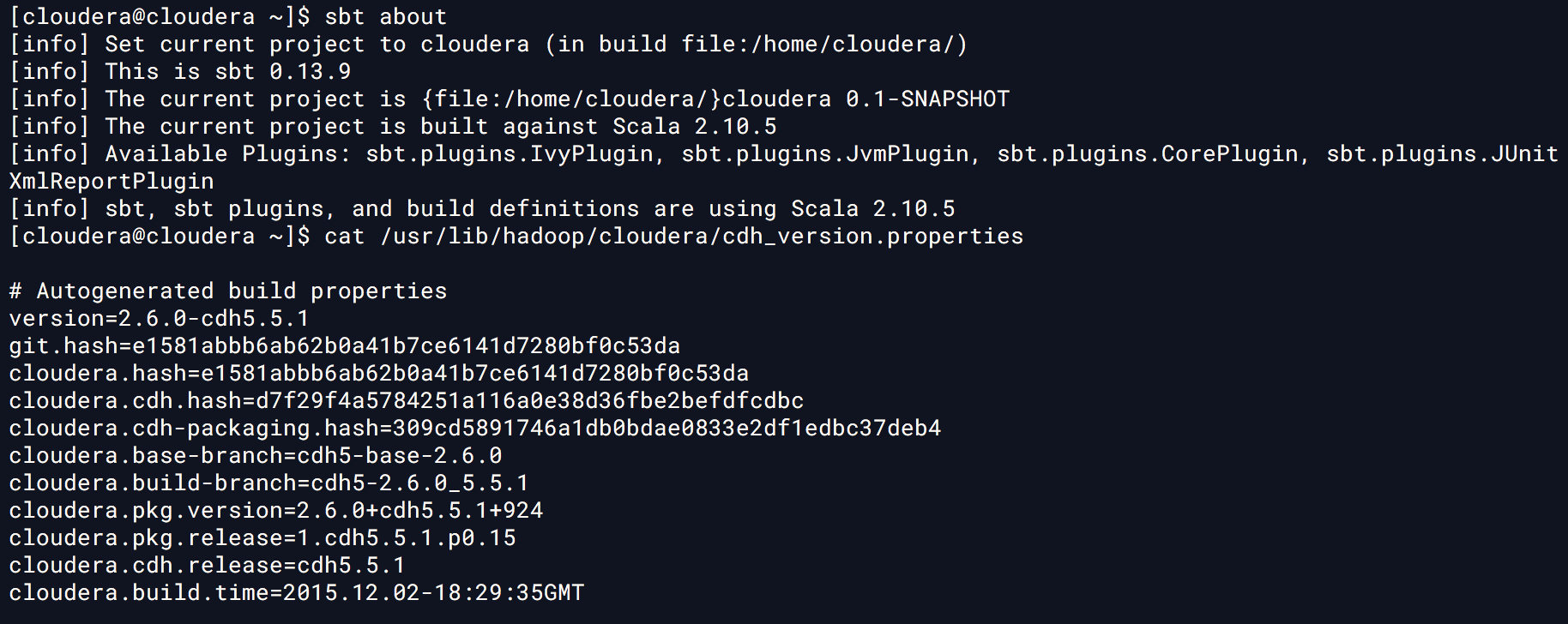
## Installation:

For this project I’ll be using my CentOS 6.7 VM which I used for almost everything during this course. I will also use IntelliJIDE in my MacBook for development.

1. Check environment

$ cat /usr/lib/hadoop/cloudera/cdh\_version.properties

$ sbt about

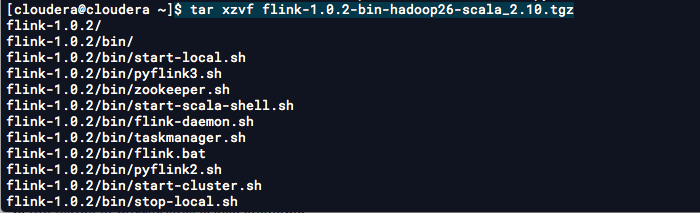


1. Download the appropriate version of Flink corresponding to Hadoop and Scala versions

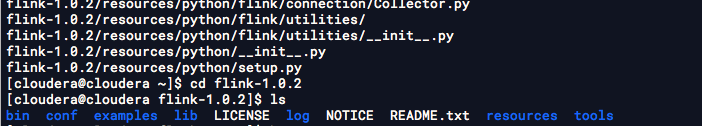


1. Unzip it

$ tar xzvf flink-1.0.2-bin-hadoop26-scala\_2.10.tgz



..



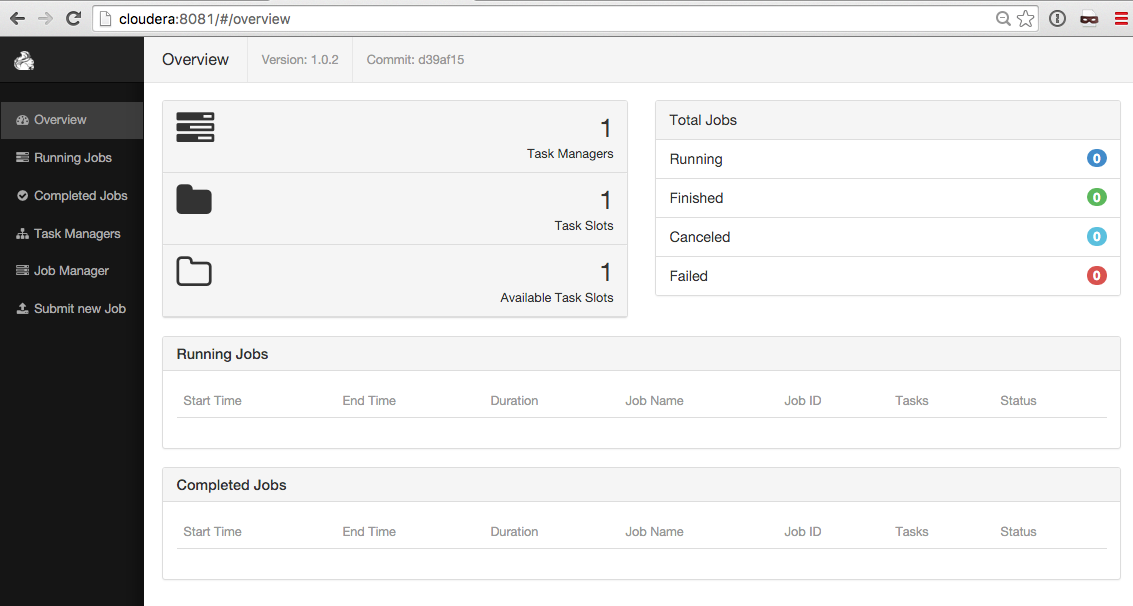
1. Run it

$ cd flink-1.0.2

$ sudo bin/start-local.sh



1. Verify it is working by visiting the page at localhost port 8081



## Demo Development

## Setup Kaka

1. We already have Kafka and Zookeeper installed. Let’s go over there and try to run a *zookeeper* server in background

$ cd /opt/kafka\_2.10-0.9.0.1/

$ screen -S zook

$ sudo bin/zookeeper-server-start.sh config/zookeeper.properties &

1. Detatch from zookeeper screen using CTRL-a + d
2. Go ahead and start *Kafka* server, we’ll keep it run in foreground in another screen session, in case we need to look at the logs or exit the shell, the server will stay there

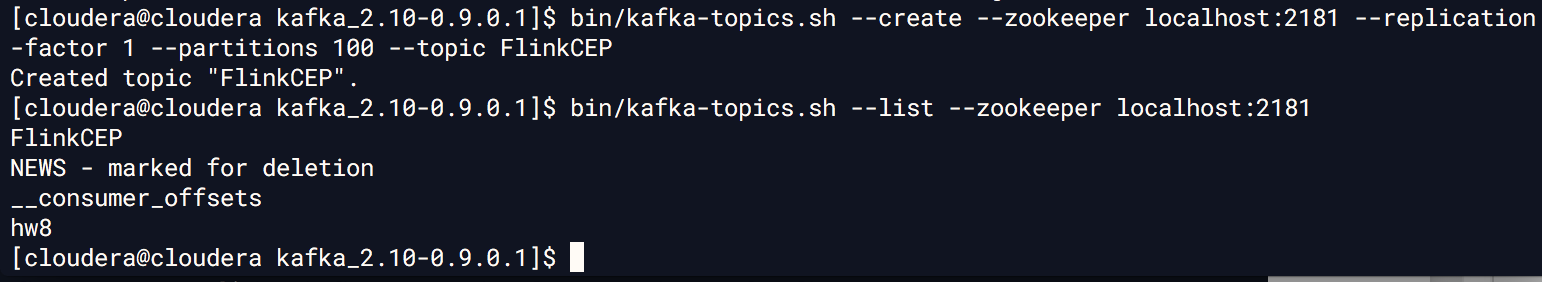
$ screen -S kafka

$ sudo bin/kafka-server-start.sh config/server.properties

1. Create a topic using *Kafka* command line a name it “FlinkCEP” and specify 100 partitions

$ bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 100 --topic FlinkCEP

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



## Python Measurement Producer:

1. The script emulates a remote external source of alarms. It uses the *KafkaProduce* module.

**from** **kafka** **import** KafkaProducer

**from** **random** **import** gauss

**from** **time** **import** sleep

**import** **json**

1. The below functions generate a random measurement of HeartRate, Blood Pressure and Temperature. I used a normally distributed probability function so each value is likely to mimic the real world ranges. The mean and SD are estimated based on MDCalc.com

## Heart rate BPM (Beat per minute)

HR\_MU = **70**

HR\_SD = **20**

**def** **getHeartRate**() :

**return** int(gauss(HR\_MU, HR\_SD))

## Systolic Blood Presure : mmHg

SBP\_MU = **120**

SBP\_SD = **25**

**def** **getSystolicBloodPressure**() :

**return** int(gauss(SBP\_MU, SBP\_SD))

## Temperature Celcius

TEMP\_MU = **37**

TEMP\_SD = **0.5**

**def** **getTemperature**() :

**return** format(gauss(TEMP\_MU, TEMP\_SD), '.2f')

1. Inside the main function we initialize the *KafkaProducer* and assign a JSON serializer

producer = KafkaProducer(value\_serializer=**lambda** v: json.dumps(v).encode('utf-8'), bootstrap\_servers=server)

1. The next piece of the code will loop infinitely and generate 3 measurements for each user a a time,

**while** True:

**for** userId **in** range(**1**,**200**) :

meas1 = { "userid":"%d"%userId, "type" : "HR", "value" : getHeartRate()}

producer.send('FlinkCEP', meas1, key = b'%d'%userId)

meas2 = { "userid":"%d"%userId, "type" : "TEMP", "value" : getTemperature()}

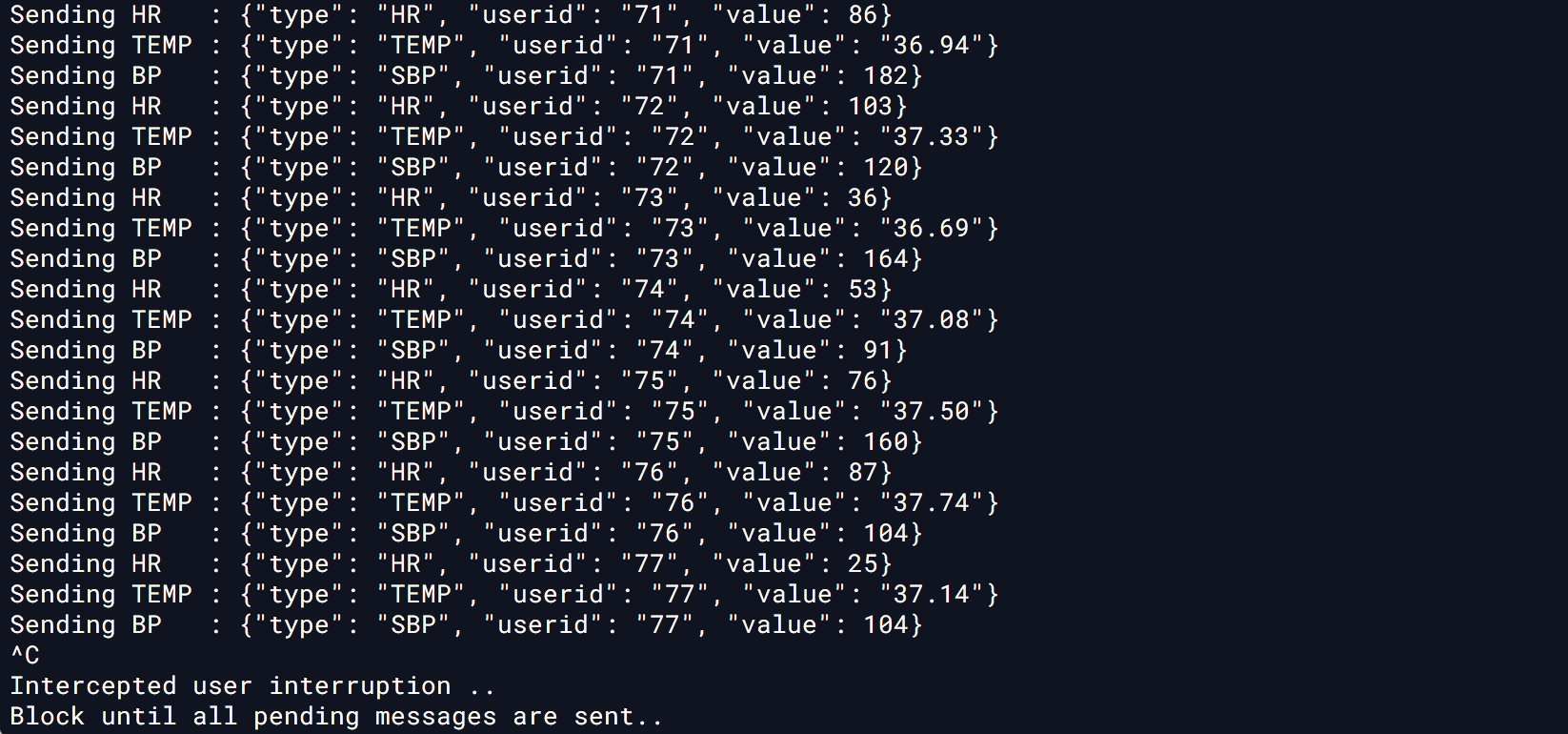
producer.send('FlinkCEP', meas2, key = b'%d'%userId)

meas3 = { "userid":"%d"%userId, "type" : "SBP", "value" : getSystolicBloodPressure()}

producer.send('FlinkCEP', meas3, key = b'%d'%userId)

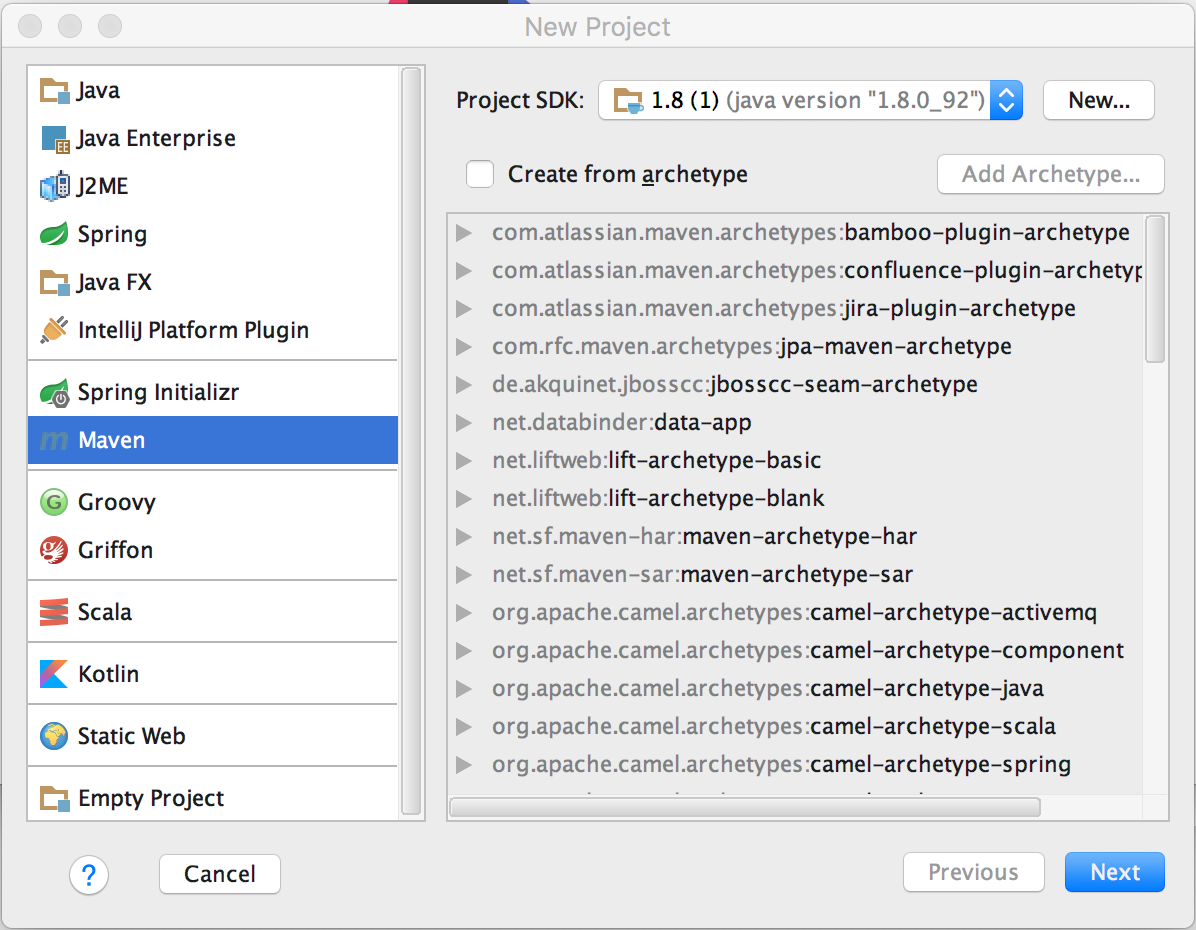
Notice below that we use the *userId* as key for partitioning

Let’s see how it works:

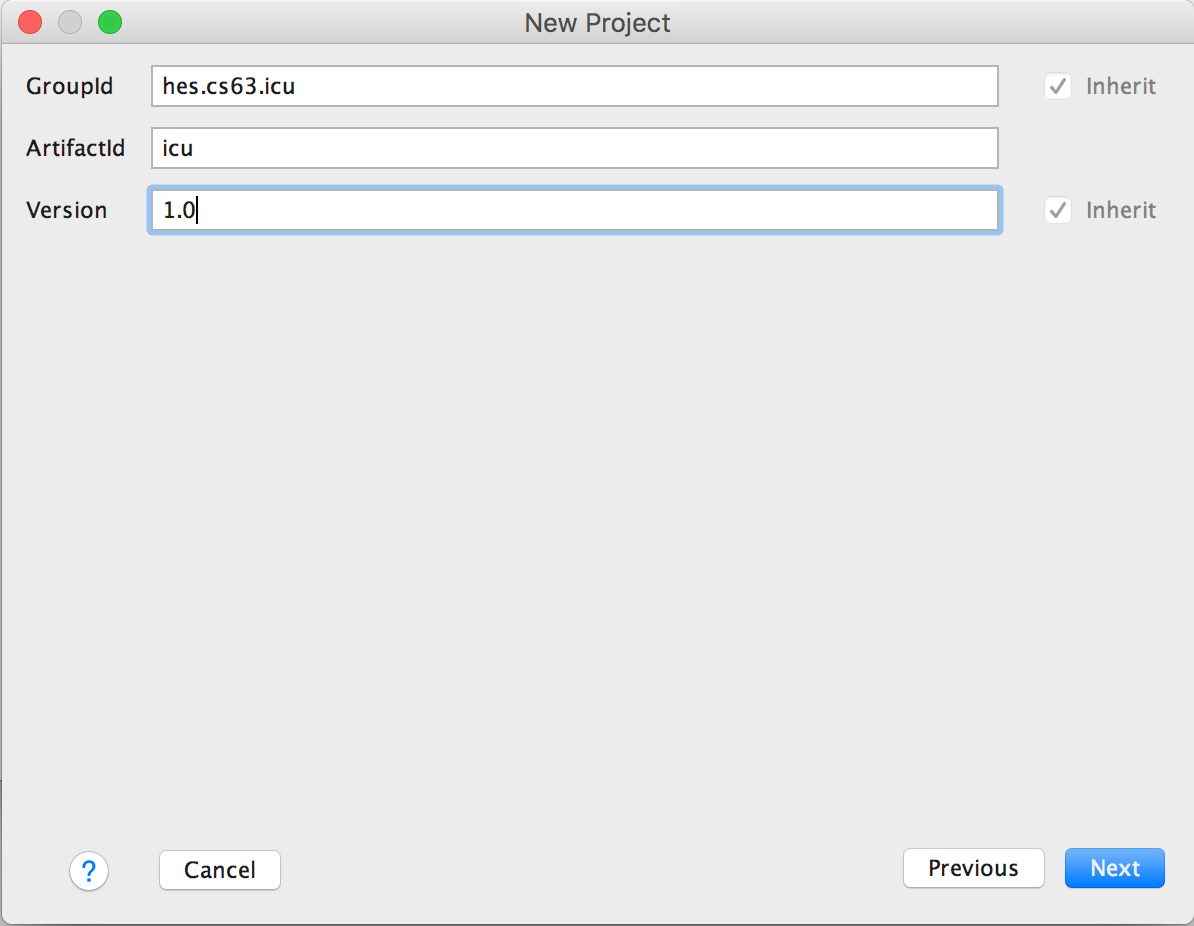


## Flink CEP Program

1. Let’s start by creating a java maven project using IntelliJIDE. I’ll use JDK 1.8 and code level 8 in order to use lambda functions



1. Give it name and artifact and hit Finish.



1. Add the Flink API dependencies to the *pom.xml* file :

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-streaming-java\_2.10</**artifactId**>  
 <**version**>1.0.2</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-cep\_2.10</**artifactId**>  
 <**version**>1.0.2</**version**>  
</**dependency**>

1. Add Kafka Flink connector and its dependencies

<**dependency**>  
 <**groupId**>org.apache.kafka</**groupId**>  
 <**artifactId**>kafka-clients</**artifactId**>  
 <**version**>0.9.0.1</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-connector-kafka-0.9\_2.10</**artifactId**>  
 <**version**>1.0.2</**version**>  
</**dependency**>

1. Include dependency for GSON library that will be used in de-serializing Kafka messages

<dependency>  
 <groupId>com.google.code.gson</groupId>  
 <artifactId>gson</artifactId>  
 <version>2.6.2</version>  
</dependency>

1. Create a main class and call it CEPMonitor. The main method is explained below:

* Setup the StreamExecutionEnvironment

**StreamExecutionEnvironment** env =  
 **StreamExecutionEnvironment**.*getExecutionEnvironment*();  
  
ParameterTool parameterTool = ParameterTool.*fromArgs*(args);  
  
*// Use ingestion time => TimeCharacteristic == EventTime + IngestionTimeExtractor*env.setStreamTimeCharacteristic(TimeCharacteristic.***EventTime***);

* Create an Input DataStream of monitoring events and associate to it the *FlinkKafkaConsumer* object. All Kafka consumer parameters are extracted from the program arguments during runtime. It’s more dynamic, this way.
* I used a custom *MeasurementDeserializer*, that we’ll see in detail later

DataStream<Measurement> messageStream = env  
 .addSource(**new** FlinkKafkaConsumer09<>(  
 parameterTool.getRequired("topic"),  
 **new** MeasurementDeserializer(),  
 parameterTool.getProperties()))  
 .assignTimestampsAndWatermarks(**new** IngestionTimeExtractor<>());

* Then partition the message stream by *UserID* attribute

DataStream<Measurement> partitionedInput = messageStream.keyBy(  
 **new KeySelector**<Measurement, String>() {  
 *@Override* **public** String getKey(Measurement value) **throws** Exception {  
 **return** value.getUserID();  
 }  
});

* Create the Warning pattern: 2 high heart rate events with a high blood pressure followed by a within 10 seconds

Pattern<Measurement, ?> alarmPattern = Pattern.<Measurement>*begin*("first")  
 .subtype(HeartMeasurement.**class**)  
 .where(evt -> evt.getRisk() >= 1)  
 .followedBy("middle")  
 .subtype(BloodPressureMeasurement.**class**)  
 .where(evt -> evt.getRisk() >= 2)  
 .followedBy("last")  
 .subtype(HeartMeasurement.**class**)  
 .where(evt -> evt.getRisk() >= 3)  
 .within(Time.*seconds*(10));

* Create a pattern stream from alarmPattern

PatternStream<Measurement> patternStream = CEP.*pattern*(partitionedInput, alarmPattern);

* Then, generate risk warnings for each matched alarm pattern

DataStream<StrokeRiskAlarm> alarms = patternStream.select(**new PatternSelectFunction**<Measurement, StrokeRiskAlarm>() {  
 *@Override* **public** StrokeRiskAlarm select(Map<String, Measurement> pattern) **throws** Exception {  
 HeartMeasurement first = (HeartMeasurement) pattern.get("first");  
 HeartMeasurement last = (HeartMeasurement) pattern.get("last");  
 BloodPressureMeasurement middle = (BloodPressureMeasurement) pattern.get("middle");  
  
 **return new** StrokeRiskAlarm(first.getUserID(), first.getRisk() + last.getRisk() + middle.getRisk());  
 }  
});

1. We mentioned previously the custom consumer. MeasurementDeserializer. This class will deserialize JSON received events into POJO objects of class Measurement.

public **class** **MeasurementDeserializer** implements KeyedDeserializationSchema<Measurement> {

private Gson gson;

**@Override**

public Measurement deserialize(byte[] messageKey,

byte[] message,

String topic,

int partition,

long offset) throws IOException {

**if** (gson == null) {

gson = new Gson();

}

Measurement m = gson.fromJson(new String(message), Measurement.class);

**if** (m.getType().equals("HR")) {

**return** new HeartMeasurement(m);

} **else** **if** (m.getType().equals("TEMP")) {

**return** new TempMeasurement(m);

} **else** **if** (m.getType().equals("SBP")) {

**return** new BloodPressureMeasurement(m);

} **else** **return** m;

}

This class inplements KeyedDeserializationSchema<T> to convert a String to JSON object.

1. I used the GSON’s **@SerializedName** annotations for JSON conversion to POJO objects:

public class Measurement {

@SerializedName("userid")

private String UserID;

@SerializedName("type")

private String Type;

@SerializedName("value")

private float Value;

1. Now build the jar using maven and run it.

$ java -jar target/flinkicu-1.0-jar-with-dependencies.jar --topic FlinkCEP --bootstrap.servers cloudera:9092 --zookeeper.connect cloudera:2181 --group.id SBG

We split the screen into producer and consumer. To the left you see the event producer running at 600 message/second. And on the right we have the CEP consumer which produces roughly an Alert each 10 seconds.

