CELLULAR IOT SOLUTIONS II

FIRST EDITION

Developers Guide - Intermediate/Advanced

Introduction **3**

Development Environment **5**

IDE 5

Flink Dependencies 5

Project Creation 6

Flink Application **7**

Pom Dependencies 7

Streaming Application Code 9

Testing 11

Cluster Deployment **13**

Package Application 13

Cluster Environment 14

Deploy Application 15

INTRODUCTION

With IoT growing at an exponential rate it is imperative that we continuously adapt our streaming architectures to stay ahead of the game.
Apache Flink provides many great capabilities for stream processing that can be applied to a modern data architecture to enable advanced stream processing like windowing of streaming data, anomaly detection, and process monitoring to name a few.

In the first book of this series we dug into actual device at the edge, its programing and Integration points and the full ingest and storage flow to bring the data in from the edge and take real time action using business logic.

AI is an often overused term that really encompasses a huge realm of possibilities. Many people think Artificial Intelligence is a machine that uses the most advanced data science models and is able to "think" on its own. This is a very narrow view of the field. AI can be as simple as the automated real time decisions and actions we programmed in the exercise of the first book. Being able to take simple business logic and program it into your ingestion framework to route and take action on the data streaming into your organization is a huge step into the AI driven Enterprise.

That said there are many other ways to grow your AI footprint. The addition of real time scoring that leverages ML models that have been trained and deployed gives you advanced routing and action capacity that rules based systems can't easily deliver.

Advanced Stream Processing (ASP) introduces more complex functionality into your stream processing framework like being able to query a stream of information, looking at a window of time or of events, defining the "time" in which you process events (ie. Event time, Processing Time, Sensor Time…) which gives your AI framework a "stream memory" on which it can operate on multiple events and even multiple streams of events simultaneously. ASP is what we will be examining in more detail as we dive into Apache Flink and deploy our first stream processing application.

Let's look at an Advanced example that will build on what we developed in the first book in this series.

I will walk you through the end to end solution designed to:

1. Set up an Apache Flink development environment
2. Create a Flink application and run tests in your development environment
3. Deploy your Application to the Flink cluster and monitor the job

If you are planning to follow along, code for all stages of the Cellular-IoT-Solutions I book can be found on GitHub by following this link:

https://github.com/vvagias/IoT_Solutions

If you haven't gone through that book and it's examples I recommend that you do as it is an assumption for the examples in this book.

You may also find additional tutorials and ask questions here:
https://community.cloudera.com

Let's begin!

DEVELOPMENT ENVIRONMENT

IDE

First things first. We need to have an IDE that we are comfortable with installed and ready to use. I will be using IntelliJ IDE as that is my personal preference but you are free to choose eclipse or even do all of this in the terminal which I will admit I like to do from time to time.

Things to think about here are the features you might want or need.

Code completion, auto imports, svn and source controls, git integration… etc…

IntelliJ Community edition is free and gives me everything I need.

FLINK DEPENDENCIES

Before the IDE will be useful to us for Flink development we need to make sure we are set up will all the dependencies.

If you feel lost or that I have skipped a step for you, the full and detailed instructions can be found on the Flink documentation page in the turorial: https://ci.apache.org/projects/flink/flink-docs-release-1.9/getting-started/tutorials/local_setup.html

The important things are that you ensure you have Java 8 installed.

Verify this with the following command:

```
java -version
```

Once you have that established you can move on to install a local Flink repo. If you are using a Mac this is made very simple using Brew!

```
$ brew install apache-flink
...
$ flink --version
Version: 1.2.0, Commit ID: 1c659cf
```

You can also pull the latest bits through a direct download and build locally:

https://flink.apache.org/downloads.html

PROJECT CREATION

Once you have Flink installed you can continue setting up your development environment.

There are many ways to set up a new project. You can use the quickstart.sh script, you can use sbt to build a Scala template project or maven to create an archetype.

We will use maven for no particular reason other than it's what most people find familiar.

You can create this through your IDE or just run the command in a terminal and import the project.

I prefer the terminal approach.

```
mvn archetype:generate                                \
      -DarchetypeGroupId=org.apache.flink              \
      -DarchetypeArtifactId=flink-quickstart-scala     \
      -DarchetypeVersion=1.9.0
```

You can modify the version to your desired value, when I wrote this chapter 1.9.0 was the go to.

You will be prompted for groupID, artifactId and version. Enter what you desire here. Below is my build as an example.

[INFO] Parameter: version, Value: 1.0-SNAPSHOT

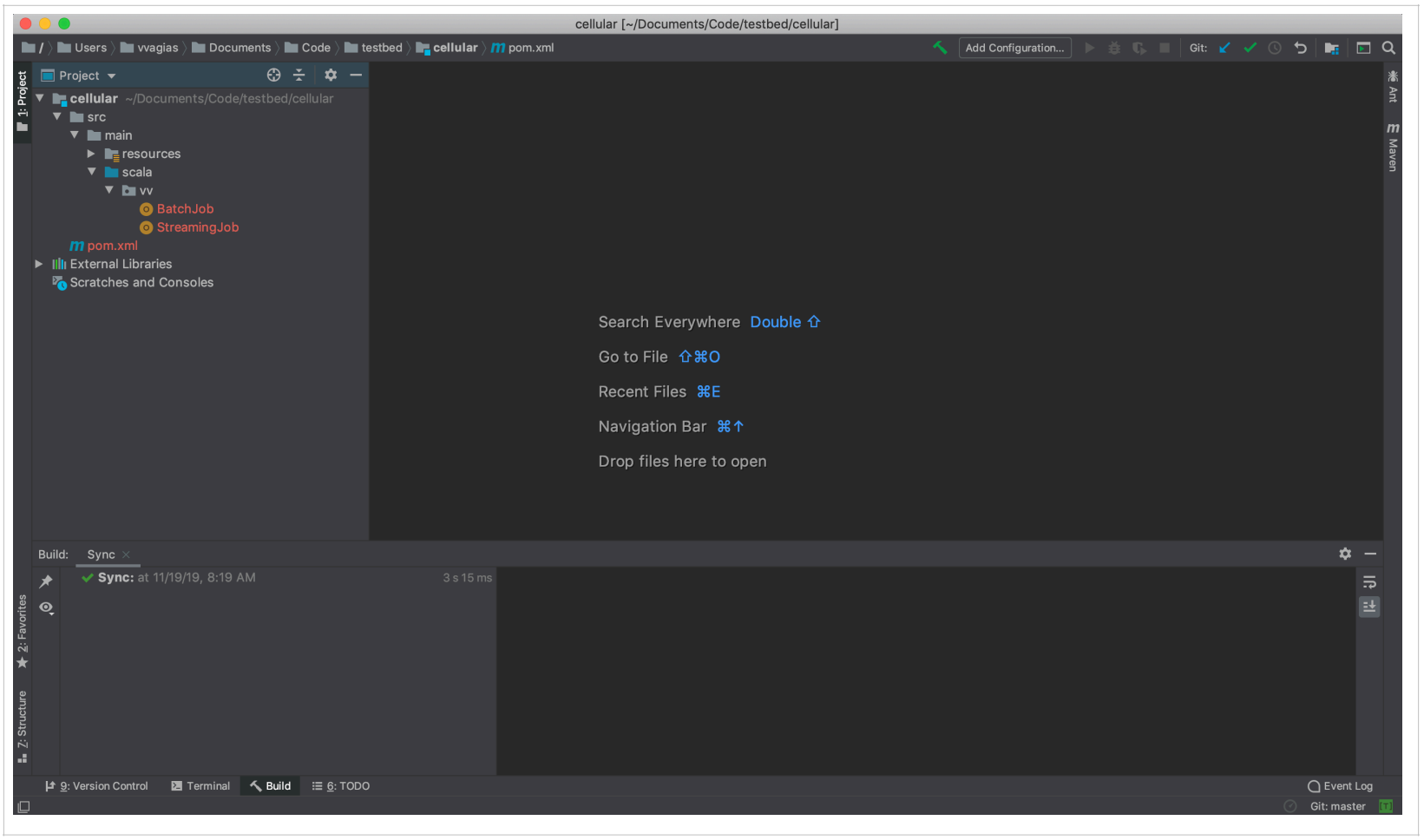
[INFO] Parameter: groupId, Value: vv

[INFO] Parameter: artifactId, Value: cellular

You get as an output a nice directory with the pom.xml we need to specify additional dependencies as well as a stub for your src folder.

- Import this into IntelliJ and select Open Project.
- Select the folder we just created using mvn
- Select open

You should now have a clean new project in your IDE. Congrats! You are all set up in your IDE and ready to take over the world.



FLINK APPLICATION

POM DEPENDENCIES

With our IDE set up and a new project ready to go, we can now begin to layout our Flink application.

Looking at your project a pom.xml should be located in the root directory.

Inside you will see a lot of information but we are going to zero in on the dependencies section.

The maven generated pom.xml has the following already included:

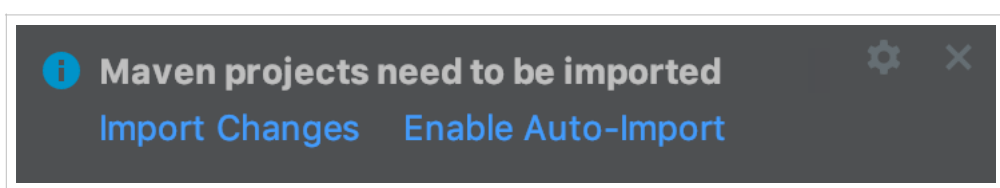
```
<dependencies>
    <!-- Apache Flink dependencies -->
    <!-- These dependencies are provided, because they should not be packaged into the JAR
file. -->
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-scala_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-scala_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- Scala Library, provided by Flink as well. -->
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
        <version>${scala.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- Add connector dependencies here. They must be in the default scope (compile). -->

    <!-- Example:

    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka-0.10_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
    </dependency>
    -->

    <!-- Add logging framework, to produce console output when running in the IDE. -->
    <!-- These dependencies are excluded from the application JAR by default. -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.7</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

For this solution we will need an additional dependency.

Add the following to your pom file.

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_2.11</artifactId>
    <version>1.9.0</version>
</dependency>
```

You can now let your IDE import the pom changes. A popup should appear where you can select to import the new dependency.



For now that is all we need in our pom.xml to develop our first Flink App.

STREAMING APPLICATION CODE

Looking at your Project you can navigate your directory by opening src->main->scala

In here you will find your source folder with two pre made artifacts for you to modify. One is for Batch and the other for Streaming jobs. We will focus on Streaming.

We have an Object named StreamingJob with a main method. In our main method we can define the properties of our Kafka cluster where we wrote our sensor date in the “Decouple” section of the first book in the series.

Here is all you need to create a KafkaConsumer and pass its data to a stream.

```
val properties = new Properties()
properties.setProperty("bootstrap.servers", "ip.ec2.internal:9092")
// only required for Kafka 0.8
//properties.setProperty("zookeeper.connect", "ip.ec2.internal:2181")
properties.setProperty("group.id", "FlinkTemp")

val myConsumer = new FlinkKafkaConsumer[ObjectNode]("highTemp", new
JSONKeyValueDeserializationSchema(false), properties)

val stream = env.addSource(myConsumer)
```

We now have a val named “stream” which we can use to act on our sensor data as it hits our “highTemp” Kafka topic. We have created a group.id “FlinkTemp”, but this is arbitrary and you can call it whatever makes sense to you and your organizations best practices.

Using the stream object you can operate on each event as well as the stream as a whole using the Flink API. There are many features available to you; however, we will just focus on one very interesting feature… Windowing!

Let’s create a window function that looks at the event stream and then triggers an action once a given number of events have been processed.

It helps to have a sample of the data you are working with available to you when developing.

We were able to grab this from NiFi in the first book in this series. Here it is for reference:

```
{
"event": "readings",
"data": "69.80",
"published_at": "2019-11-14T15:59:51.534Z", "coreid": "330032001247373333353132"

}
```

We can create a map function that takes the ObjectNode, which is how we receive the data from our Kafka Consumer, and maps it into a tuple of type (String, Double) and we select the JSON elements that we are interested in. Here I select the “coreid” and the “data” which is the temperature reading.

Then I operate on the tuple by using the keyBy(0) to key on the coreid, countWindow(3) which will trigger the function only once 3 events for that key have been processed, our function we wish to use here I simply request the max value in the window ant then print to see the data.

```
//Print max value temperature in the window...
stream.map(new MapFunction[ObjectNode, (String, Double)]() {
  @throws[Exception]
  override def map(node: ObjectNode): (String, Double) =
(node.get("value").get("coreid").asText(), node.get("value").get("data").asDouble())
})
  .keyBy(0)
  .countWindow(3)
  .maxBy(1).print()
```

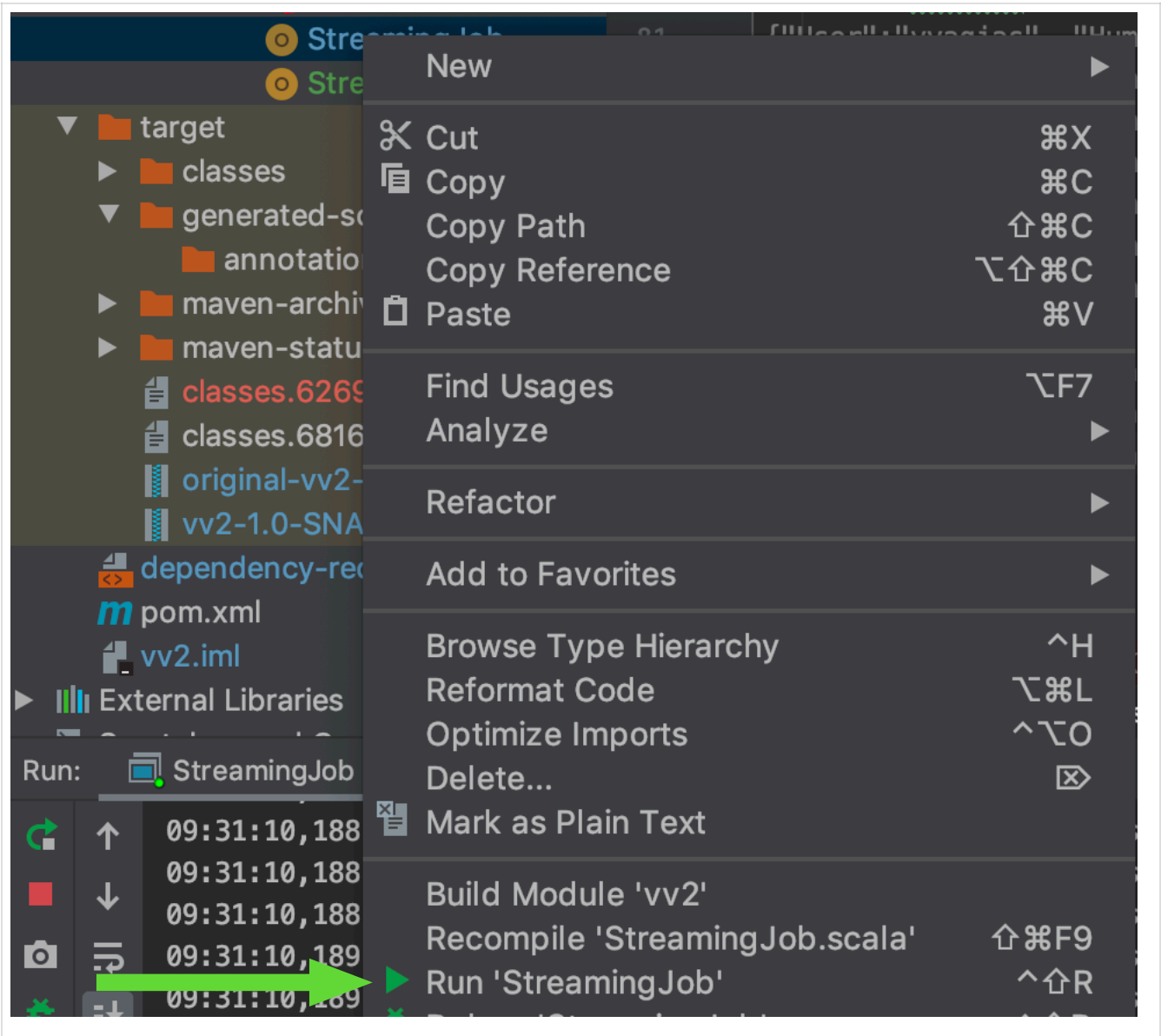
Notice we used coreid for the keyBy() parameter... This is conveniently chosen for scale because as we add devices the coreid will identify the device which will map to other important information such as the devices name, location, available functions etc. If the device is continuously moving and doesn't stay at a static location we could add the device location in the data readings. Otherwise you can save bandwidth by maintaining a lookup table with your deployed devices.

That is all we need for this exercise so we can go ahead and save our code.

TESTING

Before you submit your code to a cluster, especially one that is multi-tenant and affects many users it's a great idea to have executed the code and run some tests.

Keeping the code in your IDE we can now right click your StreamingJob file and click Run.



This will print a lot of info to the console and also start a small local Flink cluster. You can scroll through the log to find the web address if you are interested but for our testing we don't need it.

With the application running on your local test cluster you can now call the publish function a few times to get some readings.

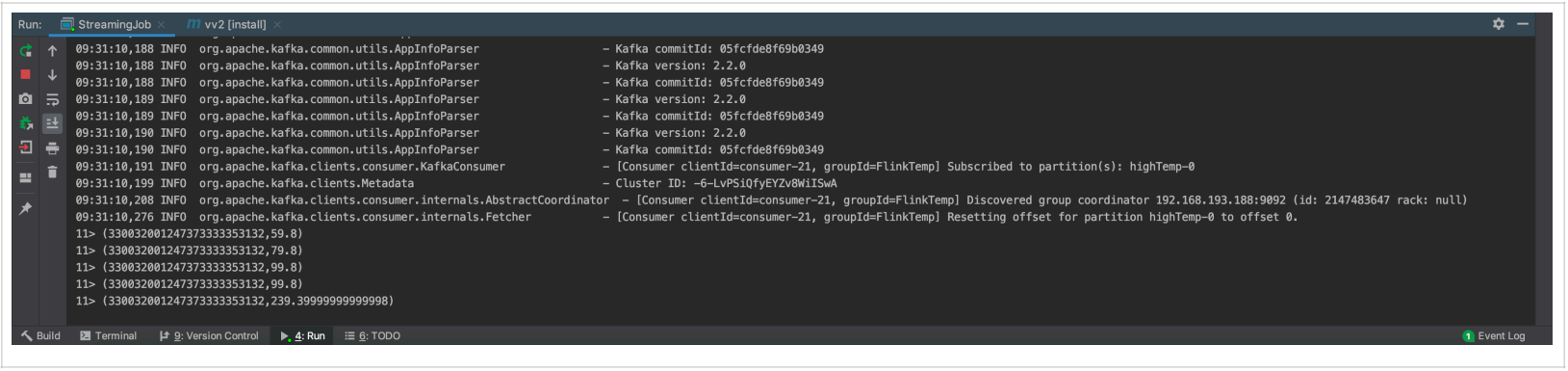
This can be done through curl commands, through a quick and dirty kafka producer in your terminal, through a NiFi flow or through the Particle console. I have even added a hardware button on my development device which publishes when I click to make it easy. Do what you enjoy and what creates the least friction for you.

Here is some data to use for testing:

```
{"event":"readings","data":"59.80","published_at":"2019-11-14T15:40:51.983Z","coreid":"3300
32001247373330053132"}
{"event":"readings","data":"79.80","published_at":"2019-11-14T15:40:51.983Z","coreid":"3300
32001247373335300132"}
{"event":"readings","data":"99.80","published_at":"2019-11-14T15:40:51.983Z","coreid":"3300
32001247373335300132"}
```

If you create a quick terminal producer for now and paste that in we can verify that our application is in fact working as expected.

When 3 events have been published, if you have set everything up properly, you will see an output in the console with the coreid and the largest of the three readings.



This is excellent! You may have noticed I have some additional printouts. This is ok, I added some additional prints. You should take some time and customize your app a bit. I am printing out readings as I get them as well as the sum of the readings here. Do whatever you want and have fun exploring the Flink documentation.

https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/datastream_api.html

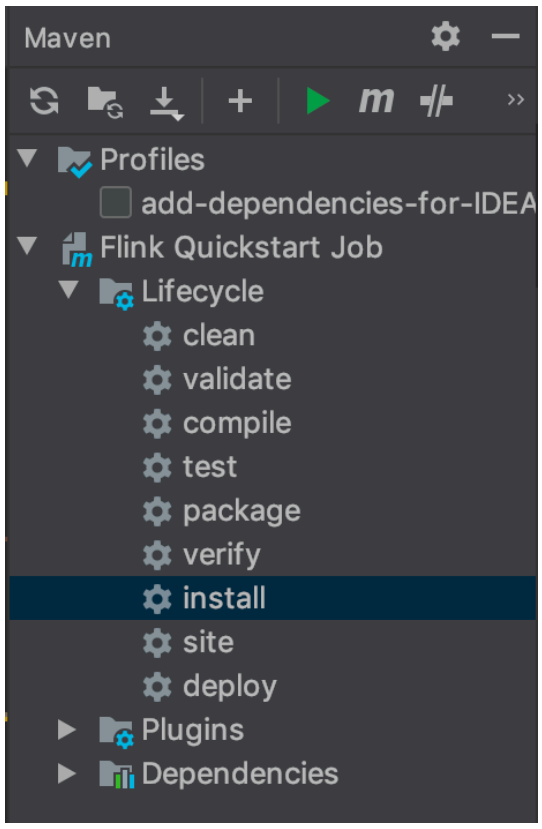
You can also find good examples and connect with others in the community:

https://community.cloudera.com

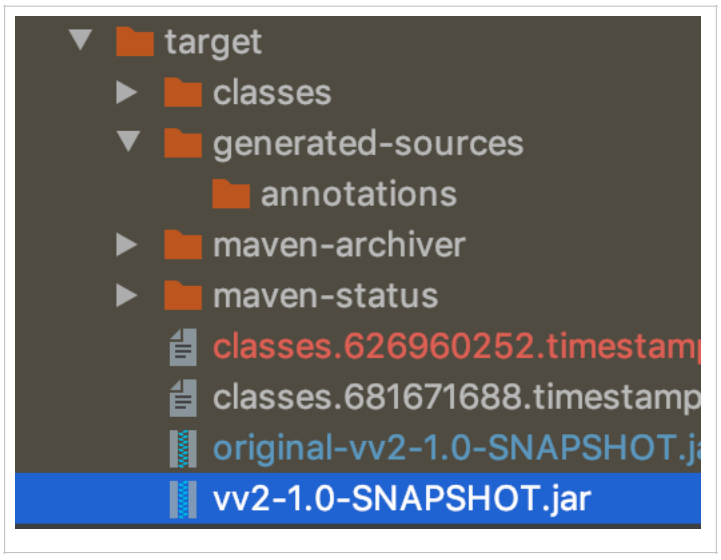
CLUSTER DEPLOYMENT

PACKAGE APPLICATION

In your IDE you can double click your pom.xml to open an options menu for maven lifecycle tasks.

In the menu run the clean and then the package and install tasks. This will create a target directory and an artifact jar that holds our Flink dependencies and application code.	

Note the location of this jar as you will need it for deployment.



CLUSTER ENVIRONMENT

Ideally you have a Cloudera CDP environment that you can deploy to and in this case the deployment has been taken care of for you. If not and you are going through this for the first time you will need to start the Flink cluster.

For this book we will set up a local staging cluster on our machine to keep things contained to one environment and make it easy for everyone to follow along.

If you are reading this as you develop and deploy an IoT solution in your enterprise environment and you need help setting up the environment reach out to your Cloudera Account Executive or if you don't have one just fill out the contact form on the website and someone will reach out to help you with the full deployment process and getting you up and running fast.

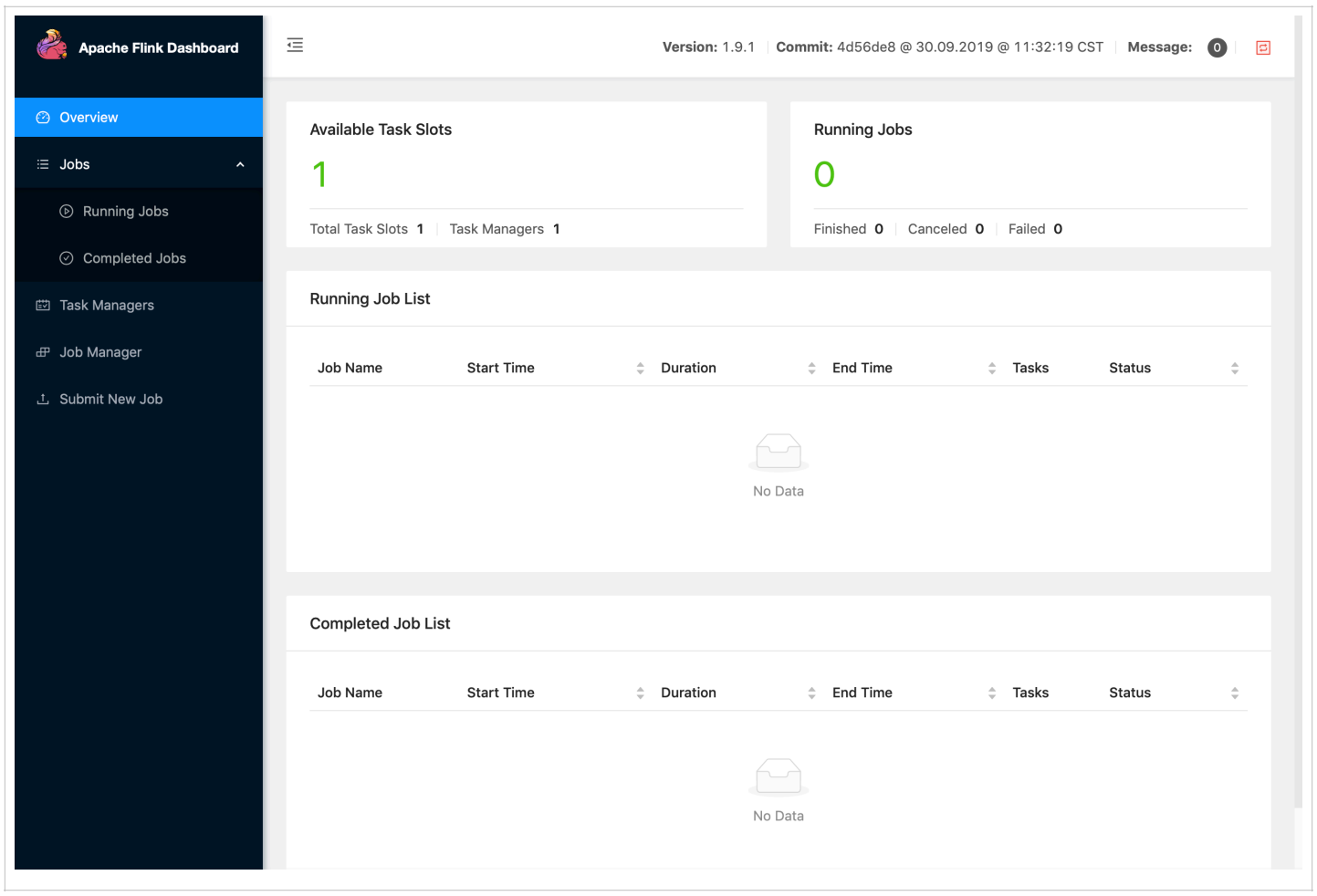
In your terminal go ahead and run the following inside the Flink directory:

```
./bin/start-cluster.sh  # Start Flink
```

This will start a Flink cluster and an admin UI which you can access here:

http://localhost:8081/#/overview

You should have a slick UI like this:



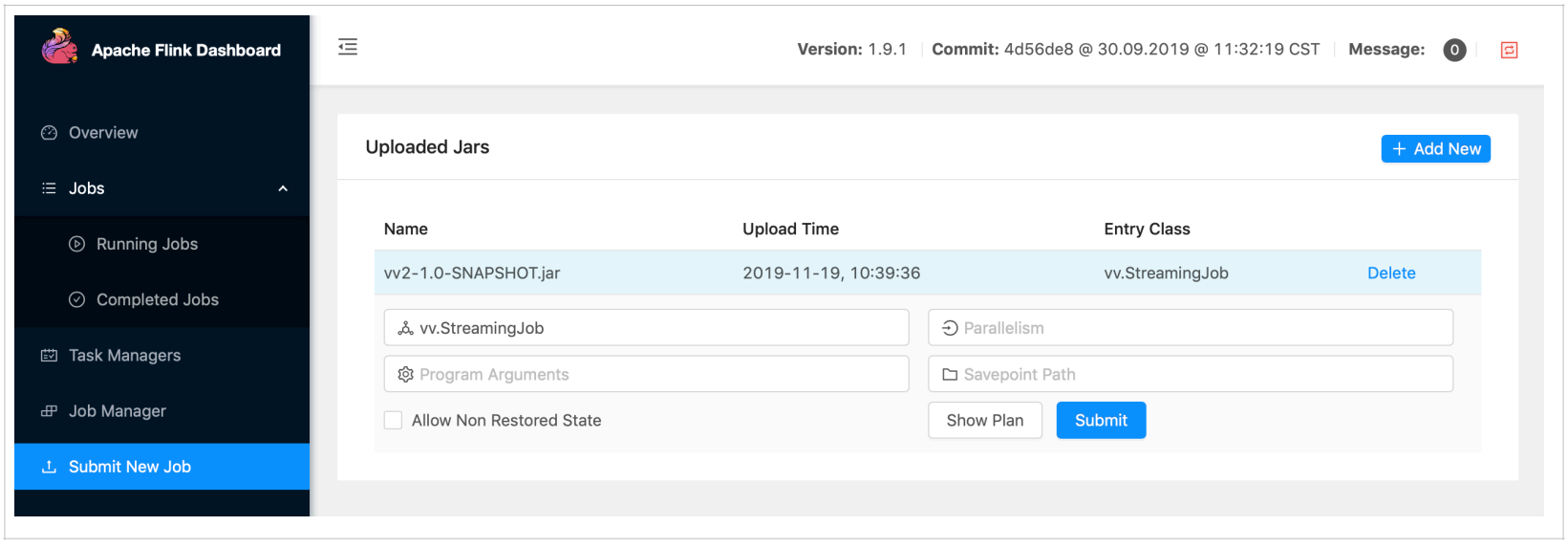
Take a look around and get comfortable. You have the overview as well as the Jobs tab. Also note the “Submit New Job” tab…

DEPLOY APPLICATION

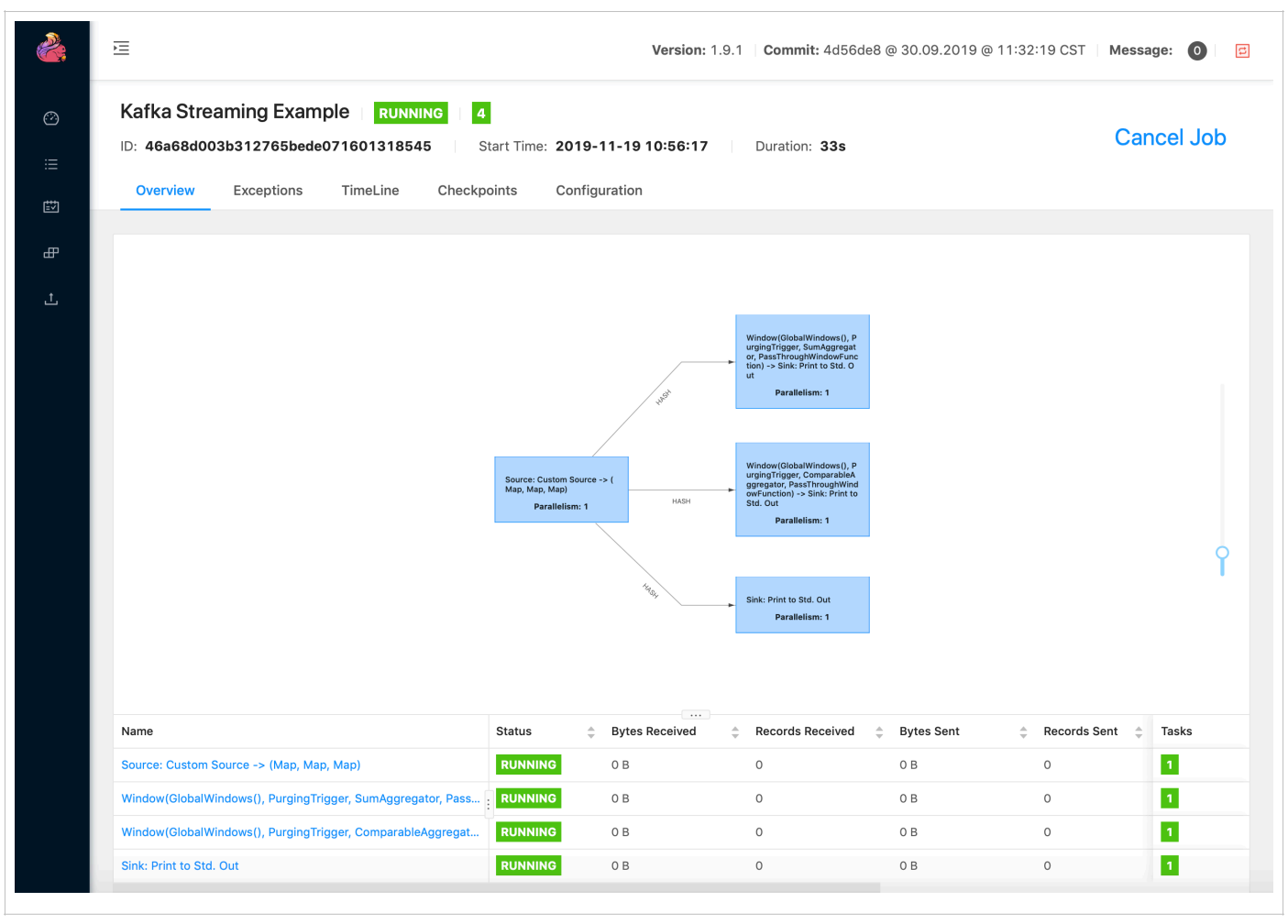
Go ahead an click “Submit New Job”.

This will give you a browser window to navigate to your jar that we created earlier.

Select your jar and submit it. Now click on your application :

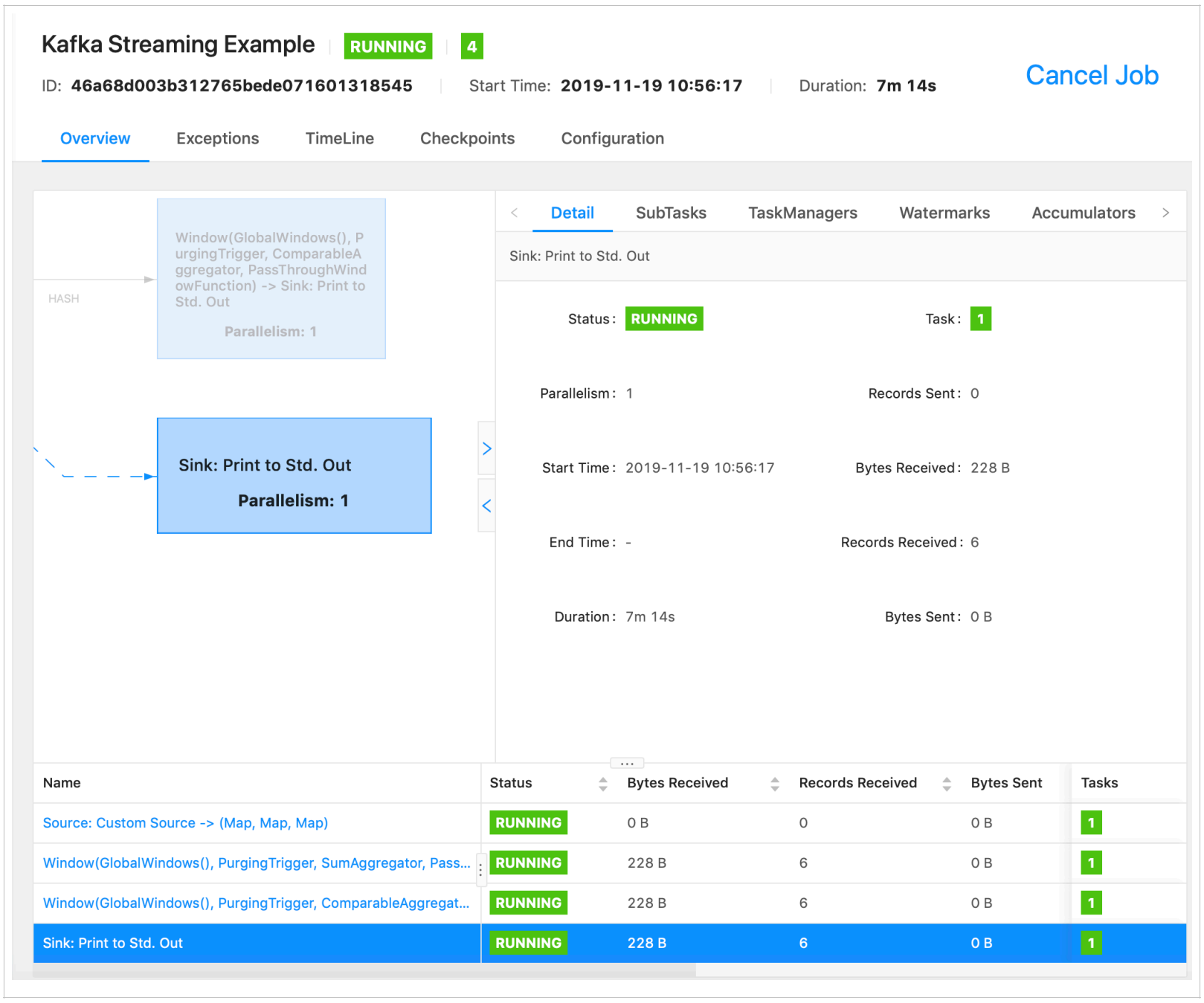


If you set anything fancy up to accept arguments you can specify them here. Otherwise just hit submit.



Awesome! We have a deployed Flink application running on a local Flink cluster. The UI shows you a DAG of the application we built and even shows you that our output is being directed to STDOUT. That is perfect! We left print statements in the code for this exercise and in the next Book in the series we will look at how Flink can write it's output dynamically to Kafka or even directly back to a NiFi input.

You can go back to your Kafka Producer and paste the test data in to see the Job Manager reflect the events:

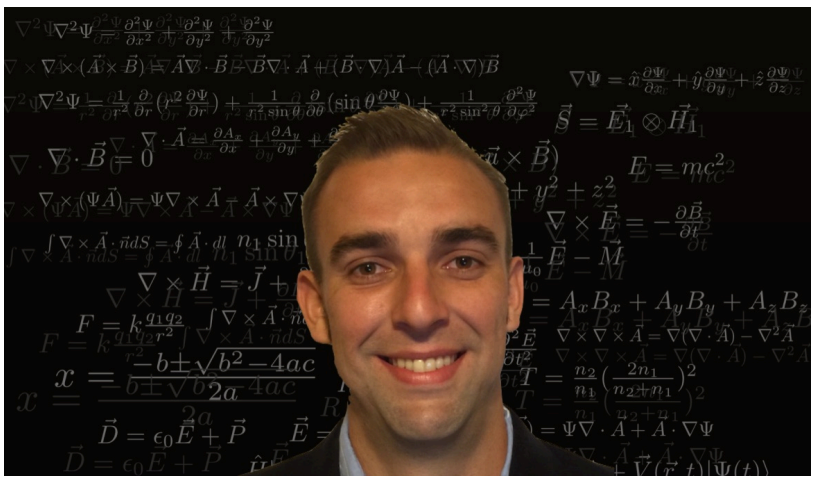


Six records received and processed successfully. Things look good.

So now you have extended the solution to incorporate Apache Flink and real time stram processing. The application is deployed and will be waiting for data events to stream into Kafka for processing.

This is the end of the exercise for this book. In the next book we will build up our solution to move our data from Flink to Kafka as well as NiFi for further processing and also examine a very important element in our solution now that we are using Kafka. Replication! We will look at Stream Replication Manager and Stream Message Manager and use it to build HA and failover into our solution. We will kill Kafka and watch how our solution keeps on processing regardless.

Thanks for reading and I hope you got value from this.



Vasilis Vagias

Senior Solutions Engineer @Cloudera

@vvagias