Chicago Java Users Group

&

Chicago Advanced Analytics Meetup

June 8th 2017

Kafka Streams For Java Enthusiasts

Slim Baltagi

Agenda

1. Apache Kafka: a Streaming Data Platform
2. Overview of Kafka Streams
3. Writing, deploying and running your first Kafka Streams application
4. Code and Demo of an end-to-end Kafka-based Streaming Data Application
5. Where to go from here for further learning?

Batch	Streaming
	

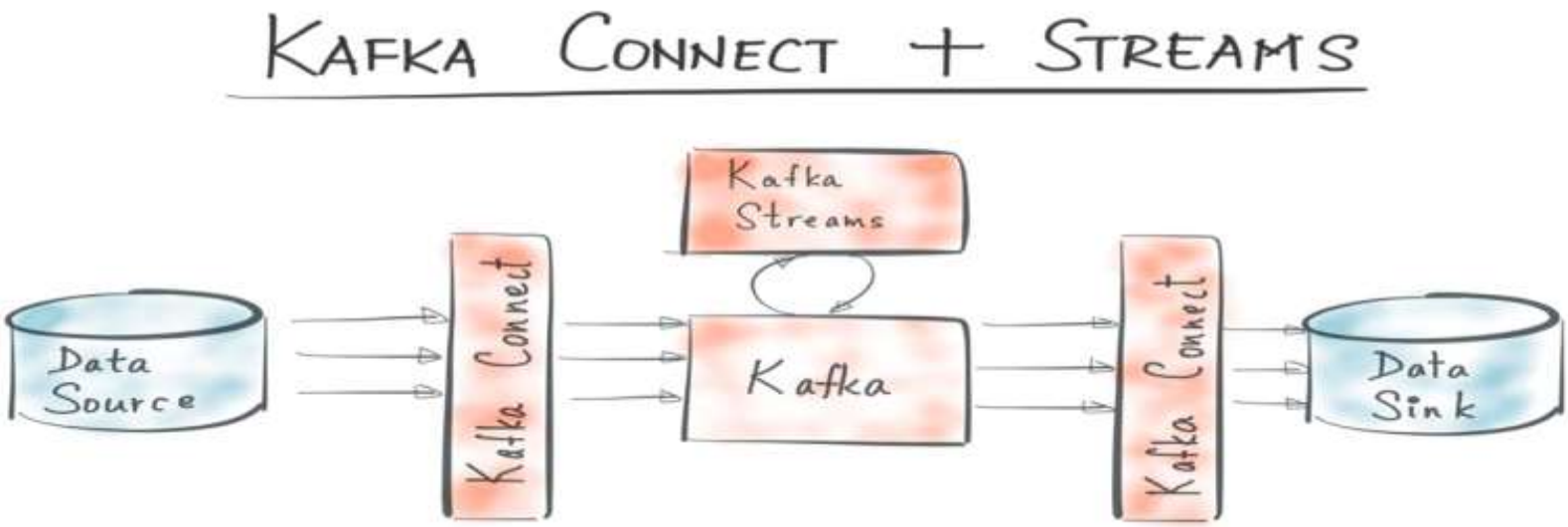
1. Apache Kafka: a Streaming Data Platform

- Most of **what a business does** can be thought as **event streams.** They are in a
  - **Retail system**: orders, shipments, returns, …
  - **Financial system**: stock ticks, orders, …
  - **Web site**: page views, clicks, searches, …
  - **IoT**: sensor readings, …

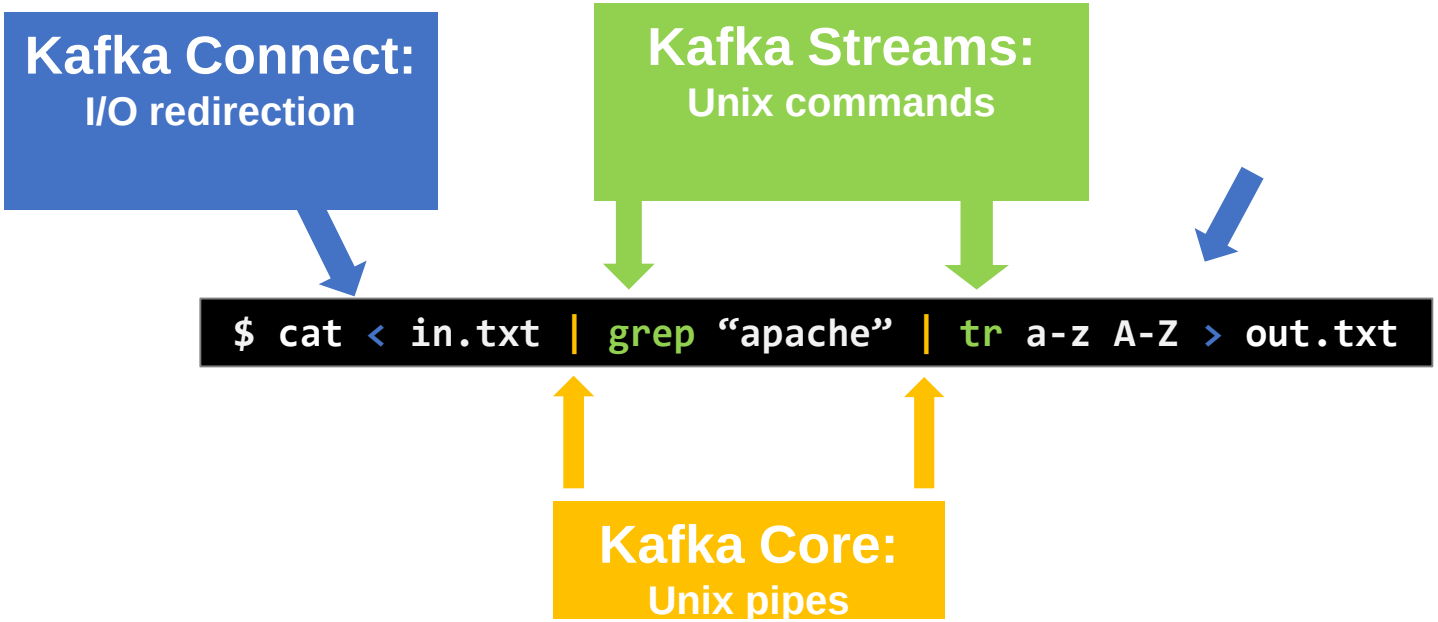
  and so on.

1. Apache Kafka: a Streaming Data Platform

- **Apache Kafka** is an open source streaming data platform (a new category of software!) with **3 major components**:
  1. **Kafka Core**: A **central hub** to **transport** and **store** event streams in real-time.
  2. **Kafka Connect**: A **framework** to **import** event streams from other source data systems into Kafka and **export** event streams from **Kafka** to destination data systems.
  3. **Kafka Streams**: A **Java library** to **process** event streams live as they occur.



1. Apache Kafka: a Streaming Data Platform
Unix Pipelines Analogy



- **Kafka Core: is the distributed, durable equivalent of Unix pipes. Use it to connect and compose your large-scale data applications.**
- **Kafka Streams are the commands of your Unix pipelines. Use it to transform data stored in Kafka.**
- **Kafka Connect is the I/O redirection in your Unix pipelines. Use it to get your data into and out of Kafka.**

2. Overview of Kafka Streams

2.1 Before Kafka Streams?
2.2 What is Kafka Streams?
2.3 Why Kafka Streams?
2.4 What are Kafka Streams key concepts?
2.5 Kafka Streams APIs and code examples?

2.1 Before Kafka Streams?

- **Before Kafka Streams**, to process the data in Kafka you have 4 options:
  - **Option 1: Dot It Yourself (DIY)** – Write your own 'stream processor' using Kafka client libs, typically with a narrower focus.
  - **Option 2: Use a library** such as AkkaStreams-Kafka, also known as Reactive Kafka, RxJava, or Vert.x
  - **Option 3:** Use an **existing open source stream processing framework** such as Apache Storm, Spark Streaming, Apache Flink or Apache Samza for transforming and combining data streams which live in Kafka…
  - **Option 4**: Use an **existing commercial tool for** stream processing with adapter to Kafka such as IBM InfoSphere Streams, TIBCO StreamBase, …
- Each one of the 4 options above of processing data in Kafka has advantages and disadvantages.

2.2 What is Kafka Streams?

- Available since **Apache Kafka 0.10 release in May 2016**, **Kafka Streams** is a **lightweight open source Java library** for building stream processing applications on top of Kafka.
- Kafka Streams is designed to **consume** from & **produce data** to **Kafka topics**.
- It provides a **Low-level API** for building topologies of processors, streams and tables.
- It provides a **High-Level API** for common patterns like filter, map, aggregations, joins, stateful and stateless processing.
- Kafka Streams **inherits operational characteristics** (low latency, elasticity, fault-tolerance, …) from Kafka.
- **A library is simpler than a framework** and is easy to integrate with your existing applications and services!
- Kafka Streams **runs in your application code** and imposes no change in the Kafka cluster infrastructure, or within Kafka.

What is Kafka Streams? Java analogy

1996	1 core	java.lang
2004	multi-core	java.util.concurrent
2016	multi-machine	java.distributed org.apache.kafka.streams

2.3 Why Kafka Streams?

- Processing data in Kafka with Kafka Streams has the following advantages:
  - **No need to run another framework or tool** for stream processing as Kafka Streams is already a library included in Kafka
  - **No need of external infrastructure** beyond Kafka. Kafka is already your cluster!
  - **Operational simplicity** obtained by getting rid of an additional stream processing cluster
  - As a normal library, it is **easier to integrate** with your existing applications and services
  - **Inherits Kafka features such as** fault-tolerance, scalability, elasticity, authentication, authorization
  - **Low barrier to entry**: You can quickly write and run a small-scale proof-of-concept on a single machine

2.4 Wat are Kafka Streams key concepts?

- **KStream** and **KTable** as the two basic abstractions. The distinction between them comes from how the key-value pairs are interpreted:
  - In a **stream**, each key-value is an **independent piece of information**. For example, in a stream of user addresses: Alice -> New York, Bob -> San Francisco, Alice -> Chicago, we know that Alice lived in both cities: New York and Chicago.
  - If the **table** contains a **key-value pair** for the **same key twice, the latter overwrites the mapping**. For example, a table of user addresses with Alice -> New York, Bob -> San Francisco, Alice -> Chicago means that Alice moved from New York to Chicago, not that she lives at both places at the same time.
- There's a **duality between the two concepts**: a stream can be viewed as a table, and a table as a stream. See more on this in the documentation: http://docs.confluent.io/current/streams/concepts.html#duality-of-streams-and-tables

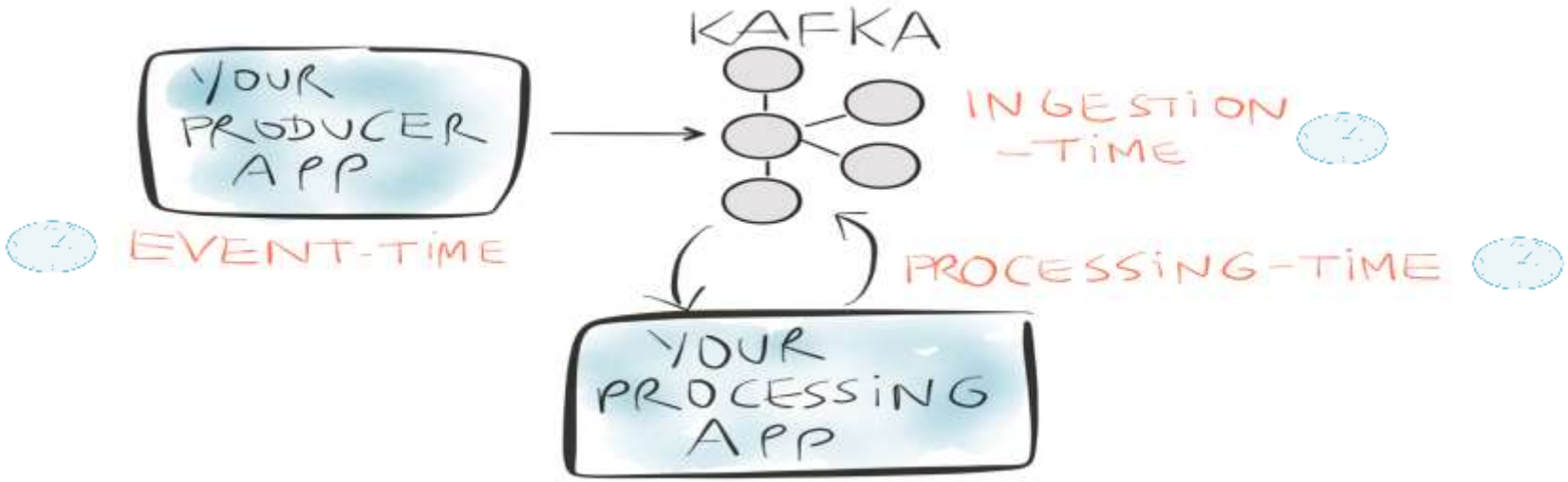
KStream vs KTable

Example	When you need…	then you'd read the Kafka topic into a	so that the topic is interpreted as a	with messages interpreted as
All the cities Alice has ever lived in	All the values of a key	KStream	*record* stream	INSERT (append)
In what city Alice lives right now?	Latest value of a key	KTable	*changelog* stream	UPDATE (overwrite existing)

KStream = immutable log
KTable = mutable materialized view

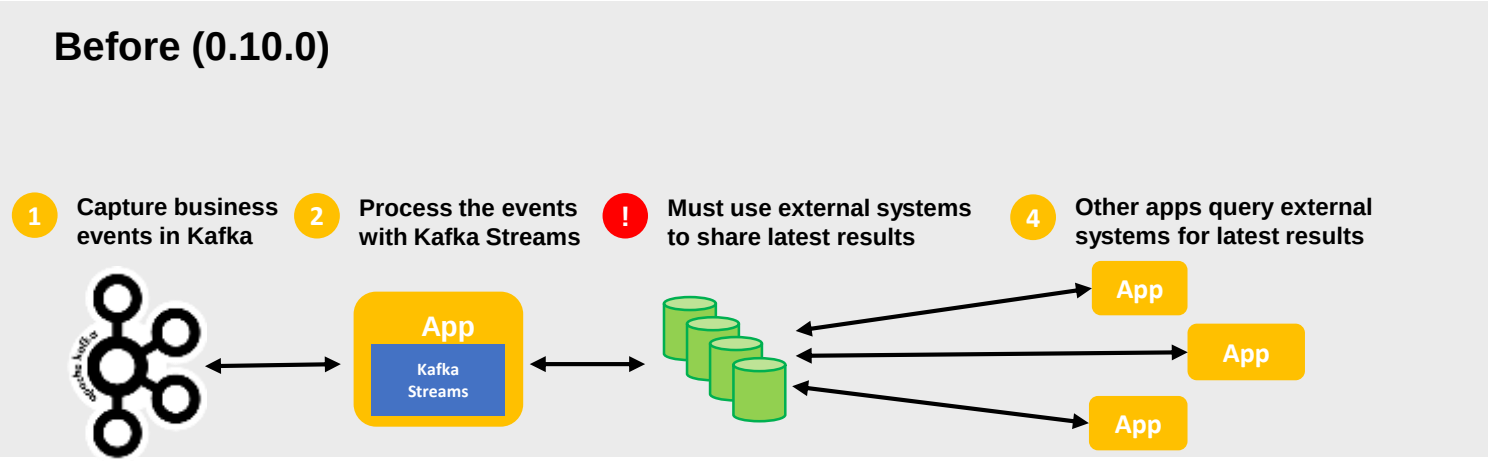
2.4 What are Kafka Streams key concepts?

- **Event Time**: A critical aspect in stream processing is the notion of **time**, and how it is modeled and integrated.
  - **Event time**: The point in time when an **event** or **data record** occurred, i.e. was originally **created** “by the source”.
  - **Ingestion time**: The point in time when an **event** or **data record** is **stored** in a topic partition by a Kafka broker.
  - **Processing time**: The point in time when the **event** or **data record** happens to be **processed** by the stream processing application, i.e. when the record is being consumed.



2.4 What are Kafka Streams key concepts?

➢ Interactive Queries: Local queryable state



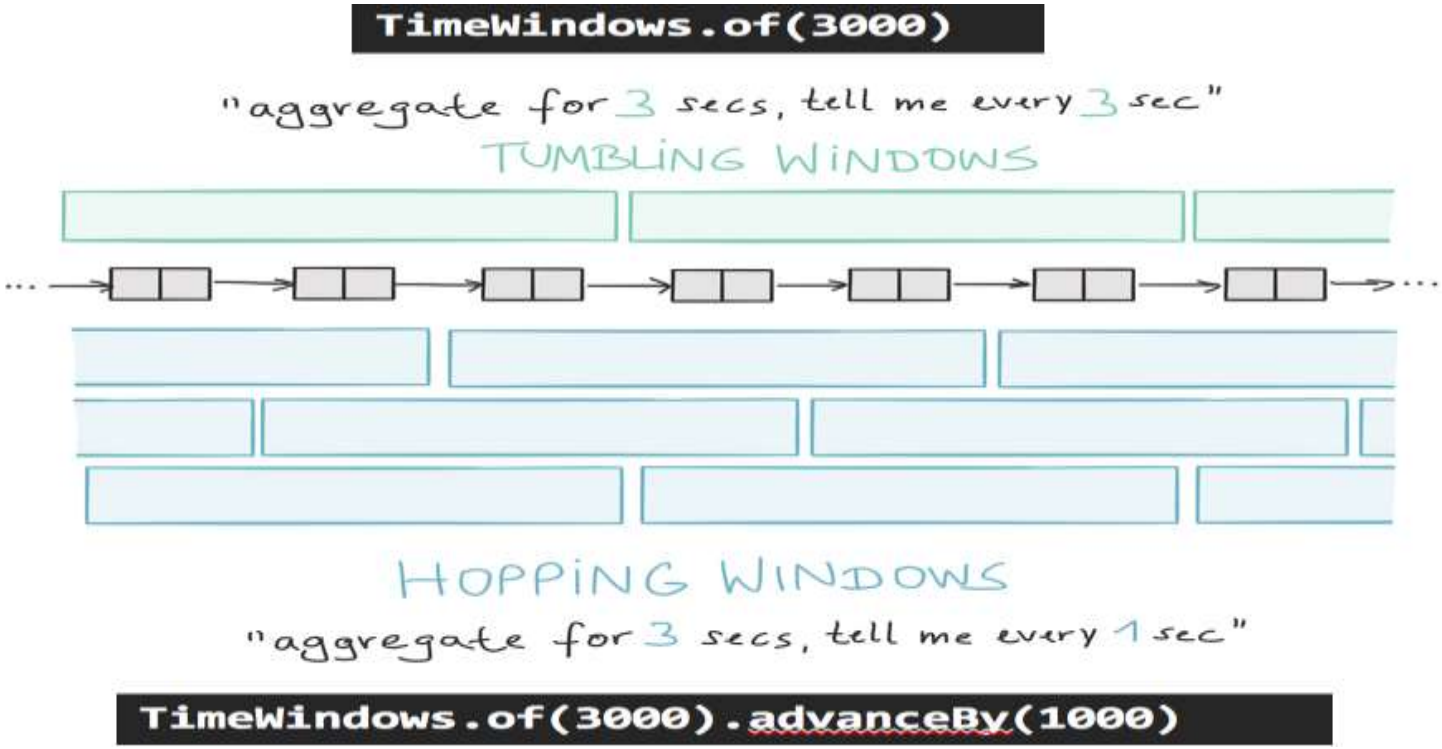
After (0.10.1): simplified, more app-centric architecture

See blogs:

- **Why local state is a fundamental primitive in stream processing**? Jay Kreps, July 31st 2014 https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing
- **Unifying Stream Processing and Interactive Queries in Apache Kafka,** Eno Thereska, October 26th 2016 https://www.confluent.io/blog/unifying-stream-processing-and-interactive-queries-in-apache-kafka/

2.4 What are Kafka Streams key concepts?

- **Windowing**: Windowing lets you control how to group records that have the same key for stateful operations such as **aggregations** or **joins** into so-called windows.



- **More concepts** in Kafka Streams documentation: http://docs.confluent.io/current/streams/concepts.htm

2.5 Kafka Streams APIs and code examples?

API option 1: DSL (high level, declarative)

```
KStream<Integer, Integer> input =
    builder.stream("numbers-topic");

// Stateless computation
KStream<Integer, Integer> doubled =
    input.mapValues(v -> v * 2);

// Stateful computation
KTable<Integer, Integer> sumOfOdds = input
    .filter((k,v) -> v % 2 != 0)
    .selectKey((k, v) -> 1)
    .groupByKey()
    .reduce((v1, v2) -> v1 + v2, "sum-of-odds");
```

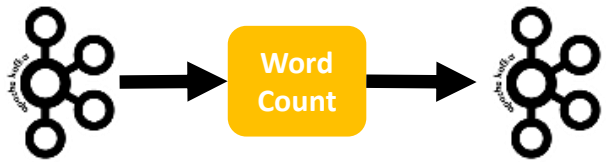
The **preferred API** for most use cases.

The DSL particularly appeals to users:

- familiar with Spark, Flink, Beam
- fans of Scala or functional programming

- If you're used to the functions that real-time processing systems like Apache Spark, Apache Flink, or Apache Beam expose, you'll be right at home in the DSL.
- If you're not, you'll need to spend some time understanding what methods like **map**, **flatMap**, or **mapValues** mean.

Code Example 1: complete app using DSL




```
public static void main(final String[] args) throws Exception {
  Properties config = new Properties();
  config.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-example");
  config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092");
  config.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
  config.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

  KStreamBuilder builder = new KStreamBuilder();
  KStream<String, String> textLines = builder.stream("TextLinesTopic");
  KStream<String, Long> wordCounts = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    .groupBy((key, word) -> word)
    .count("Counts")
    .toStream();
  wordCounts.to(Serdes.String(), Serdes.Long(), "WordsWithCountsTopic");

  KafkaStreams streams = new KafkaStreams(builder, config);
  streams.start();
}
```

App configuration

Define processing (here: WordCount)

Start processing

API option 2: Processor API (low level, imperative)

```
class PrintToConsoleProcessor
    implements Processor<K, V> {

  @Override
  public void init(ProcessorContext context) {}

  @Override
  void process(K key, V value) {
    System.out.println("Got value " + value);
  }

  @Override
  void punctuate(long timestamp) {}

  @Override
  void close() {}
}
```

Full flexibility but **more manual work:**

- The **Processor API** appeals to users:
  - **familiar with Storm, Samza**
    - Still, check out the DSL!
  - requiring **functionality that is not yet available in the DSL**
- Some people have begun using the low-level Processor API to **port their Apache Storm code to Kafka Streams**.

Code Example 2: Complete app using Processor API

```
public PrintToConsoleProcessor implements Processor<K, V> {

    @Override
    public void init(ProcessorContext context) {
      // No initialization needed in this case.
    }

    @Override
    public void process(K key, V value) {
      System.out.println("Received data record with " +
          "key=" + key + ", value=" + value);
    }

    @Override
    public void punctuate(long timestamp) {
      // No periodic actions needed in this case.
    }

    @Override
    public void close() {
      // No shutdown logic needed in this case.
    }
}
```

Startup

Process a record

Periodic action

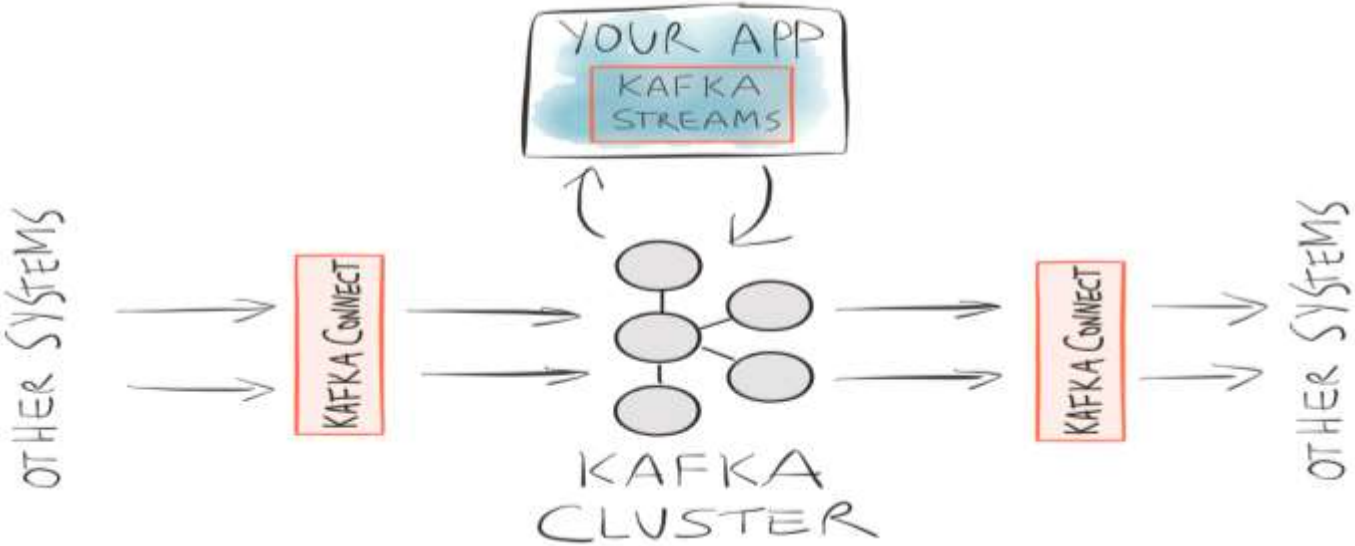
Shutdown

3. Writing, deploying and running your first Kafka Streams application

- **Step 1**: Ensure Kafka cluster is accessible and has data to process
- **Step 2**: Write the application code in Java or Scala
- **Step 3**: Packaging and deploying the application
- **Step 4**: Run the application

Step 1: Ensure Kafka cluster is accessible and has data to process

- Get the **input data into Kafka** via:
  - Kafka Connect (part of Apache Kafka)
  - or your own application that write data into Kafka
  - or tools such as StreamSets, Apache Nifi, ...
- **Kafka Streams** will then be used to **process the data** and write the results back to Kafka.



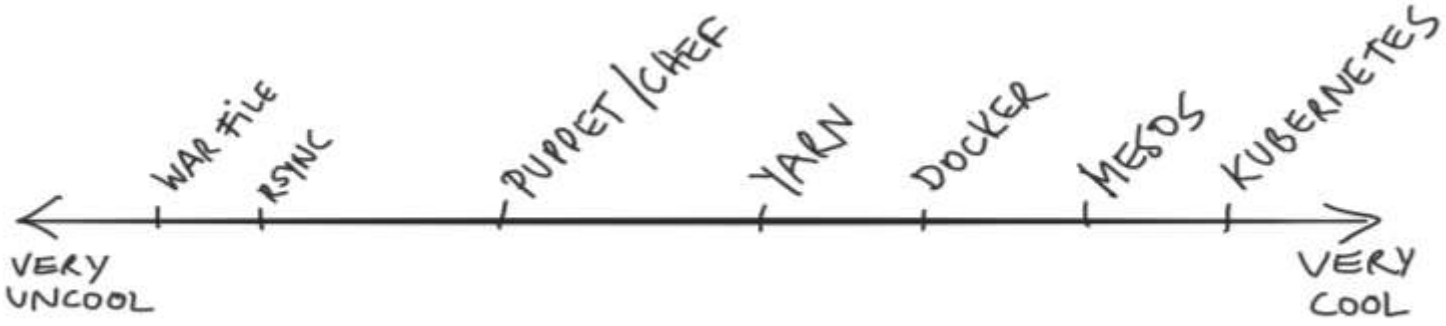
Step 2: Write the application code in Java or Scala

- How to start?
  - Learn from existing code examples: https://github.com/confluentinc/examples
  - Documentation: http://docs.confluent.io/current/streams/
- How do I install Kafka Streams?
  - **There is no "installation"!** It's a Java library. Add it to your client applications like any other Java library.
  - Example adding 'kafka-streams' library using Maven:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
  <version>0.10.2.0</version>
</dependency>
```

Step 3: Packaging and deploying the application

- How do you package and deploy your Kafka Streams apps?
  - **Whatever works for you!** Stick to what you/your company think is the best way for deploying and packaging a java application.
  - Kafka Streams integrates well with what you already use because an application that uses **Kafka Streams is a normal Java application**.



Step 4: Run the application

- **You don't need to install a cluster** as in other stream processors (Storm, Spark Streaming, Flink, …) and **submit jobs to it**!
- **Kafka Streams runs as part of your client applications**, it does not run in the Kafka brokers.
- In production, **bundle as fat jar**, then `java -cp my-fatjar.jar com.example.MyStreamsApp`
  http://docs.confluent.io/current/streams/developer-guide.html#running-a-kafka-streams-application
- TIP: During development from your IDE or from CLI, the '**Kafka Streams Application Reset Tool**', available since Apache Kafka 0.10.0.1, is great for playing around.
  https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool

Example: complete app, ready for production at large-scale!

```
public static void main(String[] args) throws Exception {
    Properties config = new Properties();
    config.put(StreamsConfig.JOB_ID_CONFIG, "wordcount-lambda-example");
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    config.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "localhost:2181");
    config.put(StreamsConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    config.put(StreamsConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    config.put(StreamsConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    config.put(StreamsConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);

    final Serializer<String> stringSerializer = new StringSerializer();
    final Deserializer<String> stringDeserializer = new StringDeserializer();
    final Serializer<Long> longSerializer = new LongSerializer();
    final Deserializer<Long> longDeserializer = new LongDeserializer();

    KStreamBuilder builder = new KStreamBuilder();
    KStream<String, String> textLines = builder.stream(stringDeserializer, stringDeserializer, "TextLinesTopic");

    KStream<String, Long> wordCounts = textLines
        .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
        .map((key, value) -> new KeyValue<>(value, value))
        .countByKey(stringSerializer, longSerializer, stringDeserializer, longDeserializer, "Counts")
        .toStream();
    wordCounts.to("WordsWithCountsTopic", stringSerializer, longSerializer);

    KafkaStreams streams = new KafkaStreams(builder, config);
    streams.start();
}
```

4. Code and Demo of an end-to-end Kafka-based Streaming Data Application

4.1 Scenario of this demo
4.2 Architecture of this demo
4.3 Setup of this demo
4.4 Results of this demo
4.5 Stopping the demo!

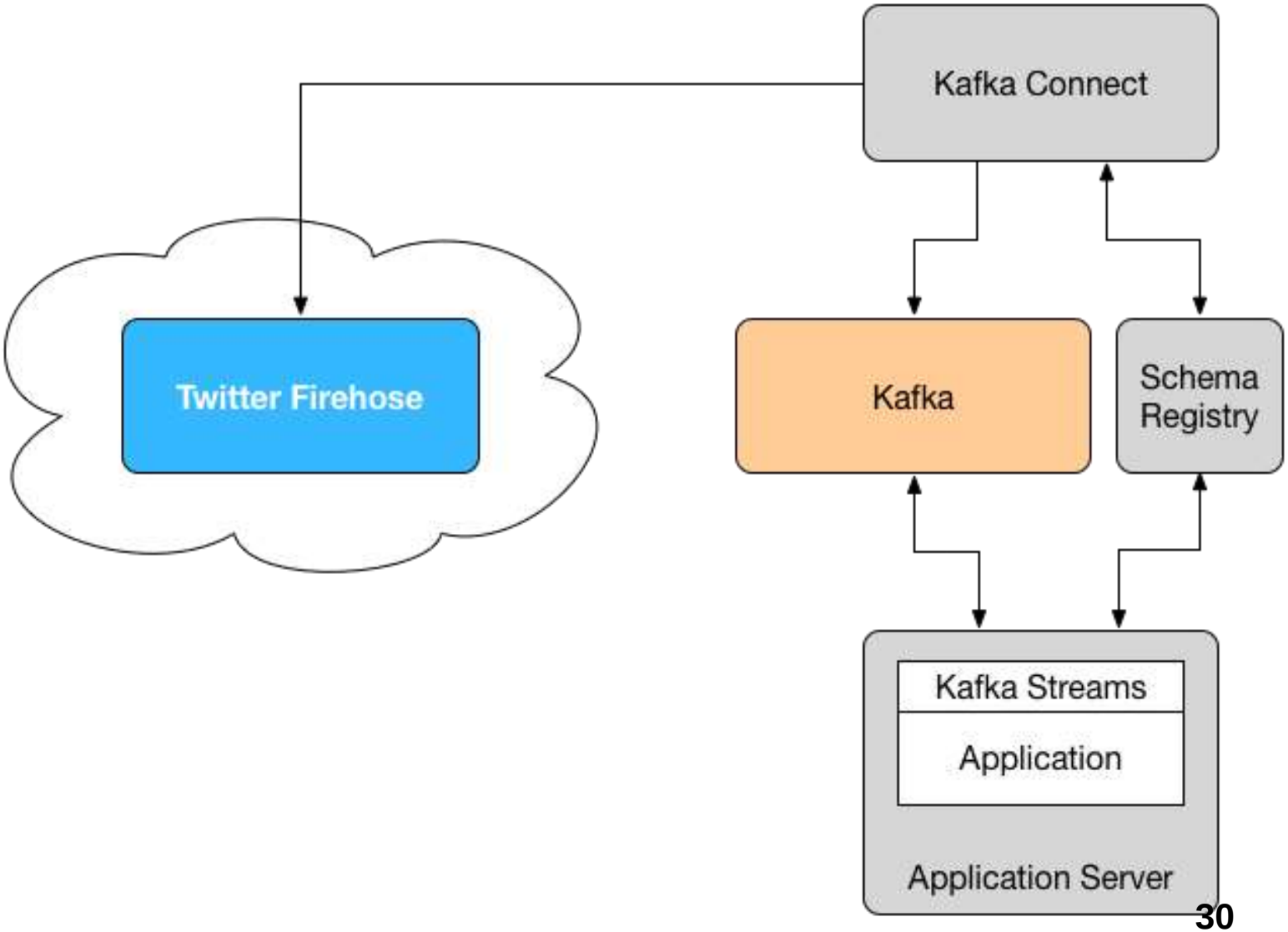
4.1. Scenario of this demo

- This demo consists of:
  - reading live stream of data (tweets) from Twitter using **Kafka Connect** connector for Twitter
  - storing them in Kafka broker leveraging **Kafka Core** as publish-subscribe message system.
  - performing some basic stream processing on tweets in Avro format from a Kafka topic using **Kafka Streams** library to do the following:
    - **Raw word count** - every occurrence of individual words is counted and written to the **topic** ***wordcount*** (a predefined list of stopwords will be ignored)
    - **5-Minute word count** - words are counted per 5 minute window and every word that has more than 3 occurrences is written to the **topic** ***wordcount5m***
    - **Buzzwords** - a list of special interest words can be defined and those will be tracked in the **topic** ***buzzwords***

4.1. Scenario of this demo

- This demo is adapted from one that was given by Sönke Liebau on July 27th 2016 from OpenCore, Germany. See blog entry titled: 'Processing Twitter Data with Kafka Streams" http://www.opencore.com/blog/2016/7/kafka-streams-demo/ and related code at GitHub https://github.com/opencore/kafkastreamsdemo
- In addition:
  - I'm using a **Docker container** instead of the confluent platform they are providing with their Virtual Machine defined in Vagrant.
  - I'm also using **Kafka Connect UI** from Landoop for easy and fast configuration of Twitter connector and also other Landoop's Fast Data Web UIs.

4.2. Architecture of this demo



4.3. Setup of this demo

Step 1: Setup your Kafka Development Environment

Step 2: Get twitter credentials to connect to live data

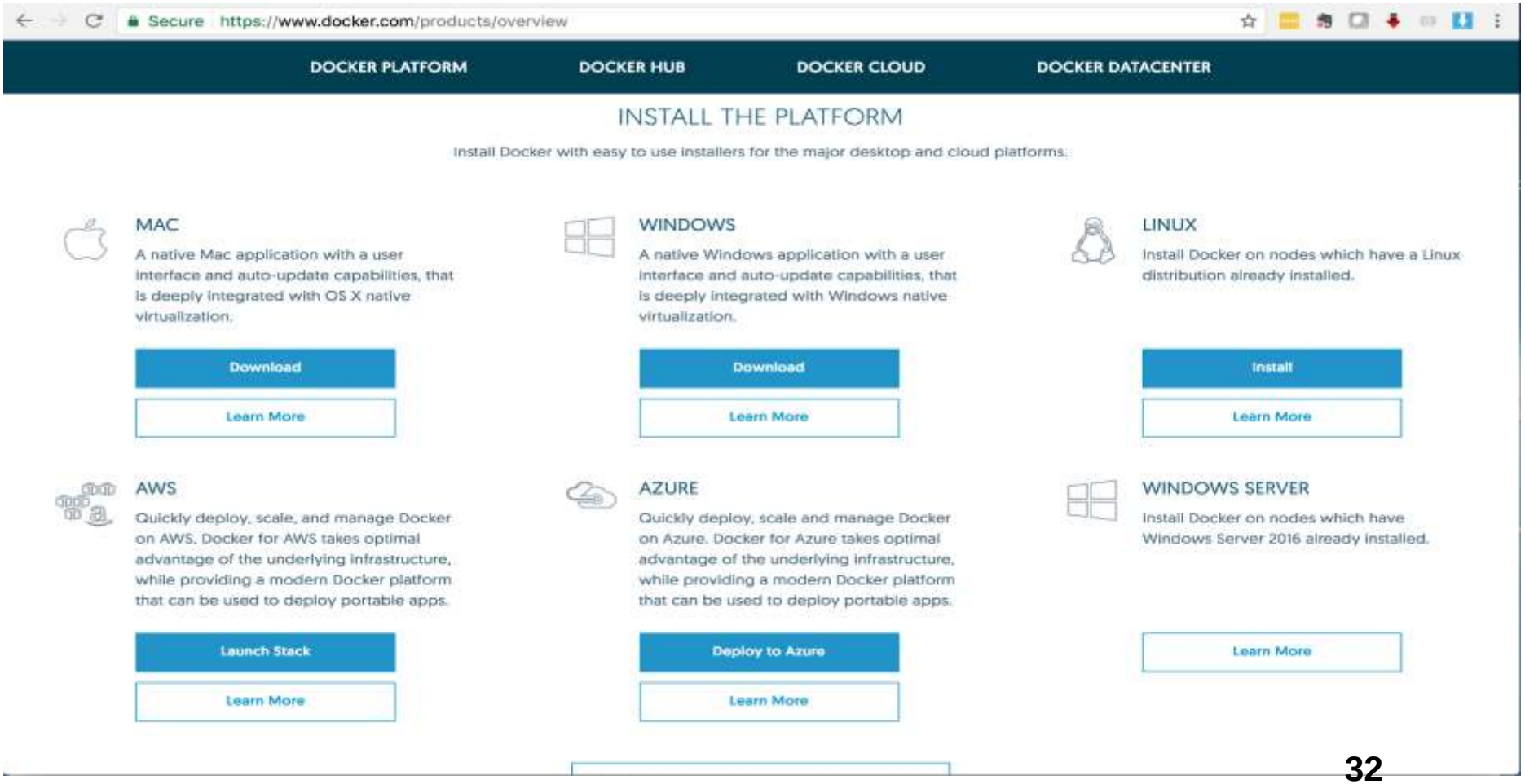
Step 3: Get twitter live data into Kafka broker

Step 4: Write and test the application code in Java

Step 5: Run the application

Step 1: Setup your Kafka Development Environment

- The easiest way to get up and running quickly is to use a **Docker container** with all components needed.
- **First,** install Docker on your desktop or on the cloud https://www.docker.com/products/overview and start it



Step 1: Setup your Kafka Development Environment

- **Second**, install **Fast-data-dev**, a **Docker image for Kafka** developers which is packaging:
  - **Kafka broker**
  - **Zookeeper**
  - **Open source version of the Confluent Platform** with its Schema registry, REST Proxy and bundled connectors
  - Certified DataMountaineer **Connectors** (ElasticSearch, Cassandra, Redis, ..)
  - Landoop's Fast Data **Web UIs** : schema-registry, kafka-topics, kafka-connect.
  - Please note that **Fast Data Web UIs are licensed under BSL**. You should contact Landoop if you plan to use them on production clusters with more than 4 nodes.

by executing the command below, while Docker is running and you are connected to the internet:

```
docker run --rm -it --net=host landoop/fast-data-dev
```

- If you are on Mac OS X, you have to expose the ports instead:

```
docker run --rm -it \
        -p 2181:2181 -p 3030:3030 -p 8081:8081 \
        -p 8082:8082 -p 8083:8083 -p 9092:9092 \
        -e ADV_HOST=127.0.0.1 \
         landoop/fast-data-dev
```

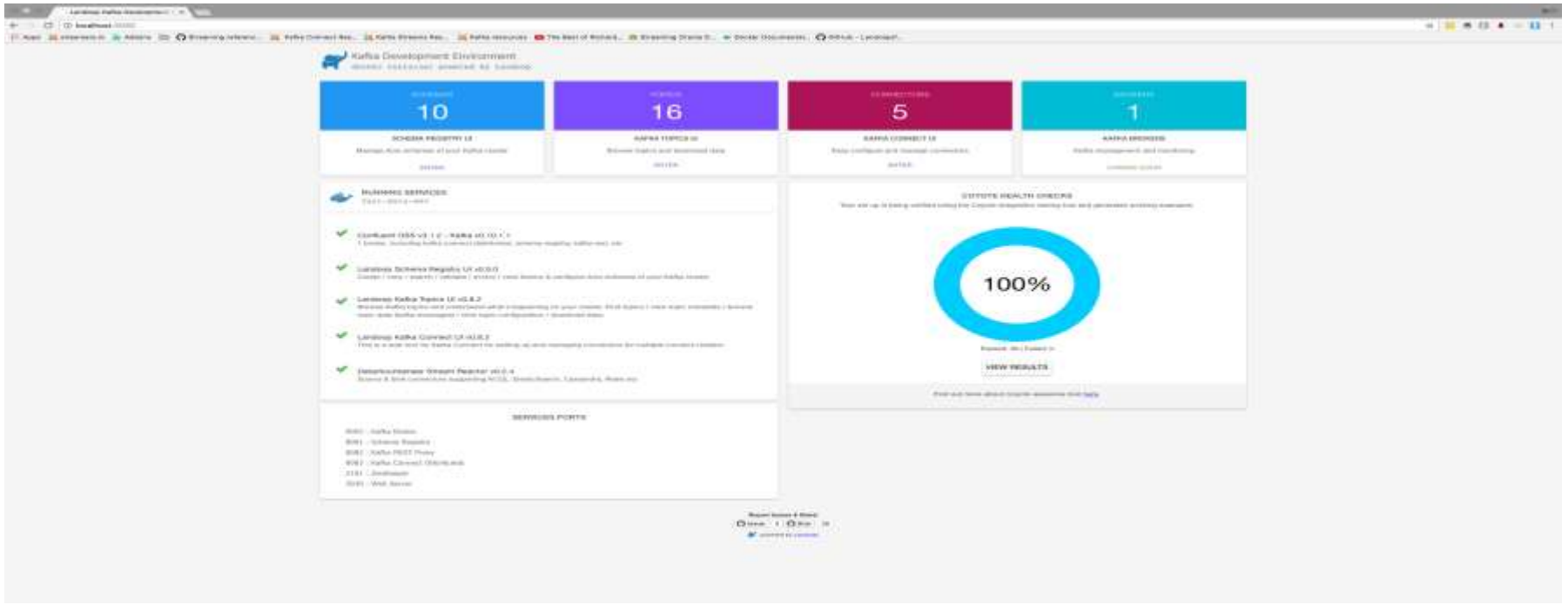
- This will download the fast-data-dev Docker image from the Dock Hub. https://hub.docker.com/r/landoop/fast-data-dev/
- Future runs will use your local copy.
- More details about Fast-data-dev docker image https://github.com/Landoop/fast-data-dev

Step 1: Setup your Kafka Development Environment

- **Points of interest:**
  - the **-p** flag is used to publish a network port. Inside the container, ZooKeeper listens at 2181 and Kafka at 9092. If we don't publish them with -p, they are not available outside the container, so we can't really use them.
  - the **–e** flag sets up environment variables.
  - the last part specifies the image we want to run: **landoop/fast-data-dev**
  - Docker will realize it doesn't have the landoop/fast-data-dev image locally, so it will first download it.
- That's it.
  - Your **Kafka Broker** is at localhost:9092,
  - your **Kafka REST Proxy** at localhost:8082,
  - your **Schema Registry** at localhost:8081,
  - your **Connect Distributed** at localhost:8083,
  - your **ZooKeeper** at localhost:2181

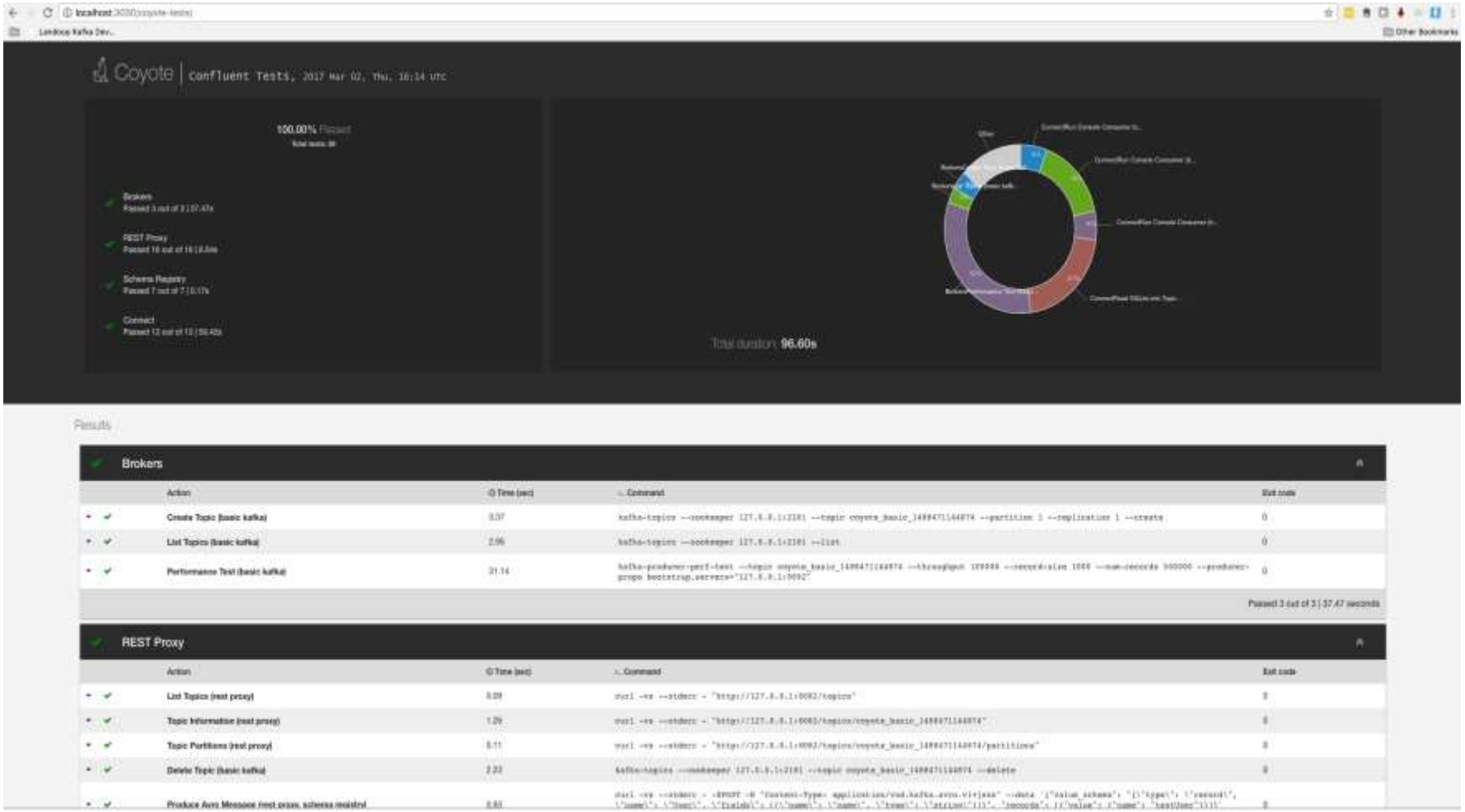
Step 1: Setup your Kafka Development Environment

- At http://localhost:3030, you will find **Landoop's Web UIs** for:
  - Kafka Topics
  - Schema Registry
  - as well as a integration test report for connectors & infrastructure using Coyote. https://github.com/Landoop/coyote
- If you want to stop all services and remove everything, simply hit **Control+C**.



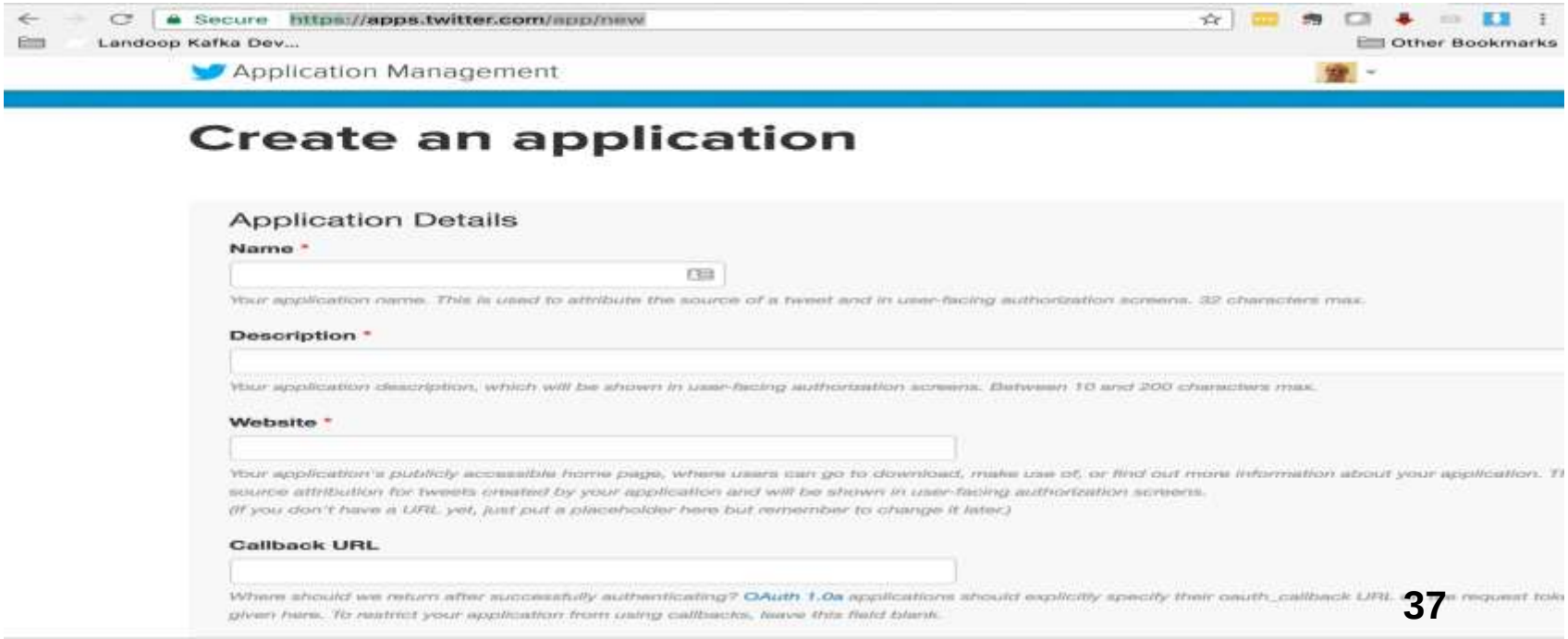
Step 1: Setup your kafka Development Environment

- Explore Integration test results at http://localhost:3030/coyote-tests/



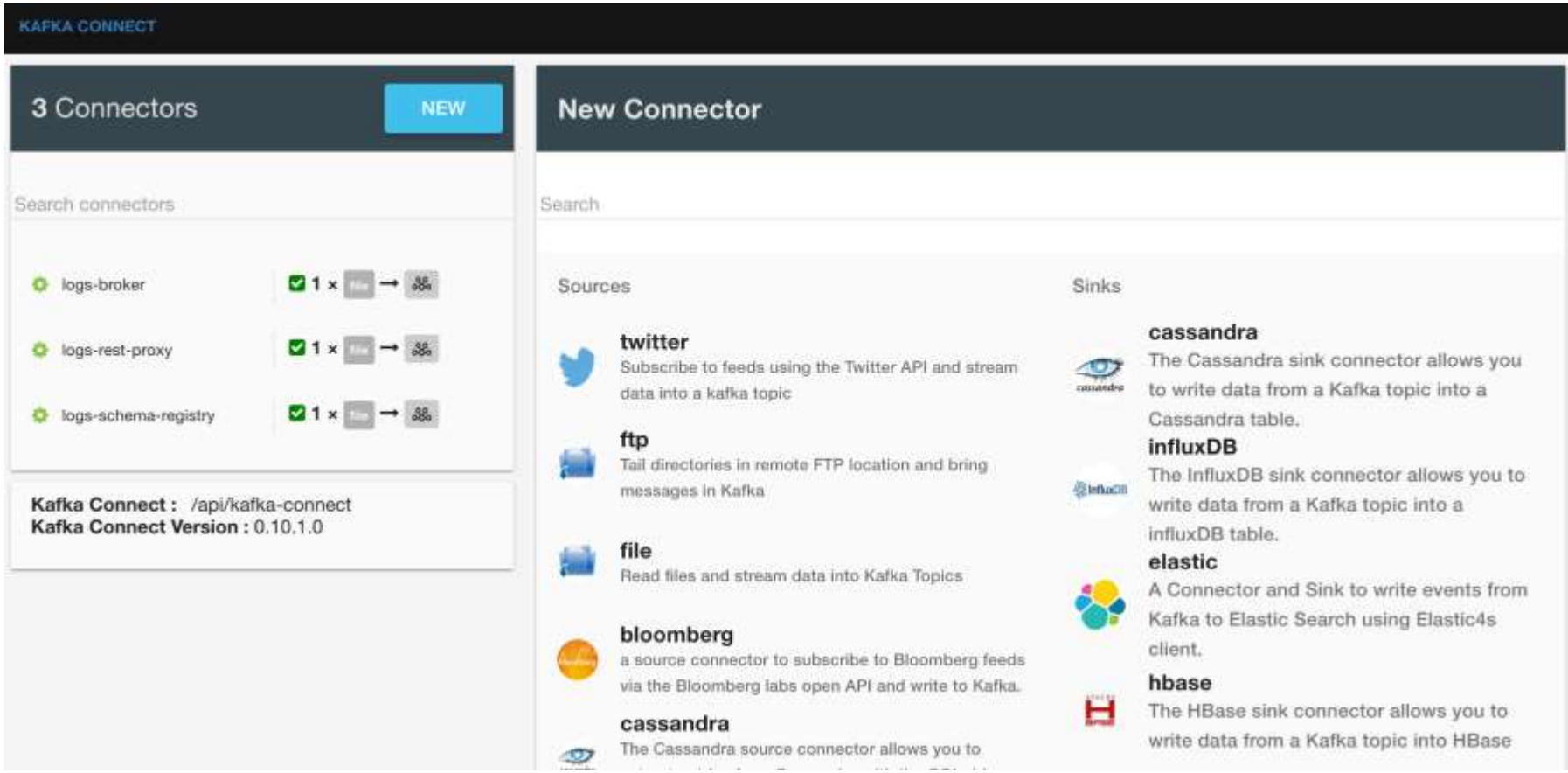
Step 2: Get twitter credentials to connect to live data

- Now that our **single-node Kafka cluster is fully up and running**, we can proceed to preparing the input data:
  - First you need to **register an application with Twitter.**
  - Second, once the application is created **copy the Consumer key and Consumer Secret.**
  - Third, **generate the Access Token Access and Secret Token** required to give your twitter account access to the new application
- Full instructions are here: **https://apps.twitter.com/app/new**



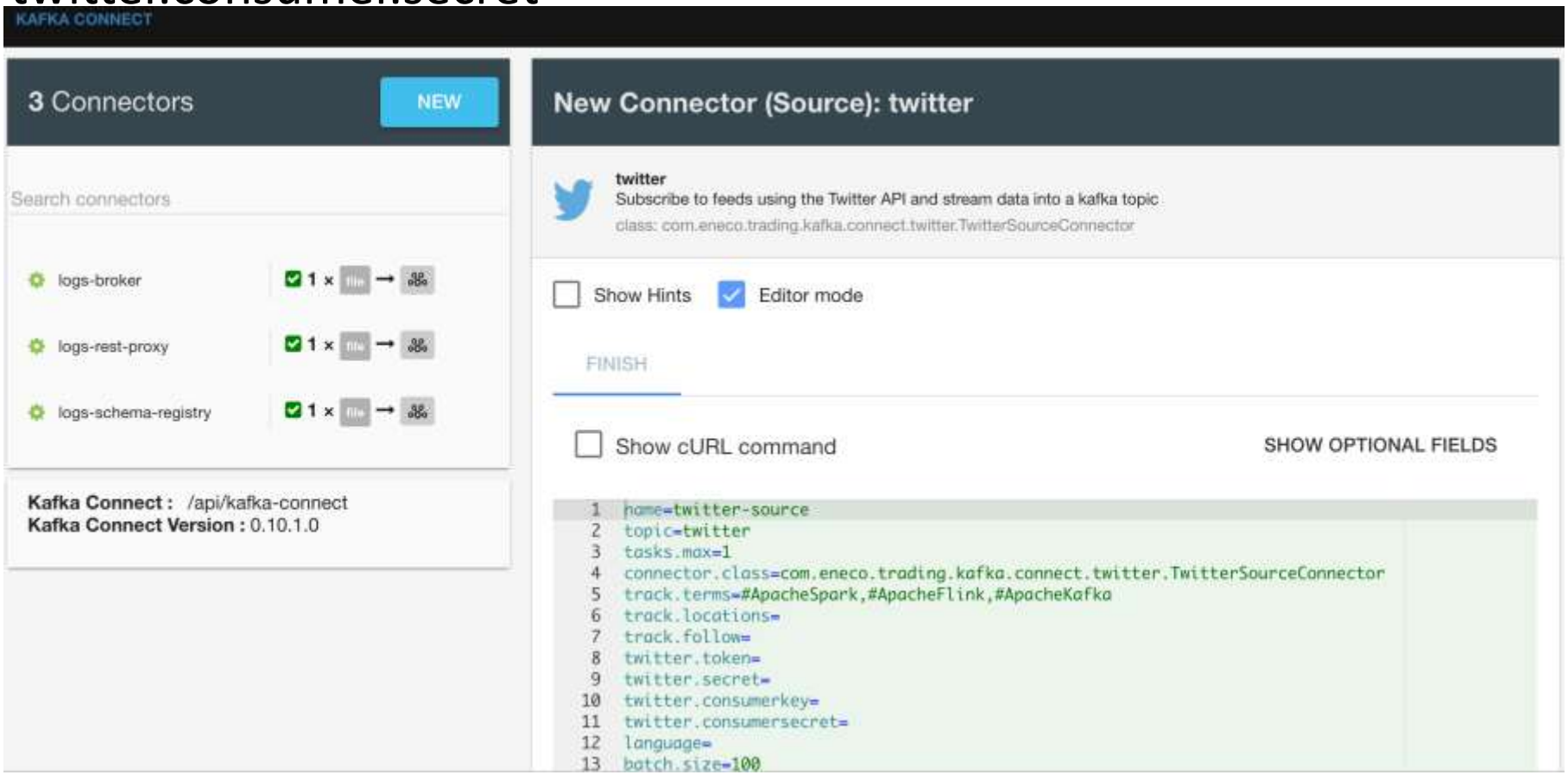
Step 3: Get twitter live data into Kafka broker

- First, **create a new Kafka Connect for Twitter**



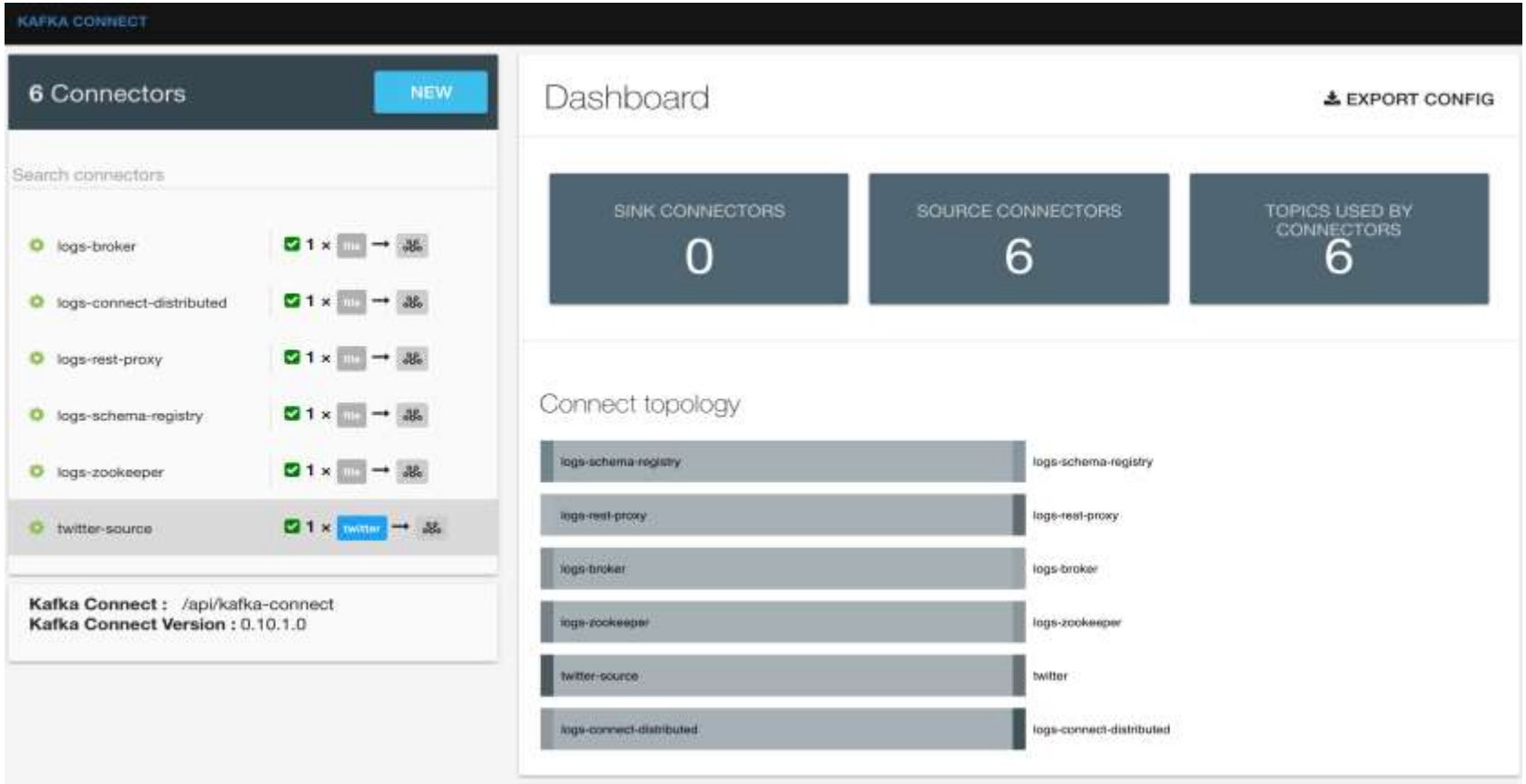
Step 3: Get twitter live data into Kafka broker

- Second, **configure this Kafka Connect for Twitter** to write to the **topic *twitter*** by entering your own track.terms and also the values of twitter.token, twitter.secret, twitter.comsumerkey and twitter.consumer.secret



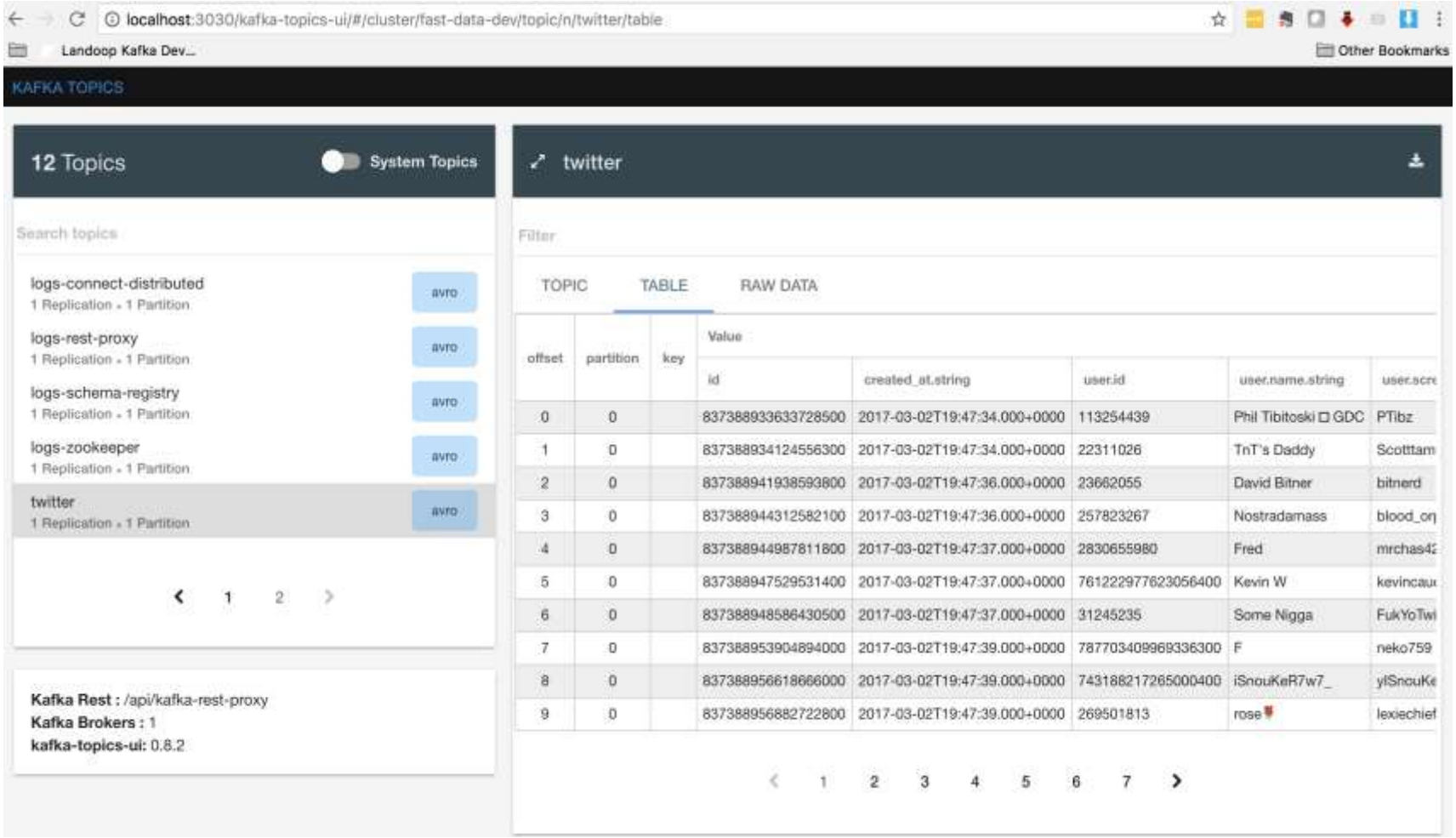
Step 3: Get twitter live data into Kafka broker

- **Kafka Connect for Twitter** is now configured to write data to the **topic *twitter***.



Step 3: Get twitter live data into Kafka broker

➢Data is now being written to the **topic *twitter***.



Step 4: Write and test the application code in Java

- Instead of writing our own code for this demo, we will be leveraging an existing code from GitHub by Sonke Liebau: https://github.com/opencore/kafkastreamsdemo

Step 4: Write and test the application code in Java

- **git clone https://github.com/opencore/kafkastreamsdemo**

```
Streamera:~ sbaltagi$ git clone https://github.com/opencore/kafkastreamsdemo
Cloning into 'kafkastreamsdemo'...
remote: Counting objects: 34, done.
remote: Total 34 (delta 0), reused 0 (delta 0), pack-reused 34
Unpacking objects: 100% (34/34), done.
Streamera:~ sbaltagi$
```

- Edit the buzzwords.txt file with your own works and probably one of the twitter terms that you are watching live:

buzzwords.txt — Edited

```
chicago
kafka
meetup
streaming
```

```
bash
total 0
drwxr-xr-x   3 sbaltagi   staff   102 Mar   2 14:00 java
drwxr-xr-x   5 sbaltagi   staff   170 Mar   2 14:00 resources
Streamera:main sbaltagi$ cd resources/
Streamera:resources sbaltagi$ ls -l
total 24
-rw-r--r--   1 sbaltagi   staff    51 Mar   2 14:00 buzzwords.txt
-rw-r--r--   1 sbaltagi   staff   332 Mar   2 14:00 log4j.properties
-rw-r--r--   1 sbaltagi   staff   959 Mar   2 14:00 stopwords.txt
Streamera:resources sbaltagi$ open -e buzzwords.txt
Streamera:resources sbaltagi$
```

Step 5: Run the application

- The next step is to run the Kafka Streams application that processes twitter data.
- First, install Maven http://maven.apache.org/install.html
- Then, **compile the code into a fat jar with Maven.**

$ *mvn package*

```
Streamera:kafkastreamsdemo sbaltagi$ pwd
/Users/sbaltagi/kafkastreamsdemo
Streamera:kafkastreamsdemo sbaltagi$ ls -l
total 40
-rw-r--r--  1 sbaltagi  staff  11357 Mar  2 14:00 LICENSE
-rw-r--r--  1 sbaltagi  staff   2135 Mar  2 14:00 README.md
-rw-r--r--  1 sbaltagi  staff   2910 Mar  2 14:00 pom.xml
drwxr-xr-x  3 sbaltagi  staff    102 Mar  2 14:00 src
Streamera:kafkastreamsdemo sbaltagi$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building KafkaStreamsDemo 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
Downloading: http://packages.confluent.io/maven/org/apache/kafka/kafka-streams/0.10.0.0/kafka-s
treams-0.10.0.0.pom
```

```
Component: org.apache.maven.artifact.handler.ArtifactHandlerejb-client is already defined. Skipping.
Component: org.apache.maven.artifact.handler.ArtifactHandlerrar is already defined. Skipping.
Component: org.apache.maven.artifact.handler.ArtifactHandlerpar is already defined. Skipping.
Component: org.apache.maven.artifact.handler.ArtifactHandlerejb3 is already defined. Skipping.
Component: org.apache.maven.artifact.factory.ArtifactFactory is already defined. Skipping.
Component: org.apache.maven.artifact.resolver.ArtifactCollector is already defined. Skipping.
Component: org.apache.maven.plugin.registry.MavenPluginRegistryBuilder is already defined. Skipping.
Component: org.apache.maven.shared.io.download.DownloadManagerdefault is already defined. Skipping.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 11.475 s
[INFO] Finished at: 2017-03-02T14:24:58-06:00
[INFO] Final Memory: 37M/422M
[INFO] ------------------------------------------------------------------------
Streamera:kafkastreamsdemo sbaltagi$
```

Step 5: Run the application

- Two jar files will be created in the **target** folder:
  1. KafkaStreamsDemo-1.0-SNAPSHOT.jar – Only your project classes
  2. KafkaStreamsDemo-1.0-SNAPSHOT-**jar-with-dependencies**.jar – Project and dependency classes in a single jar.

```
Streamera:kafkastreamsdemo sbaltagi$ pwd
/Users/sbaltagi/kafkastreamsdemo
Streamera:kafkastreamsdemo sbaltagi$ ls -l
total 40
-rw-r--r--  1 sbaltagi  staff  11357 Mar  2 14:00 LICENSE
-rw-r--r--  1 sbaltagi  staff   2135 Mar  2 14:00 README.md
-rw-r--r--  1 sbaltagi  staff   2910 Mar  2 14:00 pom.xml
drwxr-xr-x  3 sbaltagi  staff    102 Mar  2 14:00 src
drwxr-xr-x  8 sbaltagi  staff    272 Mar  2 14:24 target
Streamera:kafkastreamsdemo sbaltagi$ cd target/
Streamera:target sbaltagi$ ls -l
total 31672
-rw-r--r--  1 sbaltagi  staff  16203673 Mar  2 14:24 KafkaStreamsDemo-1.0-SNAPSHOT-jar-with-dependencies.jar
-rw-r--r--  1 sbaltagi  staff     11537 Mar  2 14:24 KafkaStreamsDemo-1.0-SNAPSHOT.jar
drwxr-xr-x  2 sbaltagi  staff        68 Mar  2 14:24 archive-tmp
drwxr-xr-x  6 sbaltagi  staff       204 Mar  2 14:24 classes
drwxr-xr-x  3 sbaltagi  staff       102 Mar  2 14:24 generated-sources
drwxr-xr-x  3 sbaltagi  staff       102 Mar  2 14:24 maven-archiver
Streamera:target sbaltagi$
```

Step 5: Run the application

- Then

java -cp target/KafkaStreamsDemo-1.0-SNAPSHOT-**jar-with-dependencies**.jar com.opencore.sapwebinarseries.KafkaStreamsDemo

- **TIP**: During development: from your IDE, from CLI … Kafka Streams Application Reset Tool, **available since** Apache Kafka 0.10.0.1, is great for playing around.
  https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool

4.4. Results of this demo

- Once the above is running, the following topics will be populated with data :
  - **Raw word count** - Every occurrence of individual words is counted and written to the topic *wordcount* (a predefined list of stopwords will be ignored)
  - **5-Minute word count** - Words are counted per 5 minute window and every word that has more than three occurrences is written to the topic *wordcount5m*
  - **Buzzwords** - a list of special interest words can be defined and those will be tracked in the topic *buzzwords* - the list of these words can be defined in the file *buzzwords.txt*

4.4. Results of this demo

- **Accessing the data generated by the code** is as simple as starting a **console consumer** which is shipped with Kafka
- You need first to enter the container to use any tool as you like: ***docker run --rm -it --net=host landoop/fast-data-dev bash***
- Use the following command to check the topics:
  - ***kafka-console-consumer --topic wordcount --new-consumer --bootstrap-server 127.0.0.1:9092 --property print.key=true***
  - ***kafka-console-consumer --topic wordcount5m --new-consumer --bootstrap-server 127.0.0.1:9092 --property print.key=true***
  - ***kafka-console-consumer --topic buzzwords --new-consumer --bootstrap-server 127.0.0.1:9092 --property print.key=true***

4.4. Results of this demo

5. docker

docker

```
RT @SandraTXAS: Mayor Rahm Emanuel and Democrats allow Chicago citizens to be murdered in their neighborhoods for decades, yet blam… en
SandraTXAS
null    2017-03-02T21:28:48.000+0000Marsha GanzMarshaGanz$Ft. Lauderdale, FLRT @Science4UsSays: Bwhahahahahaha #thursdaythoughts #ICE1
7 #Chicago #davinci  https://t.co/89NqAgfHwn @MarshaGanz https://t.co/324aVsDUXot thursdaythoughts
ICE17Chicagodavinci4pic.twitter.com/324aVsDUXohttps://twitter.com/Science4UsSays/status/837413032342999040/photo/1
photo.https://t.co/324aVsDUXobuff.ly/2lyHLAH,http://buff.ly/2lyHLAH.https://t.co/89NqAgfHwn
                                                                                                "Jenna and SciefusScience4UsSaysarsha GanzMarshaGan
z
null    2017-03-02T21:29:10.000+0000illy BlokeeBayDownUnder8Australian Capital Territoryjhicago girl writes 'I hate this life' d
ays before death https://t.co/kJ6Zmq7FlF https://t.co/jzvOWF0qwyen4pic.twitter.com/jzvOWF0qwyhttps://twitter.com/eBayDownUnder/status/837414503738322
944/photo/1
photo.https://t.co/jzvOWF0qwydlvr.it/NWnpJ3*http://dlvr.it/NWnpJ3.https://t.co/kJ6Zmq7FlF
```

docker

```
https://t.co/89nqagfhwn 2
@marshaganz     2
https://t.co/324avsduxo 2
chicago 2263
girl    155
writes  51
'i      9
hate    134
life'   1
days    38
death   18
https://t.co/kj6zmq7flf 1
https://t.co/jzvowf0qwy 1
```

docker

```
Thu Mar 02 15:30:00 CST 2017-democrats  5
Thu Mar 02 15:30:00 CST 2017-allow      5
Thu Mar 02 15:30:00 CST 2017-chicago    99
Thu Mar 02 15:30:00 CST 2017-citizens   5
Thu Mar 02 15:30:00 CST 2017-murdered   5
Thu Mar 02 15:30:00 CST 2017-neighborhoods    5
Thu Mar 02 15:30:00 CST 2017-decades,   5
Thu Mar 02 15:30:00 CST 2017-yet        5
Thu Mar 02 15:30:00 CST 2017-blam…      5
Thu Mar 02 15:30:00 CST 2017-rt 124
Thu Mar 02 15:30:00 CST 2017-#chicago   17
Thu Mar 02 15:30:00 CST 2017-chicago    100
Thu Mar 02 15:30:00 CST 2017-days       3
```

docker

```
Thu Mar 02 15:30:00 CST 2017-chicago    91
Thu Mar 02 15:30:00 CST 2017-chicago    92
Thu Mar 02 15:30:00 CST 2017-chicago    93
Thu Mar 02 15:30:00 CST 2017-chicago    94
Thu Mar 02 15:30:00 CST 2017-chicago    95
Thu Mar 02 15:30:00 CST 2017-meetup     8
Thu Mar 02 15:30:00 CST 2017-chicago    96
Thu Mar 02 15:30:00 CST 2017-chicago    97
Thu Mar 02 15:30:00 CST 2017-chicago    98
Thu Mar 02 15:30:00 CST 2017-meetup     9
Thu Mar 02 15:30:00 CST 2017-chicago    99
Thu Mar 02 15:30:00 CST 2017-#chicago   17
Thu Mar 02 15:30:00 CST 2017-chicago    100
```

4.5. Stopping the demo!

- To stop the Kafka Streams Demo application:
  - **$ ps – A | grep java**
  - **$ kill -9 PID**

```
1. tmux
Streamera:~ sbaltagi$ ps -A |grep java
35532 ttys007    0:00.00 grep java
35253 ttys013    0:29.90 /usr/bin/java -cp target/KafkaStreamsDemo-1.0-SNAPSHOT-
jar-with-dependencies.jar com.opencore.sapwebinarseries.KafkaStreamsDemo
Streamera:~ sbaltagi$ kill -9 35253
Streamera:~ sbaltagi$ ps -A |grep java
35536 ttys007    0:00.00 grep java
Streamera:~ sbaltagi$
```

- If you want to **stop all services in fast-data-dev Docker** image and remove everything, simply hit **Control+C.**

5. Where to go from here for further learning?

- **Kafka Streams code examples**
  - Apache Kafka https://github.com/apache/kafka/tree/trunk/streams/examples/src/main/java/org/apache/kafka/streams/examples
  - Confluent https://github.com/confluentinc/examples/tree/master/kafka-streams
- **Source Code** https://github.com/apache/kafka/tree/trunk/streams
- **Kafka Streams Java docs** http://docs.confluent.io/current/streams/javadocs/index.html
- **First book on Kafka Streams (MEAP)**
  - Kafka Streams in Action https://www.manning.com/books/kafka-streams-in-action
- **Kafka Streams download**
  - Apache Kafka https://kafka.apache.org/downloads
  - Confluent Platform http://www.confluent.io/download

5. Where to go from here for further learning?

- **Kafka Users mailing list** https://kafka.apache.org/contact

- **Kafka Streams at Confluent Community on Slack**
  - https://confluentcommunity.slack.com/messages/streams/

- **Free ebook**:
  - Making Sense of Stream processing by Martin Klepmann https://www.confluent.io/making-sense-of-stream-processing-ebook-download/

- **Kafka Streams documentation**
  - Apache Kafka http://kafka.apache.org/documentation/streams
  - Confluent http://docs.confluent.io/3.2.0/streams/

- **All web resources related to Kafka Streams**

http://sparkbigdata.com/component/tags/tag/69-kafka-streams

Thank you!

Let’s keep in touch!

@SlimBaltagi

https://www.linkedin.com/in/slimbaltagi

sbaltagi@gmail.com