1. **Apache Kafka: a Streaming Data Platform**

➢ Most of **what a business does** can be thought as **event streams.** They are in a

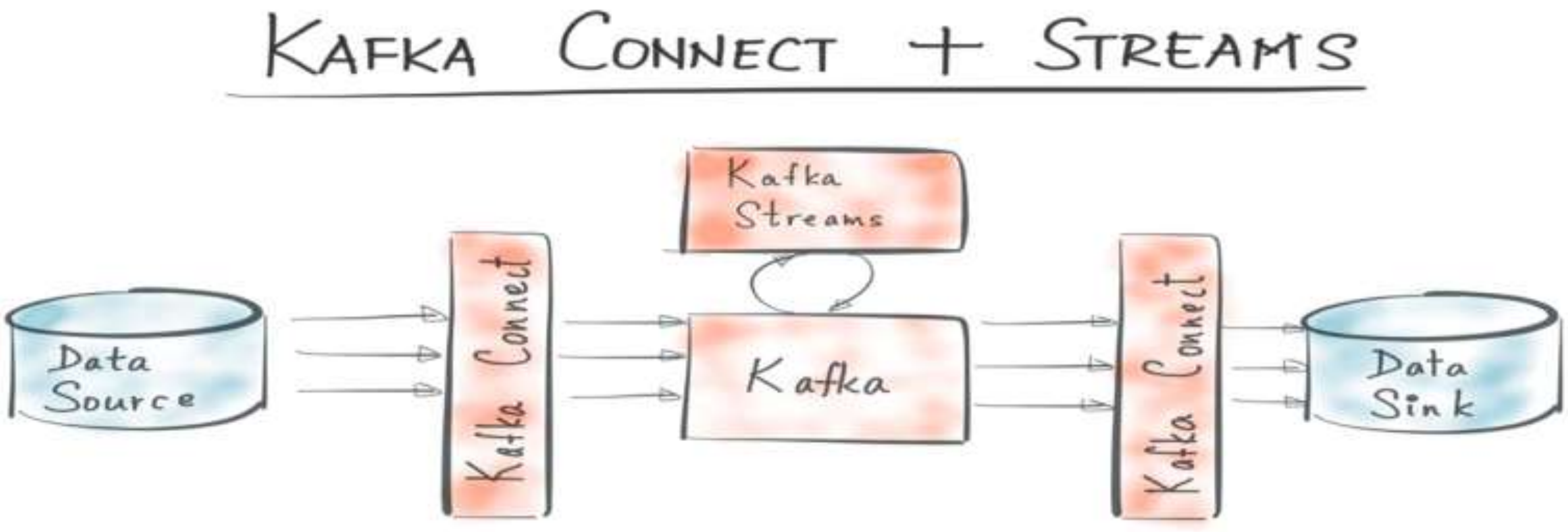
- **Retail system**: orders, shipments, returns, …
- **Financial system**: stock ticks, orders, …
- **Web site**: page views, clicks, searches, …
- **IoT**: sensor readings, …

and so on.

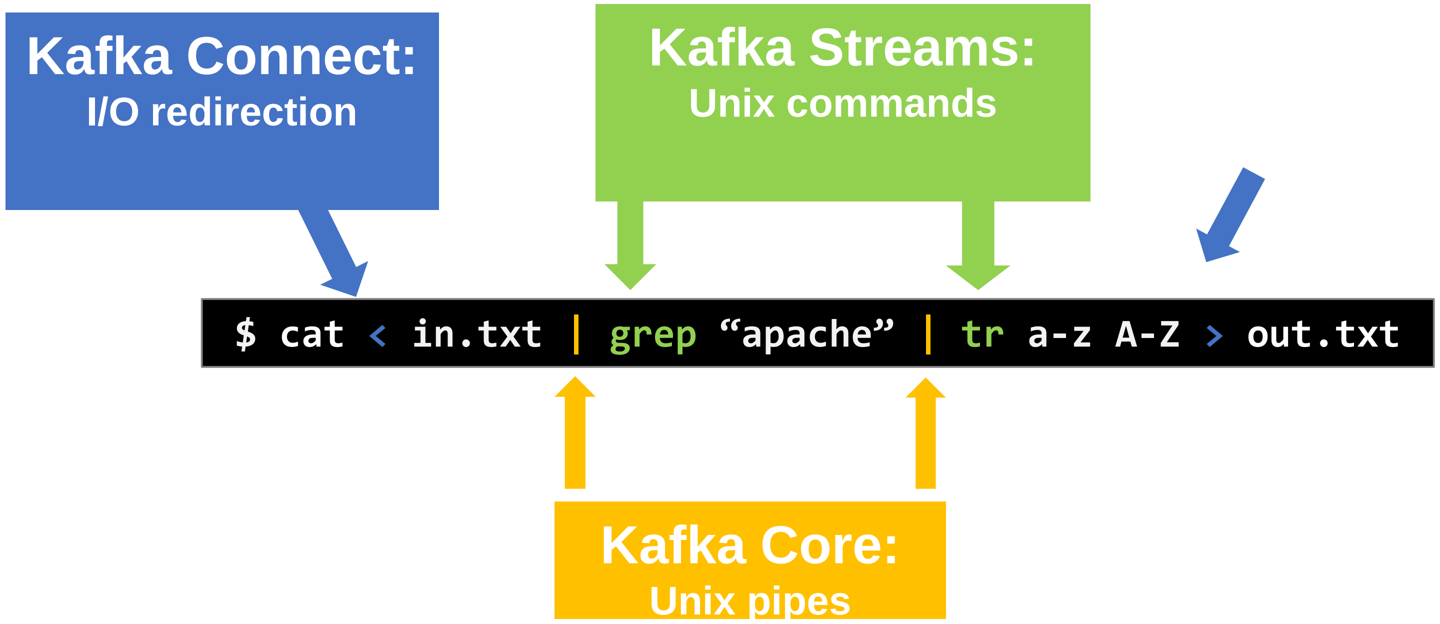
RETAIL :
SALES
SHIPMENTS
PRICE ADJUSTMENTS
INVENTORY ADJUSTMENTS
ANALYTICS
FRAUD
REORDERING

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



```
$ cat < in.txt | grep “apache” | tr a-z A-Z > out.txt
```

- 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?

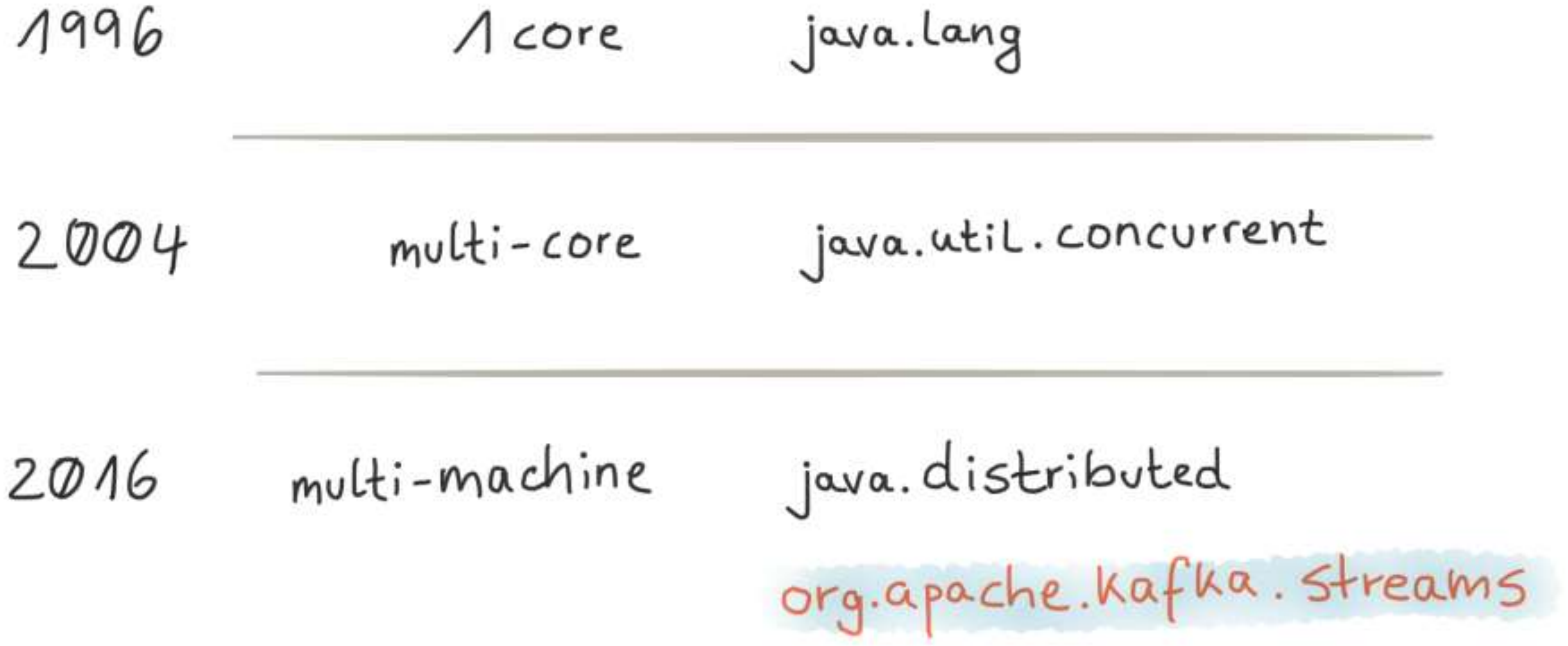
2.1 Before Kafka Streams?

- **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

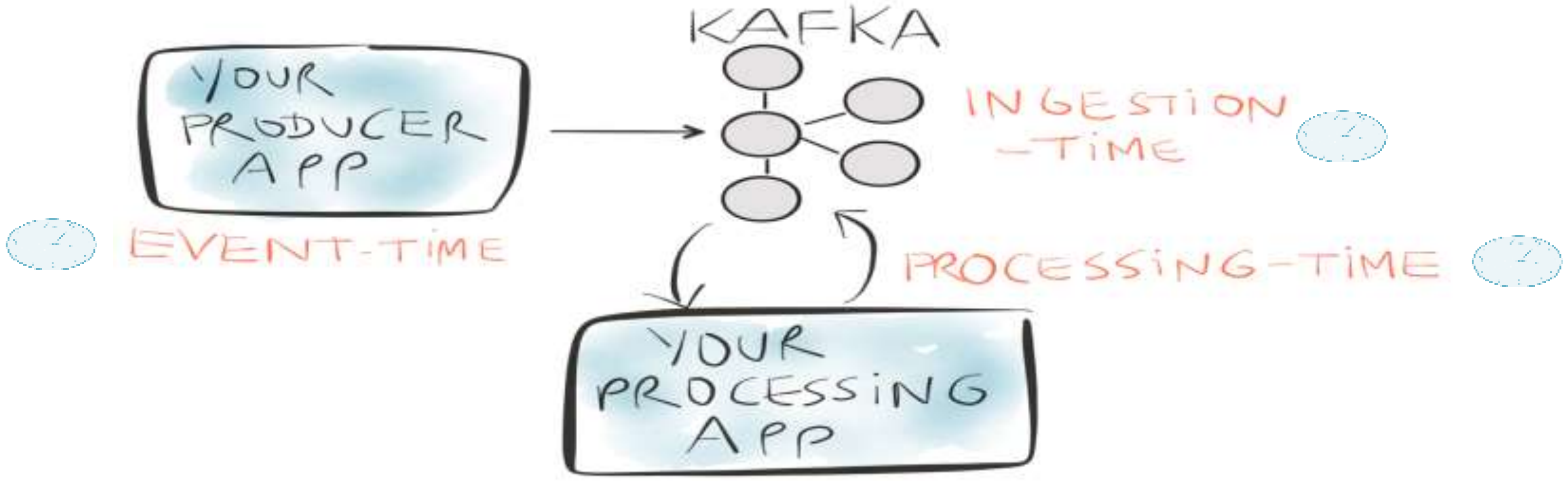
KStream vs KTable

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

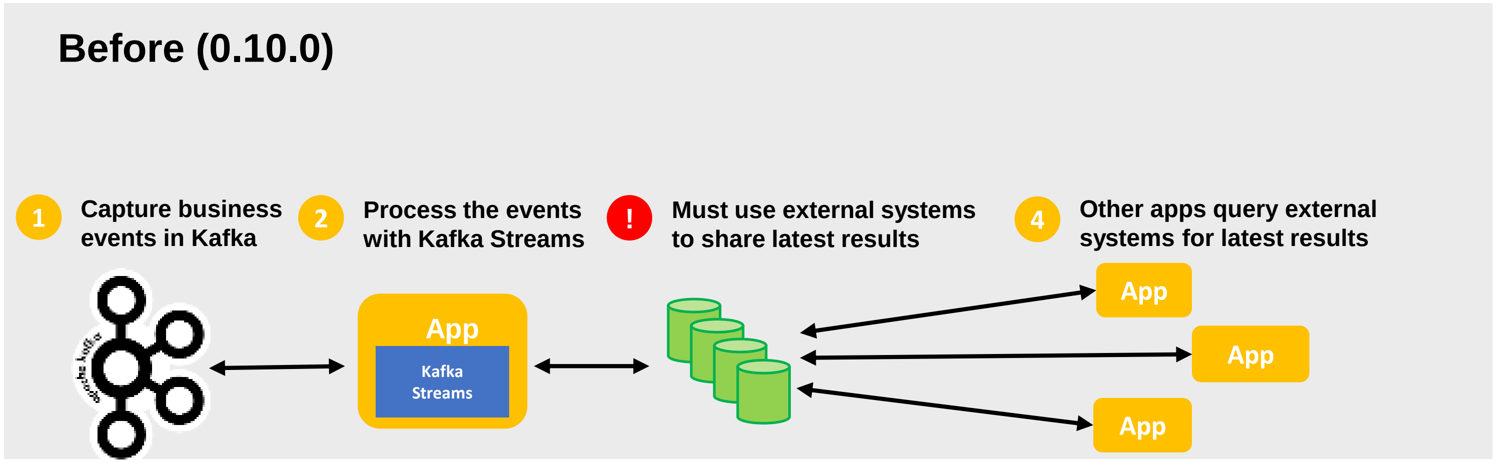
2.4 What are Kafka Streams key concepts?

- **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.

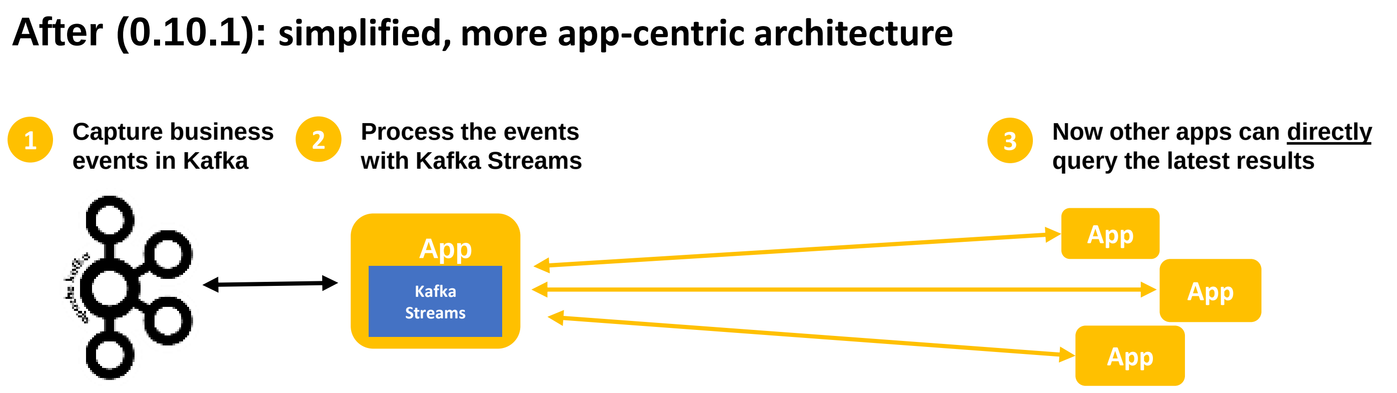


2.4 What are Kafka Streams key concepts?

➢ **Interactive Queries: Local queryable state**



Before (0.10.0)
1 Capture business events in Kafka
2 Process the events with Kafka Streams
! Must use external systems to share latest results
4 Other apps query external systems for latest results
App
Kafka Streams
App
App
App



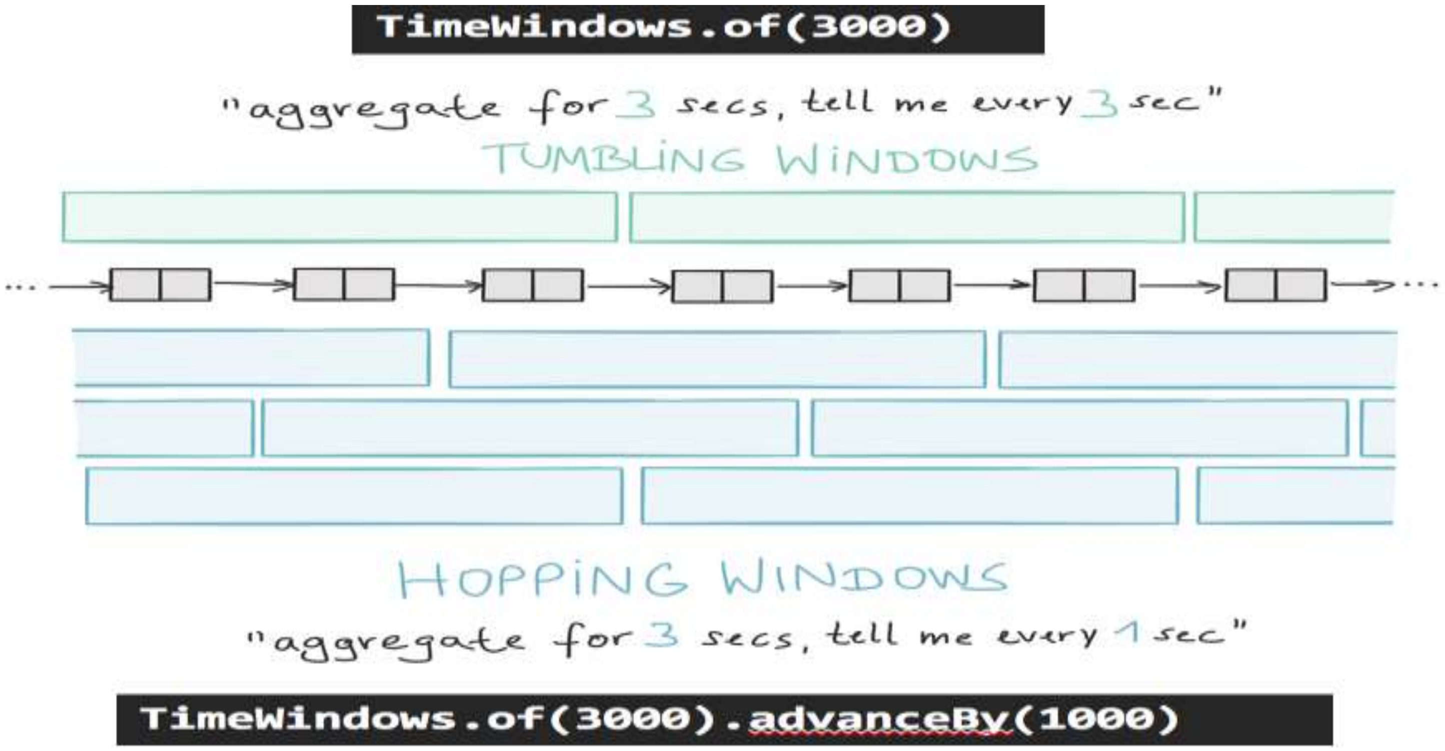
After (0.10.1): simplified, more app-centric architecture
1 Capture business events in Kafka
2 Process the events with Kafka Streams
3 Now other apps can directly query the latest results
App
Kafka Streams
App
App
App

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.


TimeWindows.of(3000)
"aggregate for 3 secs, tell me every 3 sec"
TUMBLING WINDOWS
HOPPING WINDOWS
"aggregate for 3 secs, tell me every 1 sec"
TimeWindows.of(3000).advanceBy(1000)

- **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");
```

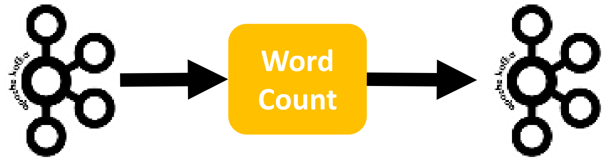
The **preferred API** for most use cases.

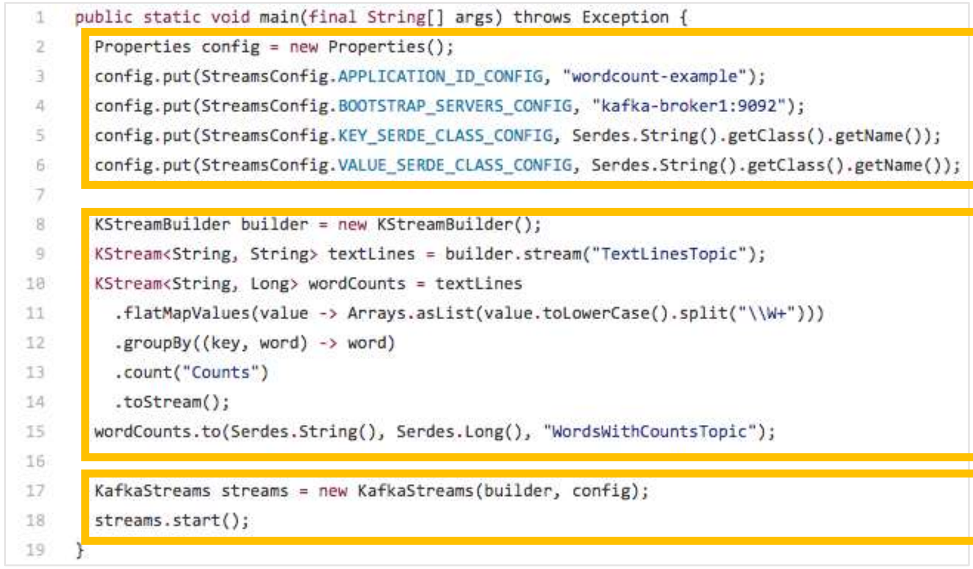
The DSL particularly appeals to users:

- 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





```java
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

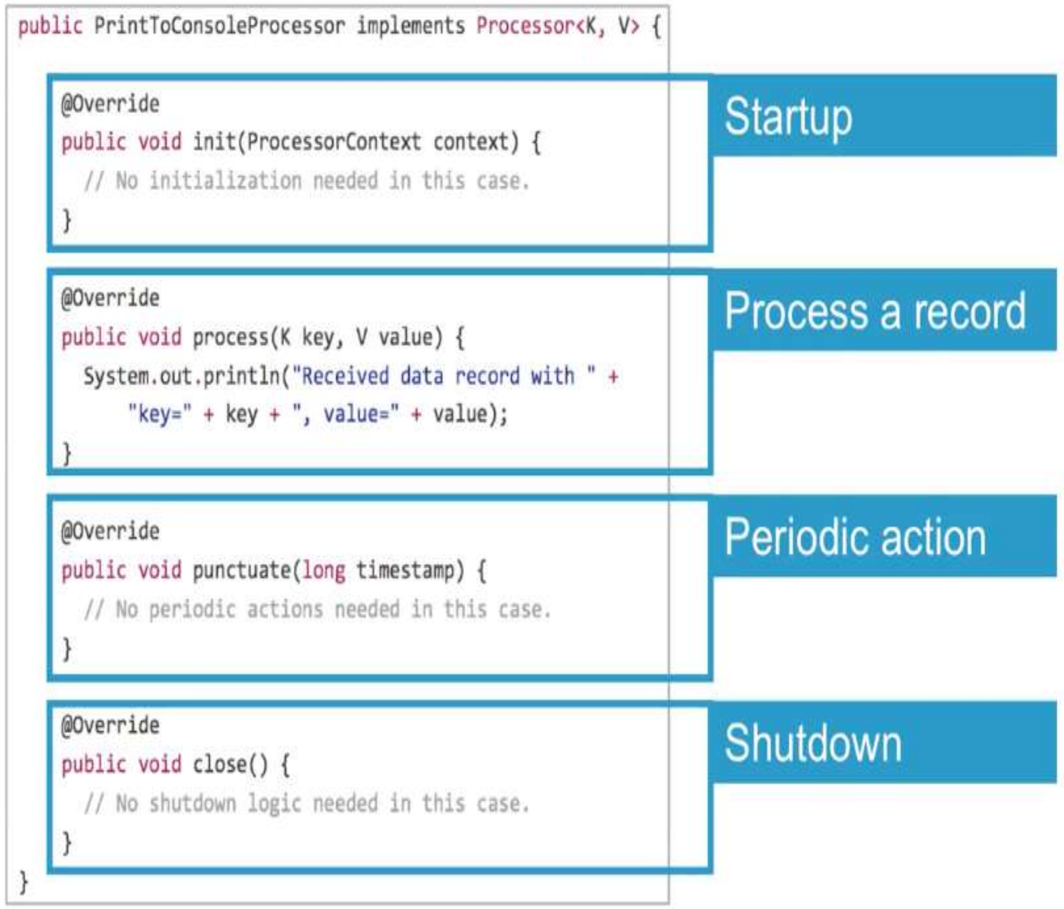
API option 2: Processor API (low level, imperative)

```
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



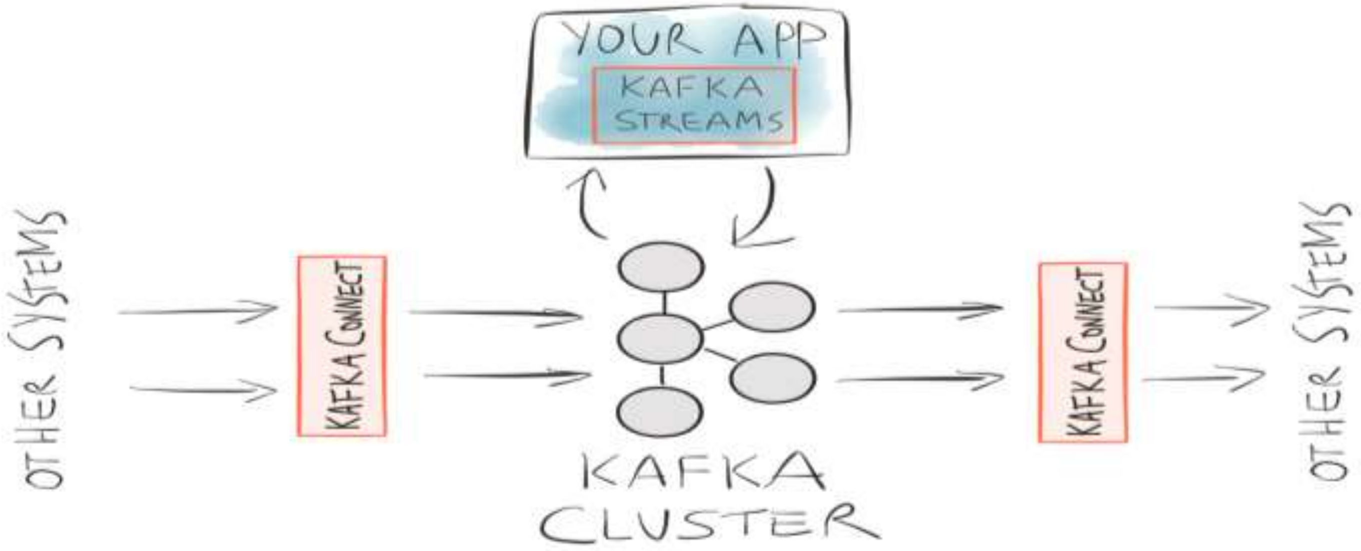
```
public PrintToConsoleProcessor implements Processor<K, V> {

  @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.
  }
}
```

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.



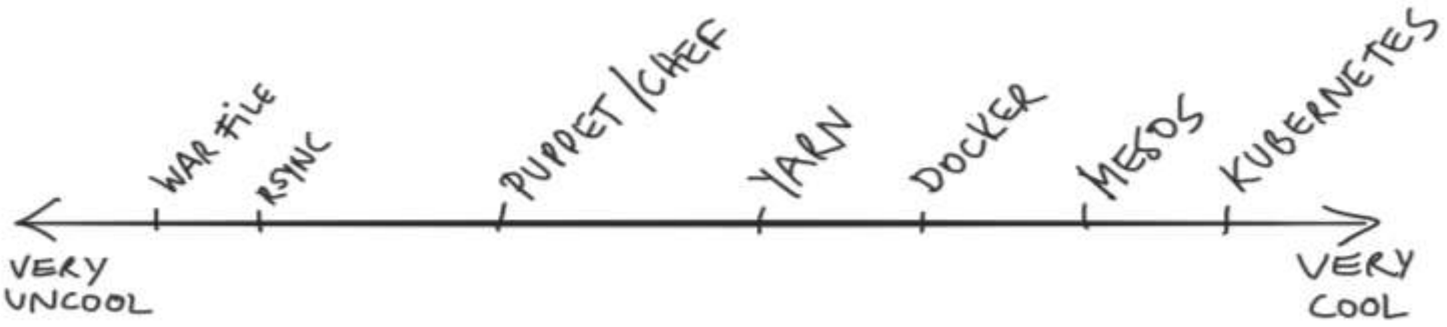
Step 2: Write the application code in Java or Scala

- 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

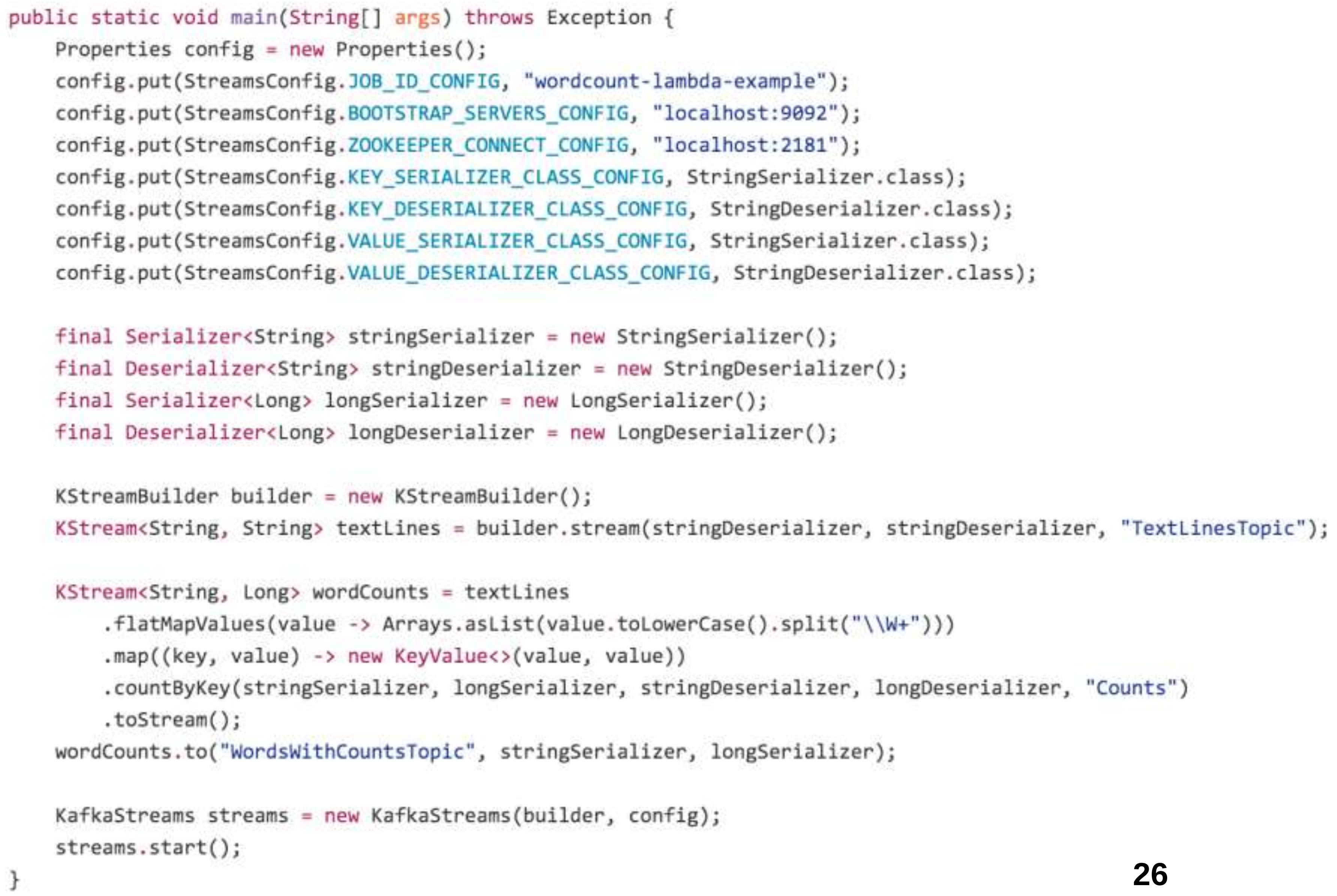
- ➢ How do you package and deploy your Kafka Streams apps?
  - **Whatever works for you!** Stick to what you/your company think is the best way for deploying and packaging a java application.
  - Kafka Streams integrates well with what you already use because an application that uses **Kafka Streams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

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