Writing a Streams Application

Any Java application that makes use of the Kafka Streams library is considered a Kafka Streams application. The computational logic of a Kafka Streams application is defined as a processor topology, which is a graph of stream processors (nodes) and streams (edges).

You can define the processor topology with the Kafka Streams APIs:

Kafka Streams DSL

A high-level API that provides the most common data transformation operations such as `map`, `filter`, `join`, and `aggregations` out of the box. The DSL is the recommended starting point for developers new to Kafka Streams, and should cover many use cases and stream processing needs.

Processor API

A low-level API that lets you add and connect processors as well as interact directly with state stores. The Processor API provides you with even more flexibility than the DSL but at the expense of requiring more manual work on the side of the application developer (e.g., more lines of code).

Libraries and Maven artifacts

This section lists the Kafka Streams related libraries that are available for writing your Kafka Streams applications.

The corresponding Maven artifacts of these libraries are available in Confluent's Maven repository:

```
<!-- Example pom.xml snippet when using Maven to build your Java applications. -->
<repositories>
    <repository>
        <id>confluent</id>
        <url>https://packages.confluent.io/maven/</url>
    </repository>
</repositories>
```

You can define dependencies on the following libraries for your Kafka Streams applications.

Group ID	Artifact ID	Version	Description
`org.apache.kafka`	`kafka-streams`	2.3.0-ccs	(Required) Base library for Kafka Streams.
`org.apache.kafka`	`kafka-streams-scala_2.11` or `kafka-streams-scala_2.12`	2.3.0-ccs	Scala API for Kafka Streams. Optional.
`org.apache.kafka`	`kafka-clients`	2.3.0-ccs	(Required) Apache Kafka® client library. Contains built-in serializers/deserializers.
`org.apache.avro`	`avro`	1.8.2	Apache Avro library. Optional (only needed when using Avro).
`io.confluent`	`kafka-streams-avro-serde`	5.3.0	Confluent's Avro Serializer/Deserializer. Optional (only needed when using Avro).

Tip

See the section Data Types and Serialization for more information about Serializers/Deserializers.

Example `pom.xml` snippet when using Maven:

```
<repositories>
    <repository>
        <id>confluent</id>
        <url>https://packages.confluent.io/maven/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>2.3.0-ccs</version>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>2.3.0-ccs</version>
    </dependency>

    <!-- For Scala developers -->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams-scala_2.11</artifactId>
        <!-- or
        <artifactId>kafka-streams-scala_2.12</artifactId>
        -->
        <version>2.3.0-ccs</version>
    </dependency>

    <!-- Dependencies below are required/recommended only when using Apache Avro. -->
    <dependency>
        <groupId>io.confluent</groupId>
        <artifactId>kafka-avro-serializer</artifactId>
        <version>5.3.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro</artifactId>
        <version>1.8.2</version>
    </dependency>
    <dependency>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro-maven-plugin</artifactId>
        <version>1.8.2</version>
    </dependency>
</dependencies>
```

See the Kafka Streams examples in the Confluent examples repository for a full Maven Project Object Model (POM) setup.

Using Kafka Streams within your application code

You can call Kafka Streams from anywhere in your application code, but usually these calls are made within the `main()` method of your application, or some variant thereof. The basic elements of defining a processing topology within your application are described below.

First, you must create an instance of `KafkaStreams`.

- The first argument of the `KafkaStreams` constructor takes a topology (either `StreamsBuilder#build()` for the DSL or `Topology` for the Processor API) that is used to define a topology.
- The second argument is an instance of `StreamsConfig`, which defines the configuration for this specific topology.

Code example:

```
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.kstream.StreamsBuilder;
import org.apache.kafka.streams.processor.Topology;

// Use the builders to define the actual processing topology, e.g. to specify
// from which input topics to read, which stream operations (filter, map, etc.)
// should be called, and so on.  We will cover this in detail in the subsequent
// sections of this Developer Guide.

StreamsBuilder builder = ...;  // when using the DSL
Topology topology = builder.build();
//
// OR
//
Topology topology = ...; // when using the Processor API

// Use the configuration properties to tell your application where the Kafka cluster is,
// which Serializers/Deserializers to use by default, to specify security settings,
// and so on.
Properties props = ...;

KafkaStreams streams = new KafkaStreams(topology, props);
```

At this point, internal structures are initialized, but the processing is not started yet. You have to explicitly start the Kafka Streams thread by calling the `KafkaStreams#start()` method:

```
// Start the Kafka Streams threads
streams.start();
```

If there are other instances of this stream processing application running elsewhere (e.g., on another machine), Kafka Streams transparently re-assigns tasks from the existing instances to the new instance that you just started. For more information, see Stream Partitions and Tasks and Threading Model.

To catch any unexpected exceptions, you can set an `java.lang.Thread.UncaughtExceptionHandler` before you start the application. This handler is called whenever a stream thread is terminated by an unexpected exception:

```
// Java 8+, using lambda expressions
streams.setUncaughtExceptionHandler((Thread thread, Throwable throwable) -> {
  // here you should examine the throwable/exception and perform an appropriate action!
});


// Java 7
streams.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
  public void uncaughtException(Thread thread, Throwable throwable) {
    // here you should examine the throwable/exception and perform an appropriate action!
  }
});
```

To stop the application instance, call the `KafkaStreams#close()` method:

```
// Stop the Kafka Streams threads
streams.close();
```

To allow your application to gracefully shutdown in response to SIGTERM, it is recommended that you add a shutdown hook and call `KafkaStreams#close`.

- Here is a shutdown hook example in Java 8+:

  ```
  // Add shutdown hook to stop the Kafka Streams threads.
  // You can optionally provide a timeout to `close`.
  Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
  ```

- Here is a shutdown hook example in Java 7:

```
// Add shutdown hook to stop the Kafka Streams threads.
// You can optionally provide a timeout to `close`.
Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
  @Override
  public void run() {
      streams.close();
  }
}));
```

After an application is stopped, Kafka Streams will migrate any tasks that had been running in this instance to available remaining instances.

Please report any inaccuracies on this page or suggest an edit.

5 Votes

Last updated on Sep 10, 2019.