Confluence KAFKA Administration

Rajesh Pasham

Kafka Connectors

Overview

- Apache Kafka is a distributed streaming platform.
- In this topic, we'll learn how to use Kafka Connectors.
- We'll have a look at:
  - Different types of Kafka Connectors
  - Features and modes of Kafka Connect
  - Connectors configuration using property files as well as the REST API

Basics of Kafka Connect and Kafka Connectors

- Kafka Connect is a framework for connecting Kafka with external systems such as databases, key-value stores, search indexes, and file systems, using so-called *Connectors*.
- Kafka Connectors are ready-to-use components, which can help us to import data from external systems into Kafka topics and export data from Kafka topics into external systems.
- We can use existing connector implementations for common data sources and sinks or implement our own connectors.

Basics of Kafka Connect and Kafka Connectors

- A *source connector* collects data from a system.
- Source systems can be entire databases, streams tables, or message brokers.
- A source connector could also collect metrics from application servers into Kafka topics, making the data available for stream processing with low latency.
- A *sink connector* delivers data from Kafka topics into other systems, which might be indexes such as Elasticsearch, batch systems such as Hadoop, or any kind of database.

Basics of Kafka Connect and Kafka Connectors

- Some connectors are maintained by the community, while others are supported by Confluent or its partners.
- Really, we can find connectors for most popular systems, like S3, JDBC, and Cassandra, just to name a few.

Features

- Kafka Connect features include:
  - A framework for connecting external systems with Kafka – it simplifies the development, deployment, and management of connectors
  - Distributed and standalone modes – it helps us to deploy large clusters by leveraging the distributed nature of Kafka, as well as setups for development, testing, and small production deployments
  - REST interface – we can manage connectors using a REST API
  - Automatic offset management – Kafka Connect helps us to handle the offset commit process, which saves us the trouble of implementing this error-prone part of connector development manually

Features

- Kafka Connect features include:
  - Distributed and scalable by default – Kafka Connect uses the existing group management protocol; we can add more workers to scale up a Kafka Connect cluster
  - Streaming and batch integration – Kafka Connect is an ideal solution for bridging streaming and batch data systems in connection with Kafka's existing capabilities
  - Transformations – these enable us to make simple and lightweight modifications to individual messages

Quick Start Kafka Connect

- For starters, we'll discuss the principle of Kafka Connect, using its most basic Connectors, which are the file *source* connector and the file *sink* connector.
- Conveniently, kafka comes with both of these connectors, as well as reference configurations.

Quick Start Kafka Connect

Source Connector Configuration

- For the source connector, the reference configuration is available at *$KAFKA_HOME/config/connect-file-source.properties*:
  - name=local-file-source
  - connector.class=FileStreamSource
  - tasks.max=1
  - topic=connect-test
  - file=test.txt

Quick Start Kafka Connect

Source Connector Configuration

- This configuration has some properties that are common for all source connectors:
  - *name* is a user-specified name for the connector instance
  - *connector.class* specifies the implementing class, basically the kind of connector
  - *tasks.max* specifies how many instances of our source connector should run in parallel, and
  - *topic* defines the topic to which the connector should send the output
- In this case, we also have a connector-specific attribute:
  - *File* defines the file from which the connector should read the input

Quick Start Kafka Connect

Source Connector Configuration

- For this to work then, let's create a basic file with some content:
  - echo -e "foo\nbar\n" > $KAFKA_HOME/test.txt

- Note that the working directory is $KAFKA_HOME.

Quick Start Kafka Connect

Sink Connector Configuration

- For our sink connector, we'll use the reference configuration at *$KAFKA_HOME/config/connect-file-sink.properties*:
  - name=local-file-sink
  - connector.class=FileStreamSink
  - tasks.max=1
  - file=test.sink.txt
  - topics=connect-test
- Logically, it contains exactly the same parameters, though this time *connector.class* specifies the sink connector implementation, and *file* is the location where the connector should write the content.

Quick Start Kafka Connect

Worker Config

- Finally, we have to configure the Connect worker, which will integrate our two connectors and do the work of reading from the source connector and writing to the sink connector.
- For that, we can use *$KAFKA_HOME/config/connect-standalone.properties*:
  - bootstrap.servers=localhost:9092
  - key.converter=org.apache.kafka.connect.json.JsonConverter
  - value.converter=org.apache.kafka.connect.json.JsonConverter
  - key.converter.schemas.enable=false
  - value.converter.schemas.enable=false
  - offset.storage.file.filename=/tmp/connect.offsets
  - offset.flush.interval.ms=10000
  - plugin.path=/share/java

Quick Start Kafka Connect

Worker Config

- Note that *plugin.path* can hold a list of paths, where connector implementations are available
- As we'll use connectors bundled with Kafka, we can set *plugin.path* to *$KAFKA_HOME/share/java*. Working with Windows, it might be necessary to provide an absolute path here.
- For the other parameters, we can leave the default values:
  - *bootstrap.servers* contains the addresses of the Kafka brokers
  - *key.converter* and *value.converter* define converter classes, which serialize and deserialize the data as it flows from the source into Kafka and then from Kafka to the sink
  - *key.converter.schemas.enable* and *value.converter.schemas.enable* are converter-specific settings
  - ***offset.storage.file.filename* is the most important setting when running Connect in standalone mode: it defines where Connect should store its offset data**
  - *offset.flush.interval.ms* defines the interval at which the worker tries to commit offsets for tasks

Quick Start Kafka Connect

Kafka Connect in Standalone Mode

- And with that, we can start our first connector setup:
  - $KAFKA_HOME/bin/connect-standalone.sh \
    $KAFKA_HOME/config/connect-standalone.properties \
    $KAFKA_HOME/config/connect-file-source.properties \
    $KAFKA_HOME/config/connect-file-sink.properties
- First off, we can inspect the content of the topic using the command line:
  - $KAFKA_HOME/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning

Quick Start Kafka Connect

Kafka Connect in Standalone Mode

- As we can see, the source connector took the data from the *test.txt* file, transformed it into JSON, and sent it to Kafka:
  - {"schema":{"type":"string","optional":false},"payload":"foo"}
  - {"schema":{"type":"string","optional":false},"payload":"bar"}
- And, if we have a look at the folder *$KAFKA_HOME*, we can see that a file *test.sink.txt* was created here:
  - cat $KAFKA_HOME/test.sink.txt
    - foo
    - Bar
- As the sink connector extracts the value from the *payload* attribute and writes it to the destination file, the data in *test.sink.txt* has the content of the original *test.txt* file.

Quick Start Kafka Connect

Kafka Connect in Standalone Mode

- Now let's add more lines to *test.txt*.
- When we do, we see that the source connector detects these changes automatically.
- We only have to make sure to insert a newline at the end, otherwise, the source connector won't consider the last line.
- At this point, let's stop the Connect process, as we'll start Connect in *distributed mode* in a few lines.

Connect's REST API

- Until now, we made all configurations by passing property files via the command line.
- However, as Connect is designed to run as a service, there is also a REST API available.
- By default, it is available at *http://localhost:8083*. A few endpoints are:
  - *GET /connectors* – returns a list with all connectors in use
  - *GET /connectors/{name}* – returns details about a specific connector
  - *POST /connectors* – creates a new connector; the request body should be a JSON object containing a string name field and an object config field with the connector configuration parameters

Connect's REST API

- *GET /connectors/{name}/status* – returns the current status of the connector – including if it is running, failed or paused – which worker it is assigned to, error information if it has failed, and the state of all its tasks
- *DELETE /connectors/{name}* – deletes a connector, gracefully stopping all tasks and deleting its configuration
- *GET /connector-plugins* – returns a list of connector plugins installed in the Kafka Connect cluster

Kafka Connect in Distributed Mode

- The standalone mode works perfectly for development and testing, as well as smaller setups.
- However, if we want to make full use of the distributed nature of Kafka, we have to launch Connect in distributed mode.
- By doing so, connector settings and metadata are stored in Kafka topics instead of the file system.
- As a result, the worker nodes are really stateless.

Kafka Connect in Distributed Mode

Starting Connect

- A reference configuration for distributed mode can be found at $KAFKA_HOME/*config/connect-distributed.properties*.
- Parameters are mostly the same as for standalone mode. There are only a few differences:
  - ***group.id*** defines the name of the Connect cluster group. The value must be different from any consumer group ID
  - ***offset.storage.topic***, ***config.storage.topic*** and ***status.storage.topic*** define topics for these settings. For each topic, we can also define a replication factor
- We can start Connect in distributed mode as follows:
  - $KAFKA_HOME/bin/connect-distributed.sh \
    $KAFKA_HOME/config/connect-distributed.properties

Kafka Connect in Distributed Mode

Adding Connectors Using the REST API

- Now, compared to the standalone startup command, we didn't pass any connector configurations as arguments.
- Instead, we have to create the connectors using the REST API.
- To set up our example from before, we have to send two POST requests to *http://localhost:8083/connectors* containing the following JSON structs.

Kafka Connect in Distributed Mode

Adding Connectors Using the REST API

- First, we need to create the body for the source connector POST as a JSON file. Here, we'll call it *connect-file-source.json*:

```
{
  "name": "local-file-source",
  "config": {
    "connector.class": "FileStreamSource",
    "tasks.max": 1,
    "file": "test-distributed.txt",
    "topic": "connect-distributed"
  }
}
```

- Note how this looks pretty similar to the reference configuration file we used the first time.
- And then we POST it:
  - curl -d @"$KAFKA_HOME/connect-file-source.json" \
    -H "Content-Type: application/json" \
    -X POST http://localhost:8083/connectors

Kafka Connect in Distributed Mode

Adding Connectors Using the REST API

- Then, we'll do the same for the sink connector, calling the file *connect-file-sink.json*:

```
{
  "name": "local-file-sink",
  "config": {
    "connector.class": "FileStreamSink",
    "tasks.max": 1,
    "file": "test-distributed.sink.txt",
    "topics": "connect-distributed"
  }
}
```

- And perform the POST like before:
  - curl -d @$KAFKA_HOME/connect-file-sink.json \
    -H "Content-Type: application/json" \
    -X POST http://localhost:8083/connectors

Kafka Connect in Distributed Mode

Adding Connectors Using the REST API

- If needed, we can verify, that this setup is working correctly:
- $KAFKA_HOME/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-distributed --from-beginning
  - {"schema":{"type":"string","optional":false},"payload":"foo"}
  - {"schema":{"type":"string","optional":false},"payload":"bar"}
- And, if we have a look at the folder *$KAFKA_HOME*, we can see that a file *test-distributed.sink.txt* was created here:
  - cat $KAFKA_HOME/test-distributed.sink.txt
    - foo
    - Bar
- After we tested the distributed setup, let's clean up, by removing the two connectors:
  - curl -X DELETE http://localhost:8083/connectors/local-file-source
  - curl -X DELETE http://localhost:8083/connectors/local-file-sink

Transforming Data

Supported Transformations

- Transformations enable us to make simple and lightweight modifications to individual messages.
- Kafka Connect supports the following built-in transformations:
  - *InsertField* – Add a field using either static data or record metadata
  - *ReplaceField* – Filter or rename fields
  - *MaskField* – Replace a field with the valid null value for the type (zero or an empty string, for example)
  - *HoistField* – Wrap the entire event as a single field inside a struct or a map

Transforming Data

Supported Transformations

- Kafka Connect supports the following built-in transformations:
  - *ExtractField* – Extract a specific field from struct and map and include only this field in the results
  - *SetSchemaMetadata* – Modify the schema name or version
  - *TimestampRouter* – Modify the topic of a record based on original topic and timestamp
  - *RegexRouter* – Modify the topic of a record based on original topic, a replacement string, and a regular expression

Transforming Data

Supported Transformations

- A transformation is configured using the following parameters:
  - *transforms* – A comma-separated list of aliases for the transformations
  - *transforms.$alias.type* – Class name for the transformation
  - *transforms.$alias.$transformationSpecificConfig* – Configuration for the respective transformation

Transforming Data

Applying a Transformer

- To test some transformation features, let's set up the following two transformations:
  - First, let's wrap the entire message as a JSON struct
  - After that, let's add a field to that struct
- Before applying our transformations, we have to configure Connect to use schemaless JSON, by modifying the *connect-distributed.properties*:
  - key.converter.schemas.enable=false
  - value.converter.schemas.enable=false
- After that, we have to restart Connect, again in distributed mode:
  - $KAFKA_HOME/bin/connect-distributed.sh \
    $KAFKA_HOME/etc/kafka/connect-distributed.properties

Transforming Data

Applying a Transformer

- Again, we need to create the body for the source connector POST as a JSON file. Here, we'll call it *connect-file-source-transform.json.*
- Besides the already known parameters, we add a few lines for the two required transformations:

```
{
  "name": "local-file-source",
  "config": {
    "connector.class": "FileStreamSource",
    "tasks.max": 1,
    "file": "test-transformation.txt",
    "topic": "connect-transformation",
    "transforms": "MakeMap,InsertSource",
    "transforms.MakeMap.type": "org.apache.kafka.connect.transforms.HoistField$Value",
    "transforms.MakeMap.field": "line",
    "transforms.InsertSource.type": "org.apache.kafka.connect.transforms.InsertField$Value",
    "transforms.InsertSource.static.field": "data_source",
    "transforms.InsertSource.static.value": "test-file-source"
  }
}
```

Transforming Data

Applying a Transformer

- After that, let's perform the POST:
  - curl -d @$KAFKA_HOME/connect-file-source-transform.json \
  - -H "Content-Type: application/json" \
  - -X POST http://localhost:8083/connectors
- Let's write some lines to our *test-transformation.txt*:
  - Foo
  - Bar
- If we now inspect the *connect-transformation* topic, we should get the following lines:
  - {"line":"Foo","data_source":"test-file-source"}
  - {"line":"Bar","data_source":"test-file-source"}

Using Ready Connectors

- After using these simple connectors, let's have a look at more advanced ready-to-use connectors, and how to install them.

Where to Find Connectors

- Pre-built connectors are available from different sources:
  - A few connectors are bundled with plain Apache Kafka (source and sink for files and console)
  - Some more connectors are bundled with Confluent Platform (ElasticSearch, HDFS, JDBC, and AWS S3)

Using Ready Connectors

- Also check out **Confluent Hub**, which is kind of an app store for Kafka connectors. The number of offered connectors is growing continuously:
  - Confluent connectors (developed, tested, documented and are fully supported by Confluent)
  - Certified connectors (implemented by a 3rd party and certified by Confluent)
  - Community-developed and -supported connectors
- Beyond that, Confluent also provides a **Connectors Page**, with some connectors which are also available at the Confluent Hub, but also with some more community connectors
- And finally, there are also vendors, who provide connectors as part of their product. For example, Landoop provides a streaming library called **Lenses**, which also contains a set of ~25 open source connectors (many of them also cross-listed in other places)

Using Ready Connectors

Installing Connectors from Confluent Hub

- The enterprise version of Confluent provides a script for installing Connectors and other components from Confluent Hub (the script is not included in the Open Source version). If we're using the enterprise version, we can install a connector using the following command:
  - $CONFLUENT_HOME/bin/confluent-hub install confluentinc/kafka-connect-mqtt:1.0.0-preview

Using Ready Connectors

Installing Connectors Manually

- If we need a connector, which is not available on Confluent Hub or if we have the Open Source version of Confluent, we can install the required connectors manually. For that, we have to download and unzip the connector, as well as move the included libs to the folder specified as *plugin.path*.
- For each connector, the archive should contain two folders that are interesting for us:
- The *lib* folder contains the connector jar, for example, *kafka-connect-mqtt-1.0.0-preview.jar*, as well as some more jars required by the connector
- The *etc* folder holds one or more reference config files

Using Ready Connectors

Installing Connectors Manually

- We have to move the *lib* folder to *$CONFLUENT_HOME/share/java*, or whichever path we specified as *plugin.path* in *connect-standalone.properties* and *connect-distributed.properties*. In doing so, it might also make sense to rename the folder to something meaningful.
- We can use the config files from *etc* either by referencing them while starting in standalone mode, or we can just grab the properties and create a JSON file from them.