Connector Developer Guide

This guide describes how developers can write new connectors for Kafka Connect to move data between Apache Kafka® and other systems. It briefly reviews a few key Kafka Connect concepts and then describes how to create a simple connector.

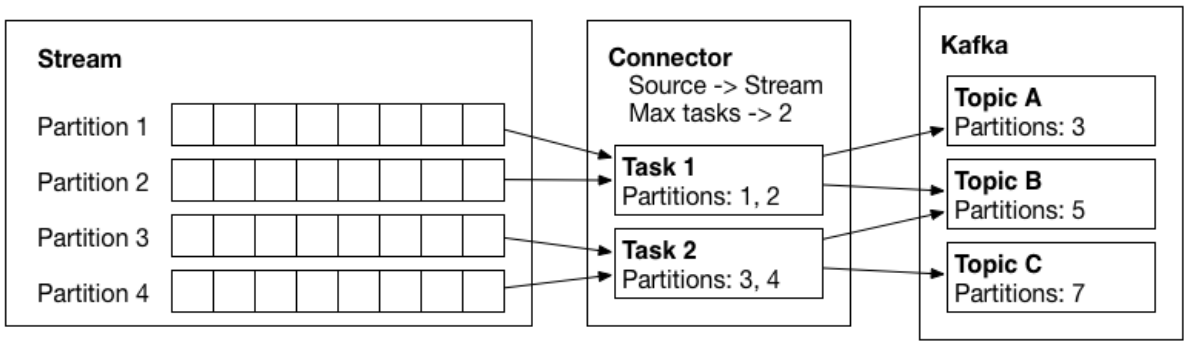
Core Concepts and APIs

Connectors and Tasks

To copy data between Kafka and another system, users instantiate KafkaConnectors for the systems they want to pull data from or push data to. Connectors come in two flavors: SourceConnectors, which import data from another system, andSinkConnectors, which export data to another system. For example, `JDBCSourceConnector` would import a relational database into Kafka, and `HDFSSinkConnector` would export the contents of a Kafka topic to HDFS files.

Implementations of the `Connector` class do not perform data copying themselves: their configuration describes the set of data to be copied, and the `Connector` is responsible for breaking that job into a set ofTasks that can be distributed to Kafka Connect workers. Tasks also come in two corresponding flavors: SourceTask and SinkTask. Optionally, the implementation of the `Connector` class can monitor the data changes of external systems and request task reconfiguration.

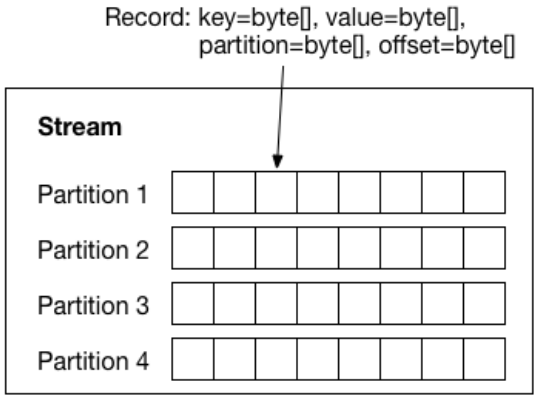
With an assignment of data to be copied in hand, each `Task` must copy its subset of the data to or from Kafka. The data that a connector copies must be represented as a **partitioned stream**, similar to the model of a Kafka topic, where each partition is an ordered sequence of records with offsets. Each task is assigned a subset of the partitions to process. Sometimes this mapping is clear: each file in a set of log files can be considered a partition, each line within a file is a record, and offsets are simply the position in the file. In other cases it may require a bit more effort to map to this model: a JDBC connector can map each table to a partition, but the offset is less clear. One possible mapping uses a timestamp column to generate queries to incrementally return new data, and the last queried timestamp can be used as the offset.



Example of a source connector which has created two tasks, which copy data from input partitions and write records to Kafka.

Partitions and Records

Each partition is an ordered sequence of key-value records. Both the keys and values can have complex structures, represented by the data structures in the org.apache.kafka.connect.data package. Many primitive types as well as arrays, structs, and nested data structures are supported. For most types, standard Java types like `java.lang.Integer`, `java.lang.Map`, and `java.lang.Collection` can be used directly. For structured records, the Struct class should be used.



A partitioned stream: the data model that connectors must map all source and sink systems to. Each record contains keys and values (with schemas), a partition ID, and offsets within that partition.

In order to track the structure and compatibility of records in partitions, Schemas may be included with each record. Because schemas are commonly generated on the fly, based on the data source, a SchemaBuilder class is included which makes constructing schemas very easy.

This runtime data format does not assume any particular serialization format; this conversion is handled by Converter implementations, which convert between `org.apache.kafka.connect.data` runtime format and serialized data represented as `byte[]`. Connector developers should not have to worry about the details of this conversion.

In addition to the key and value, records have partition IDs and offsets. These are used by the framework to periodically commit the offsets of data that have been processed. In the event of a failure, processing can resume from the last committed offsets, avoiding unnecessary reprocessing and duplication of events.

Dynamic Connectors

Not all connectors have a static set of partitions, so `Connector` implementations are also responsible for monitoring the external system for any changes that might require reconfiguration. For example, in the `JDBCSourceConnector` example, the `Connector` might assign a set of tables to each `Task`. When a new table is created, it must discover this so it can assign the new table to one of th `Tasks` by updating its configuration. When it notices a change that requires reconfiguration (or a change in the number of `Tasks`), it notifies the framework and the framework updates any corresponding `Tasks`.

Developing a Simple Connector

Developing a connector only requires implementing two interfaces, the `Connector` and `Task`. A simple example of connectors that read and write lines from and to files is included in the source code for Kafka Connect in the `org.apache.kafka.connect.file` package. The classes `SourceConnector`/`SourceTask` implement a source connector that reads lines from files and `SinkConnector`/`SinkTask` implement a sink connector that writes each record to a file.

Tip

Refer to the example source code for full examples. The following sections provide key steps and code snippets only.

Connector Example

We'll cover the `SourceConnector` as a simple example. `SinkConnector` implementations are very similar. Start by creating the class that inherits from `SourceConnector` and add a couple of fields that will store parsed configuration information (the filename to read from and the topic to send data to):

```
public class FileStreamSourceConnector extends SourceConnector {
  private String filename;
  private String topic;
```

The easiest method to fill in is `getTaskClass()`, which defines the class that should be instantiated in worker processes to actually read the data:

```
@Override
public Class<? extends Task> getTaskClass() {
  return FileStreamSourceTask.class;
}
```

We will define the `FileStreamSourceTask` class below. Next, we add some standard lifecycle methods, `start()` and `stop()`:

```
@Override
public void start(Map<String, String> props) {
  // The complete version includes error handling as well.
  filename = props.get(FILE_CONFIG);
  topic = props.get(TOPIC_CONFIG);
}

@Override
public void stop() {
  // Nothing to do since no background monitoring is required
}
```

Finally, the real core of the implementation is in `taskConfigs()`. In this case we're only handling a single file, so even though we may be permitted to generate more tasks as per the `maxTasks` argument, we return a list with only one entry:

```
@Override
public List<Map<String, String>> taskConfigs(int maxTasks) {
  ArrayList<Map<String, String>> configs = new ArrayList<>();
  // Only one input partition makes sense.
  Map<String, String> config = new HashMap<>();
  if (filename != null)
    config.put(FILE_CONFIG, filename);
  config.put(TOPIC_CONFIG, topic);
  configs.add(config);
  return configs;
}
```

Even with multiple tasks, this method implementation is usually pretty simple. It just has to determine the number of input tasks, which may require contacting the remote service it is pulling data from, and then divvy them up. Because some patterns for splitting work among tasks are so common, some utilities are provided in ConnectorUtils to simplify these cases.

Note that this simple example does not include dynamic input. See the discussion in the next section for how to trigger updates to task configs.

Task Example - Source Task

Next we'll describe the implementation of the corresponding `SourceTask`. The class is small, but too long to cover completely in this guide. We'll use helper methods which we won't provide the details of to describe most of the implementation, but you can refer to the source code for the full example.

Just as with the connector, we need to create a class inheriting from the appropriate base `Task` class. It also has some standard lifecycle methods:

```
public class FileStreamSourceTask extends SourceTask {
  private String filename;
  private InputStream stream;
  private String topic;
  private Long streamOffset;

  public void start(Map<String, String> props) {
    filename = props.get(FileStreamSourceConnector.FILE_CONFIG);
    stream = openOrThrowError(filename);
    topic = props.get(FileStreamSourceConnector.TOPIC_CONFIG);
  }

  @Override
  public synchronized void stop() {
    stream.close()
  }
```

These are slightly simplified versions, but show that that these methods should be relatively simple and the only work they perform is allocating or freeing resources. There are two points to note about this implementation. First, the `start()` method does not yet handle resuming from a previous offset, which will be addressed in a later section. Second, the `stop()` method is synchronized. This will be necessary because `SourceTasks` are given a dedicated thread which they can block indefinitely, so they need to be stopped with a call from a different thread in the Worker.

Next, we implement the main functionality of the task: the `poll()` method that gets records from the input system and returns a `List<SourceRecord>`:

```
@Override
public List<SourceRecord> poll() throws InterruptedException {
  try {
    ArrayList<SourceRecord> records = new ArrayList<>();
    while (streamValid(stream) && records.isEmpty()) {
      LineAndOffset line = readToNextLine(stream);
      if (line != null) {
        Map sourcePartition = Collections.singletonMap("filename", filename);
        Map sourceOffset = Collections.singletonMap("position", streamOffset);
        records.add(new SourceRecord(sourcePartition, sourceOffset, topic, Schema.STRING_SCHEMA, line));
      } else {
        Thread.sleep(1);
      }
    }
    return records;
  } catch (IOException e) {
    // Underlying stream was killed, probably as a result of calling stop. Allow to return
    // null, and driving thread will handle any shutdown if necessary.
  }
  return null;
}
```

Again, we've omitted some details, but we can see the important steps: the `poll()` method is going to be called repeatedly, and for each call it will loop trying to read records from the file. For each line it reads, it also tracks the file offset. It uses this information to create an output SourceRecord with four pieces of information: the source partition (there is only one, the single file being read), source offset (position in the file), output topic name, and output value (the line, including a schema indicating this value will always be a string). Other variants of the `SourceRecord` constructor can also include a specific output partition and a key.

Note that this implementation uses the normal Java `InputStream` interface and may sleep if data is not available. This is acceptable because Kafka Connect provides each task with a dedicated thread. While task implementations have to conform to the basic `poll()` interface, they have a lot of flexibility in how they are implemented. In this case, an NIO-based implementation would be more efficient, but this simple approach works, is quick to implement, and is compatible with older versions of Java.

Although not used in the example, `SourceTask` also provides two APIs to commit offsets in the source system: `commit()` and `commitRecord()`. These APIs are provided for source systems which have an acknowledgement mechanism for messages. Overriding these methods allows the source connector to acknowledge messages in the source system, either in bulk or individually, once they have been written to Kafka.

The `commit()` API stores the offsets in the source system, up to the offsets that have been returned by `poll()`. The implementation of this API should block until the commit is complete. The `commitRecord()` API saves the offset in the source system for each `SourceRecord` after it is written to Kafka. As Kafka Connect will record offsets automatically, `SourceTask` is not required to implement them. In cases where a connector does need to acknowledge messages in the source system, only one of the APIs is typically required.

Sink Tasks

The previous section described how to implement a simple `SourceTask`. Unlike `SourceConnector` and `SinkConnector`, `SourceTask` and `SinkTask` have very different interfaces because `SourceTask` uses a pull interface and `SinkTask` uses a push interface. Both share the common lifecycle methods, but the `SinkTask` interface is quite different:

```
public abstract class SinkTask implements Task {
  ... [ lifecycle methods omitted ] ...

  public void initialize(SinkTaskContext context) {
      this.context = context;
  }

  public abstract void put(Collection<SinkRecord> records);
  public abstract void flush(Map<TopicPartition, Long> offsets);

  public void open(Collection<TopicPartition> partitions) {}
  public void close(Collection<TopicPartition> partitions) {}
}
```

The SinkTask documentation contains full details, but this interface is nearly as simple as the `SourceTask`. The `put()` method should contain most of the implementation, accepting sets of `SinkRecords`, performing any required translation, and storing them in the destination system. This process does not need to ensure the data has been fully written to the destination system before returning. In fact, in many cases some internal buffering will be useful so an entire batch of records can be sent at once (much like Kafka's producer), reducing the overhead of inserting events into the downstream data store. The `SinkRecords` contain essentially the same information as `SourceRecords`: Kafka topic, partition, and offset and the event key and value.

The `flush()` method is used during the offset commit process, which allows tasks to recover from failures and resume from a safe point such that no events will be missed. The method should push any outstanding data to the destination system and then block until the write has been acknowledged. The `offsets` parameter can often be ignored, but is useful in some cases where implementations want to store offset information in the destination store to provide exactly-once delivery. For example, an HDFS connector could do this and use atomic move operations to make sure the `flush()` operation atomically commits the data and offsets to a final location in HDFS.

Internally, `SinkTask` uses a Kafka consumer to poll data. The consumer instances used in tasks for a connector belong to the same consumer group. Task reconfiguration or failures will trigger rebalance of the consumer group. During rebalance, the topic partitions will be reassigned to the new set of tasks. For more explanations of the Kafka consumer rebalance, see the Consumer section.

Note that as the consumer is single threaded and you should make sure tha`put()` or `flush()` will not take longer than the consumer session timeout. Otherwise, the consumer will be kicked out of the group, which triggers a rebalancing of partitions that stops all other tasks from making progress until the rebalance completes.

To ensure that the resources are properly released and allocated during rebalance,`SinkTask` provides two additional methods:`close()` and `open()`, which are tied to the underlying rebalance callbacks of the`KafkaConsumer` that is driving the`SinkTask`.

The `close()` method is used to close writers for partitions assigned to the`SinkTask`. This method will be called before a consumer rebalance operation starts and after the `SinkTask` stops fetching data. After being closed, Connect will not write any records to the task until a new set of partitions has been opened. The `close()` method has access to all topic partitions assigned to the`SinkTask` before rebalance starts. In general, we recommend to close writers for all topic partitions and ensures that the states for all topic partitions are properly maintained. However, you can choose to close writers for a subset of topic partitions in your implementation. In this case, you need to carefully reason about the state before and after rebalance in order to achieve the desired delivery guarantee.

The `open()` method is used to create writers for newly assigned partitions in case of consumer rebalance. This method will be called after partition re-assignment completes and before the `SinkTask` starts fetching data.

Note that any errors raised from`close()` or `open()` will cause the task to stop, report a failure status, and the corresponding consumer instance to close. This consumer shutdown triggers a rebalance, and topic partitions for this task will be reassigned to other tasks of this connector.

Resuming from Previous Offsets

The `SourceTask` implementation included a partition ID (the input filename) and offset (position in the file) with each record. The framework uses this to commit offsets periodically so that, in the case of a failure, the task can recover and minimize the number of events that are reprocessed and possibly duplicated (or to resume from the most recent offset if Kafka Connect was stopped gracefully, e.g. in standalone mode or due to a

job reconfiguration, rebalancing of work, etc). This commit process is completely automated by the framework, but only the connector knows how to seek back to the right position in the input to resume from that location.

To correctly resume upon startup, the task can use the `SourceContext` passed into its `initialize()` method to access the offset data. In `initialize()`, we would add a bit more code to read the offset (if it exists) and seek to that position:

```
stream = new FileInputStream(filename);
Map<String, Object> offset = context.offsetStorageReader().offset(Collections.singletonMap(FILENAME_FIELD, filename));
if (offset != null) {
  Long lastRecordedOffset = (Long) offset.get("position");
  if (lastRecordedOffset != null)
    seekToOffset(stream, lastRecordedOffset);
}
```

There are two important points to note about this implementation. First, offsets for this connector are jus `Longs`, a primitive type. However, more complex structures including *Maps* and *Lists* can be used as offsets too. Second, the data returned is "schema-free". This is necessary because it cannot be guaranteed that the underlying `Converter` that serializes offsets can track schemas. This makes reliable parsing of offsets a bit more challenging for the connector developer, but makes the choice of serialization formats much more flexible.

Of course, you also might need to read many keys for each of the input partitions -- only a simple connector like this one will have one input partition. The OffsetStorageReader interface also allows you to issue bulk reads to efficiently load all offsets, then apply them by seeking each input partition to the appropriate position.

Dynamic Input/Output Partitions

Kafka Connect is intended to define bulk data copying jobs, such as copying an entire database rather than creating many jobs to copy each table individually. One consequence of this design is that the set of input or output partitions for a connector can vary over time.

Source connectors need to monitor the source system for changes, e.g. table additions/deletions in a database. When they pick up changes, they should notify the framework via the `ConnectorContext` object that reconfiguration is necessary. For example, in a `SourceConnector`:

```
if (inputsChanged()) {
  this.context.requestTaskReconfiguration();
}
```

The framework will promptly request new configuration information and update the tasks, allowing them to gracefully commit their progress before reconfiguring them. Note that in the `SourceConnector` this monitoring is currently left up to the connector implementation. If an extra thread is required to perform this monitoring, the connector must allocate it itself.

Ideally this code for monitoring changes would be isolated to the `Connector` and tasks would not need to worry about them. However, changes can also affect tasks, most commonly when one of their input partitions is destroyed in the input system, e.g. if a table is dropped from a database. If the `Task` encounters the issue before the `Connector`, which will be common if the `Connector` needs to poll for changes, the `Task` will need to handle the subsequent error. Thankfully, this can usually be handled simply by catching and handling the appropriate exception.

`SinkConnectors` usually only have to handle the addition of partitions, which may translate to new entries in their outputs. The Kafka Connect framework manages any changes to the Kafka input, such as when the set of input topics changes because of a regex subscription. `SinkTasks` should expect new input partitions, which may require creating new resources in the downstream system, such as a new table in a database. The trickiest situation to handle in these cases may be conflicts between multiple `SinkTasks` seeing a new input partition for the first time and simultaneously trying to create the new resource. `SinkConnectors`, on the other hand, will generally require no special code for handling a dynamic set of partitions.

Configuration Validation

Kafka Connect allows you to validate connector configurations before submitting a connector for execution and can provide feedback about errors and recommended values. To take advantage of this, connector developers need to provide an implementation of `config()` to expose the configuration definition to the framework.

The following code in `FileStreamSourceConnector` defines the configuration and exposes it to the framework.

```
private static final ConfigDef CONFIG_DEF = new ConfigDef()
    .define(FILE_CONFIG, Type.STRING, Importance.HIGH, "Source filename.")
    .define(TOPIC_CONFIG, Type.STRING, Importance.HIGH, "The topic to publish data to");

public ConfigDef config() {
    return CONFIG_DEF;
}
```

The ConfigDef class is used for specifying the set of expected configurations. For each configuration, you can specify the name, the type, the default value, the documentation, the group information, the order in the group, the width of the configuration value and the name suitable for display in the UI. Plus, you can provide special validation logic used for single configuration validation by overriding the Validator class. Moreover, as there may be dependencies between configurations, for example, the valid values and visibility of a configuration may change according to the values of other configurations. To handle this, `ConfigDef` allows you to specify the dependents of a configuration and to provide an implementation of Recommender to get valid values and set visibility of a configuration given the current configuration values.

The `validate()` method in `Connector` provides a default validation implementation which returns a list of allowed configurations together with configuration errors and recommended values for each configuration. However, it does not use the recommended values for configuration validation. You may provide an override of the default implementation for customized configuration validation, which may use the recommended values.

Working with Schemas

The FileStream connectors are good examples because they are simple, but they also have trivially structured data -- each line is just a string. Almost all connectors will need schemas with more complex data formats.

To create more complex data, you'll need to work with the `org.apache.kafka.connect.data` API. Most structured records will need to interact with two classes in addition to primitive types: Schema and Struct.

The API documentation provides a complete reference, but here is a simple example creating a `Schema` and `Struct`:

```
Schema schema = SchemaBuilder.struct().name(NAME)
    .field("name", Schema.STRING_SCHEMA)
    .field("age", Schema.INT_SCHEMA)
    .field("admin", new SchemaBuilder.boolean().defaultValue(false).build())
    .build();

Struct struct = new Struct(schema)
    .put("name", "Barbara Liskov")
    .put("age", 75)
    .build();
```

If you are implementing a source connector, you'll need to decide when and how to create schemas. Where possible, you should avoid recomputing them if possible. For example, if your connector is guaranteed to have a fixed schema, create it statically and reuse a single instance.

However, many connectors will have dynamic schemas. One example of this is a database connector. Considering even just a single table, the schema will not be fixed for a single table over the lifetime of the connector since the user may execute an `ALTER TABLE` command. The connector must be able to detect these changes and react appropriately by creating an updated `Schema`.

Sink connectors are usually simpler because they are consuming data and therefore do not need to create schemas. However, they should take just as much care to validate that the schemas they receive have the expected format. When the schema does not match -- usually indicating the upstream producer is generating invalid data that cannot be correctly translated to the destination system -- sink connectors should throw an exception to indicate this error to the Kafka Connect framework.

When using the `AvroConverter` included with Confluent Platform, schemas are registered under the hood with Confluent Schema Registry, so any new schemas must satisfy the compatibility requirements for the destination topic.

Schema Evolution

Kafka Connect provides utilities in SchemaProjector to project values using Kafka Connect's data API between compatible schemas and throw exceptions when non compatible schemas are provided. The usage of `SchemaProjector` is straightforward. The following example shows how to project `sourceStruct` to from source schema with version 2 to target schema with version 3, which adds a field with default value. As these two schemas are compatible, we see that the `targetStruct` has will have two fields with `field2` filled with `123`, which is the default value for that field.

```
Schema source = SchemaBuilder.struct()
     .version(2)
     .field("field", Schema.INT32_SCHEMA)
     .build();

Struct sourceStruct = new Struct(source);
sourceStruct.put("field", 1);

Schema target = SchemaBuilder.struct()
     .version(3)
     .field("field", Schema.INT32_SCHEMA)
     .field("field2", SchemaBuilder.int32().defaultValue(123).build())
     .build();

Struct targetStruct = (Struct) SchemaProjector.project(source, sourceStruct, target);
```

The utility is useful for connectors that needs to handle schema evolution and maintain schema compatibility. For example, if we want the HDFS connector to maintain backward compatibility, as each file can only have one schema, we need to project the message with older schema to the latest schema seen by the connector before the message is written to HDFS. This ensures that the latest file written to HDFS will have the latest schema that can be used to query the whole data, which maintains backward compatibility.

For more information on schema compatibility, see the Data Serialization and Schema Evolution section.

Testing

Testing connectors can be challenging because Kafka Connect connectors interact with two systems that may be difficult to mock -- Kafka and the system the connector is connecting to. It can be tempting to write "unit tests" that are really integration tests. It's better to very specifically test the functionality of the Connector and Task classes *independently* while mocking the external services.

Once you have sufficiently unit tested them, we recommend adding separate integration tests to verify end-to-end functionality.

Packaging

Once you've developed and tested your connector, you must package it so that it can be easily installed into Kafka Connect installations. The two techniques described here both work with Kafka Connect's *plugin path* mechanism.

If you plan to package your connector and distribute it for others to use, you are obligated to properly license and copyright your own code and to adhere to the licensing and copyrights of all libraries your code uses and that you include in your distribution.

Creating an Archive

The most common approach to packaging a connector is to create a tarball or ZIP archive. The archive should contain a single directory whose name will be unique relative to other connector implementations, and will therefore often include the connector's name and version. All of the JAR files and other resource files needed by the connector, including third party libraries, should be placed *within* that top-level directory. Note, however, that the archive should never include the Kafka Connect API or runtime libraries.

To install the connector, a user simply unpacks the archive into the desired location. Having the name of the archive's top-level directory be unique makes it easier to unpack the archive without overwriting existing files. It also makes it easy to place this directory on Kafka Connect's plugin path or for older Kafka Connect installations to add the JARs to the `CLASSPATH`.

Creating an Uber JAR

An alternative approach is to create an *uber JAR* that contains all of the connector's JAR files and other resource files. No directory internal structure is necessary.

To install, a user simply places the connector's uber JAR into one of the directories listed in Kafka Connect's plugin path.

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

10 Votes

Last updated on Oct 17, 2019.