Partner Development Guide for Kafka Connect

Overview

This guide is intended to provide useful background to developers implementing Kafka Connect sources and sinks for their data stores. It should be read in conjunction with the Verification Guide for Confluent Platform Integrations.

Revision 1.1 (February 2018)

Contact: partners@confluent.io

Revision 1.1 (February 2018)

Contact: partners@confluent.io

Developing a Verified Integration: The Basics

Coding

Connectors are most often developed in Java. While Scala is an acceptable alternative, the incompatibilities between Scala 2.x run-time environments might make it necessary to distribute multiple builds of your connector. Java 8 is recommended.

Confluent engineers have developed a Maven archetype to generate the core structure of your connector.

```
mvn archetype:generate -B -DarchetypeGroupId=io.confluent.maven \
    -DarchetypeArtifactId=kafka-connect-quickstart \
    -DarchetypeVersion=0.10.0.0 \
    -Dpackage=com.mycompany.examples \
    -DgroupId=com.mycompany.examples \
    -DartifactId=testconnect \
    -Dversion=1.0-SNAPSHOT
```

will create the source-code skeleton automatically, or you can select the options interactively with

```
mvn archetype:generate -DarchetypeGroupId=io.confluent.maven \
    -DarchetypeArtifactId=kafka-connect-quickstart \
    -DarchetypeVersion=0.10.0.0
```

This archetype will generate a directory containing Java source files with class definitions and stub functions for both Source and Sink connectors. You can choose to remove one or the other components should you desire a uni-directional connector. The archetype will also generate some simple unit-test frameworks that should be customized for your connector.

> Note on Class Naming : The Confluent Control Center supports interactive configuration of Connectors (see notes on Connector Configuration below). The naming convention that allows Control Center to differentiate sources and sinks is the use of SourceConnector and SinkConnector as Java classname suffixes (eg. JdbcSourceConnector and JdbcSinkConnector). Failure to use these suffixes will prevent Control Center from supporting interactive configuration of your connector.

Revision 1.1 (February 2018)

Contact: partners@confluent.io

Documentation and Licensing

The connector should be well-documented from a development as well as deployment perspective. At a minimum, the details should include

- Top-level README with a simple description of the Connector, including its data model and supported delivery semantics.
- Configuration details (this should be auto-generated via the toRst/toHtml methods for the ConfigDef object within the Connector). Many developers include this generation as part of the unit test framework.
- OPTIONAL: User-friendly description of the connector, highlighting the more important configuration options and other operational details
- Quickstart Guide : end-to-end description of moving data to/from the Connector. Often, this description will leverage the kafka-console-* utilities to serve as the other end of the data pipeline (or kafka-console-avro-* when the Schema Registry-compatible Avro converter classes are utilized).

See the JDBC connector for an example of comprehensive Connector documentation https://github.com/confluentinc/kafka-connect-jdbc

Most connectors will be developed to OpenSource Software standards, though this is not a requirement. The Kafka Connect framework itself is governed by the Apache License, Version 2.0. The licensing model for the connector should be clearly defined in the documentation. When applicable, OSS LICENSE files must be included in the source code repositories.

Unit Tests

The Connector Classes should include unit tests to validate internal API's. In particular, unit tests should be written for configuration validation, data conversion from Kafka Connect framework to any data-system-specific types, and framework integration. Tools like PowerMock (https://github.com/jayway/powermock) can be utilized to facilitate testing of class methods independent of a running Kafka Connect environment.

System Tests

System tests to confirm core functionality should be developed. Those tests should verify proper integration with the Kafka Connect framework:

- proper instantiation of the Connector within Kafka Connect workers (as evidenced by proper handling of REST requests to the Connect workers)
- schema-driven data conversion with both Avro and JSON serialization classes
- task restart/rebalance in the event of worker node failure

Advanced system tests would include schema migration, recoverable error events, and performance characterization. The system tests are responsible for both the data system endpoint and any necessary seed data:

- System tests for a MySQL connector, for example, should deploy a MySQL database instance along with the client components to seed the instance with data or confirm that data has been written to the database via the Connector.
- System tests should validate the data service itself, independent of Kafka Connect. This can be a trivial shell test, but definitely confirm that the automated service deployment is functioning properly so as to avoid confusion should the Connector tests fail.

Ideally, system tests will include stand-alone and distributed mode testing

- Stand-alone mode tests should verify basic connectivity to the data store and core behaviors (data conversion to/from the data source, append/overwrite transfer modes, etc.). Testing of schemaless and schema'ed data can be done in stand-alone mode as well.
- Distributed mode tests should validate rational parallelism as well as proper failure handling. Developers should document proper behavior of the connector in the event of

worker failure/restart as well as Kafka Cluster failures. If exactly-once delivery semantics are supported, explicit system testing should be done to confirm proper behavior.

- Absolute performance tests are appreciated, but not required.

The Confluent System Test Framework (https://cwiki.apache.org/confluence/display/KAFKA/tutorial+-+set+up+and+run+Kafka+system+tests+with+ducktape) can be leveraged for more advanced system tests. In particular, the ducktape framework makes testing of different Kafka failure modes simpler. An example of a Kafka Connect ducktape test is available here : https://github.com/apache/kafka/blob/trunk/tests/kafkatest/tests/connect/connect_distributed_test.py#L356 .

Packaging

The final connector package should have minimal dependencies. The default invocation of the Connect Worker JVM's includes the core Apache and Confluent classes from the distribution in CLASSPATH. The packaged connectors (e.g. HDFS Sink and JDBC Source/Sink) are deployed to share/java/kafka-connect-* and included in CLASSPATH as well. To avoid Java namespace collisions, you ***must* not** directly include any of the following classes in your connector jar :

- io.confluent.*
- org.apache.kafka.connect.*

In concrete terms, you'll want your package to depend only on the `connect-api` artifact, and that artifact should be classified as **`provided`**. That will ensure that no potentially conflicting jar's will be included in your package.

Kafka Connect 0.10.* and earlier does not support CLASSPATH isolation within the JVM's deploying the connectors. If you Connector conflicts with classes from the packaged connectors, you should document the conflict and the proper method for isolating your Connector at runtime. Such isolation can be accomplished by disabling the packaged connectors completely (renaming the share/java/kafka-connect-* directories) or developing a customized script to launch your Connect Workers that eliminates those directories from CLASSPATH.

Developers are free to distribute their connector via whatever packaging and installation framework is most appropriate. Confluent distributes its software as rpm/deb packages as well as a self-contained tarball for relocatable deployments. Barring extraordinary circumstances, Connector jars should be made available in compiled form rather than requiring end customers to build the connector on site. The helper scripts that launch Kafka Connect workers (`connect-standalone` and `connect-distributed`) explicitly add the connector jars to the CLASSPATH. By convention, jar files in share/java/kafka-connect-* directories are added automatically, so you could document your installation process to locate your jar files in share/java/kafka-connect-<MyConnector> .

Development Best Practices for Verified Integrations

Connector Configuration

Connector classes must define the `config()` method, which returns an instance of the `ConfigDef` class representing the required configuration for the connector. The `AbstractConfig` class should be used to simplify the parsing of configuration options. That class supports `get*` functions to assist in configuration validation. Complex connectors can choose to extend the `AbstractConfig` class to deliver custom functionality. The existing JDBCConnector illustrates that with its `JDBCConnectorConfig` class, which extends `AbstractConfig` while implementing the `getBaseConfig()` method to return the necessary `ConfigDef` object when queried. You can see how `ConfigDef` provides a fluent API that lets you easily define new configurations with their default values and simultaneously configure useful UI parameters for interactive use. An interesting example of this extensibility can be found in the `MODE_CONFIG` property within `JDBCConnectorConfig`. That property is constrained to one of 4 pre-defined values and will be automatically validated by the framework.

The `ConfigDef` class instance within the Connector should handle as many of the configuration details (and validation thereof) as possible. The values from `ConfigDef` will be exposed to the REST interface and directly affect the user experience in Confluent Control Center. For that reason, you should carefully consider grouping and ordering information for the different configuration parameters. Parameters also support `Recommender` functions for use within the Control Center environment to guide users with configuration recommendations. The connectors developed by the Confluent team (JDBC, HDFS, and Elasticsearch) have excellent examples of how to construct a usable `ConfigDef` instance with the proper information.

If the configuration parameters are interdependent, implementing a `<Connector>.validate()` function is highly recommended. This ensures that the potential configuration is consistent before it is used for Connector deployment. Configuration validation can be done via the REST interface before deploying a connector; the Confluent Control Center always utilizes that capability so as to avoid invalid configurations.

Revision 1.1 (February 2018)

Contact: partners@confluent.io

Schemas and Schema Migration

Type support

The Connector documentation should, of course, include all the specifics about the data types supported by your connector and the expected message syntax. Sink Connectors should not simply cast the fields from the incoming messages to the expected data types. Instead, you should check the message contents explicitly for your data objects within the Schema portion of the SinkRecord (or with `instanceof` for schemaless data). The `PreparedStatementBinder.bindRecord()` method in the JdbcSinkConnector provides a good example of this logic. The lowest level loop walks through all the non-key fields in the SinkRecords and converts those fields to a SQLCompatible type based on the Connect Schema type associated with that field:

```
for (final String fieldName : fieldsMetadata.nonKeyFieldNames) {
    final Field field = record.valueSchema().field(fieldName);
    bindField(index++, field.schema().type(), valueStruct.get(field));
}
```

Well-designed Source Connectors will associate explicit data schemas with their messages, enabling Sink Connectors to more easily utilize incoming data. Utilities within the Connect framework simplify the construction of those schemas and their addition to the SourceRecords structure.

The code should throw appropriate exceptions if the data type is not supported. Limited data type support won't be uncommon (e.g. many table-structured data stores will require a `Struct` with name/value pairs). If your code throws Java exceptions to report these errors, a best practice is to use `ConnectException` rather than the potentially confusing `ClassCastException`. This will ensure the more useful status reporting to Connect's RESTful interface, and allow the framework to manage your connector more completely.

Logical Types

Where possible, preserve the extra semantics specified by logical types by checking for `schema.name()`'s that match known logical types. Although logical types will safely fall back on the native types (e.g. a UNIX timestamp will be preserved as a long), often times systems will provide a corresponding type that will be more useful to users. This will be particularly true in some common cases, such as Decimal, where the native type (`bytes`) does not obviously correspond to the logical type.   The use of schema in these cases actually expands the functionality of the connectors ... and thus should be leveraged as much as possible.

Schemaless data

Connect prefers to associate schemas with topics and we encourage you to preserve those schemas as much as possible. However, you can design a connector that supports schemaless data.   Indeed, some message formats implicitly omit schema (eg JSON). You should make a best effort to support these formats when possible, and fail cleanly and with an explanatory exception message when lack of a schema prevents proper handling of the messages.

Sink Connectors that support schemaless data should detect the type of the data and translate it appropriately. The community connector for DynamoDB illustrates this capability very clearly in its `AttributeValueConverter` class. If the connected data store requires schemas and doesn't efficiently handle schema changes, it will likely prove impossible to handle implicit schema changes automatically.   It is better in those circumstances to design a connector that will immediately throw an error.   In concrete terms, if the target data store has a fixed schema for incoming data, by all means design a connector that translates schemaless data as necessary.   However, if schema changes in the incoming data stream are expected to have direct effect in the target data store, you may wish to enforce explicit schema support from the framework.

Schema Migration

Schemas will change, and your connector should expect this.

Source Connectors won't need much support as the topic schema is defined by the source system; to be efficient, they may want to cache schema translations between the source system and Connect's data API, but schema migrations "just work" from the perspective of the framework.   Source Connectors may wish to add some data-system-specific details to their error logging in the event of schema incompatibility exceptions.   For example, users could inadvertently configure two instances of the JDBC Source connector to publish data for table "FOO" from two different database instances.   A different table structure for FOO in the two databases would result in the second Source Connector getting an exception when attempting to publish its data ... and error message indicating the specific JDBC Source would be helpful.

Sink Connectors must consider schema changes more carefully. Schema changes in the input data might require making schema changes in the target data system before delivering any data with the new schema. Those changes themselves could fail, or they could require compatible transformations on the incoming data. Schemas in the Kafka Connect framework include a version. Sink connectors should keep track of the latest schema version that has arrived. Remember that incoming data may reference newer or older schemas, since data may be delivered from multiple Kafka partitions with an arbitrary mix of schemas. By keeping track of the Schema version, the connector can ensure that schema changes that have been applied to the target data system are not reverted. Converters within the Connector can the version of the schema for the incoming data along with the latest schema version observed to determine whether to apply schema changes to the data target or to project the incoming data to a compatible format. Projecting data between compatible schema versions can be done using the `SchemaProjector` utility included in the Kafka Connect framework. The `SchemaProjector` utility leverages the Connect Data API, so it will always support the full range of data types and schema structures in Kafka Connect.

Offset Management

Source Connectors

Source Connectors should retrieve the last committed offsets for the configured Kafka topics during the execution of the `start()` method. To handle exactly-once semantics for message delivery, the Source Connector must correctly map the committed offsets to the Kafka cluster with some analog within the source data system, and then handle the necessary rewinding should messages need to be re-delivered. For example, consider a trivial Source connector that publishes the lines from an input file to a Kafka topic one line at a time ... prefixed by the line number. The `commit*` methods for that connector would save the line number of the posted record ... and then pick up at that location upon a restart.

An absolute guarantee of exactly once semantics is not yet possible with Source Connectors (there are very narrow windows where multiple failures at the Connect Worker and Kafka Broker levels could distort the offset management functionality). However, the necessary low-level changes to the Kafka Producer API are being integrated into Apache Kafka Core to eliminate these windows.

Sink Connectors

The proper implementation of the `flush()` method is often the simplest solution for correct offset management within Sink Connectors. So long as Sinks correctly ensure that messages delivered to the `put()` method before `flush()` are successfully saved to the target data store before returning from the `flush()` call, offset management should "just work". A conservative design may choose not to implement `flush()` at all and simply manage offsets with every `put()` call. In practice, that design may constrain connector performance unnecessarily.

Developers should carefully document and implement their handling of partitioned topics. Since different connector tasks may receive data from different partitions of the same topic, you may need some additional data processing to avoid any violations of the ordering semantics of the target data store. Additionally, data systems where multiple requests can be "in flight" at the same time from multiple Connector Task threads should make sure that relevant data ordering is preserved (eg not committing a later message while an earlier one has yet to be confirmed). Not all target data systems will require this level of detail, but many will.

Exactly-once semantics within Sink Connectors require atomic transactional semantics against the target data system ... where the known topic offset is persisted at the same time as the payload data. For some systems (notably relational databases), this requirement is simple. Other target systems require a more complex solution. The Confluent-certified HDFS connector offers a good example of supporting exactly-once delivery semantics using a connector-managed commit strategy.

Converters and Serialization

Serialization formats for Kafka are expected to be handled ***separately*** from connectors. Serialization is performed after the Connect Data API formatted data is returned by Source Connectors, or before Connect Data API formatted data is delivered to Sink Connectors. Connectors should not assume a particular format of data. However, note that Converters only address one half of the system. Connectors may still choose to implement multiple formats, and even make them pluggable. For example, the HDFS Sink Connector (taking data from Kafka and storing it to HDFS) does not assume anything about the serialization format of the data in Kafka. However, since that connector is responsible for writing the data to HDFS, it can handle converting it to Avro or Parquet, and even allows users to plug in new format implementations if desired. In other words, Source Connectors might be flexible on the format of source data and Sink Connectors might be flexible on the format of sink data, but both types of connectors should let the Connect framework handle the format of data within Kafka.

There are currently two supported data converters for Kafka Connect distributed with Confluent: `org.apache.kafka.connect.json.JsonConverter` and `io.confluent.connect.avro.AvroConverter` . Both converters support including the message schema along with the payload (when configured with the appropriate `*.converter.schemas.enable` property to true).
The `JsonConverter` includes the schema details as simply another JSON value in each record. A record such as " {"name":"Alice","age":38} " would get wrapped to the longer format

```
{
    "schema":{"type":"struct",
            "fields":[{"type":"string","optional":false,"field":"name
"},{"type":"integer","optional":false,"field":"age"}],
            "optional":false,
            "name":"htest2"},
  "payload":{"name":"Alice","age":38}
}
```

Connectors are often tested with the `JsonConverter` because the standard Kafka consumers and producers can validate the topic data.

Confluent's `AvroConverter` uses the SchemaRegistry service to store topic schemas, so the volume of data on the Kafka topic is much reduced. The SchemaRegistry enabled Kafka clients (eg `kafka-avro-console-consumer`) can be used to examine these topics (or publish data to them).

Headers

As of Apache Kafka 1.1, Connect will support the use of record headers: source connectors will be able to generate records with headers, and sink connectors will be able to access the headers on the records it sees. Consider whether you're connector should support headers, since this may be attractive for storing and accessing additional information outside of the record keys and values. Supporting headers will also constrain the Connect runtime version in which your connector can be deployed.

Single Message Transforms

Many Connect users may want to make alterations to the records produced by source connectors or those sent to sink connectors. Connect provides *Single Message Transforms* (SMTs) that allow this kind of customization, either by using the built-in SMTs, SMTs that are written as separate plugins, or as custom SMTs provided by your connector. Connect allows users to optionally configure multiple SMTs on any connector, and they are completely independent of the connectors.

SMTs may affect what customizations and options you want to support in your connector. For example, a user of a source connector might want the ability to customize the topic names for the generated records. When developing a source connector, you might choose to rely upon SMTs to keep your connector as simple as possible, or you might decide that this is a common requirement for your connector and want to make it easy for users to do this without using an SMT.

Parallelism

Most connectors will have some sort of inherent parallelism. A connector might process many log files, many tables, many partitions, etc. Connect can take advantage of this parallelism and automatically allow your connector to scale up more effectively – ***IF*** you provide the framework the necessary information.

Sink connectors need to do little to support this because they already leverage Kafka's consumer groups functionality; recall that consumer groups automatically balance and scale work between member consumers (in this case Sink Connector tasks) as long as enough Kafka partitions are available on the incoming topics.

Source Connectors, in contrast, need to express how their data is partitioned and how the work of publishing the data can be split across the desired number of tasks for the connector. The first step is to define your input data set to be broad by default, encompassing as much data as is sensible given a single configuration. This provides sufficient partitioned data to allow Connect to scale up/down elastically as needed. The second step is to use your `Connector.taskConfigs()` method implementation to divide these source partitions among (up to) the requested number of tasks for the connector.

Explicit support of parallelism is not an absolute requirement for Connectors – some data sources simply do not partition well. However, it is worthwhile to identify the conditions under which parallelism is possible. For example, a database might have a single WAL file which seems to permit no parallelism for a change-data-capture connector; however, even in this case we might extract subsets of the data (e.g. per table from a DB WAL) to different topics, in which case we can get some parallelism (split tables across tasks) at the cost of the overhead of reading the WAL multiple times.

Error Handling

The Kafka Connect framework defines its own hierarchy of throwable error classes (https://kafka.apache.org/0100/javadoc/org/apache/kafka/connect/errors/package-summary.html). Connector developers should leverage those classes (particularly `ConnectException` and `RetriableException`) to standardize connector behavior. Exceptions caught within your code should be rethrown as `connect.errors` whenever possible to ensure proper visibility of the problem outside the framework. Specifically, throwing a `RuntimeException` beyond the scope of your own code should be avoided because the framework will have no alternative but to terminate the connector completely.

Recoverable errors during normal operation can be reported differently by sources and sinks. Source Connectors can return `null` (or an empty list of `SourceRecords`) from the `poll()` call. Those connectors should implement a reasonable backoff model to avoid wasteful Connector operations; a simple call to `sleep()` will often suffice. Sink Connectors may throw a `RetriableException` from the `put()` call in the event that a subsequent attempt to store the SinkRecords is likely to succeed. The backoff period for that subsequent `put()` call is specified by the timeout value in the `sinkContext`. A default timeout value is often included with the connector configuration, or a customized value can be assigned using `sinkContext.timeout()` before the exception is thrown. Connectors that deploy multiple threads should use `context.raiseError()` to ensure that the framework maintains the proper state for the Connector as a whole. This also ensures that the exception is handled in a thread-safe manner.

Verification Process

See **Verification Guide for Confluent Platform Connectors & Integrations**

Post-Certification Evangelism

Confluent is happy to support Connect Partners in evangelizing their work. Activities include

- Blog posts and social media amplification
- Community education (meet-ups, webinars, conference presentations, etc)
- Press Releases (on occasion)
- Cross-training of field sales teams