CONFLUENT

Verified Integration Program Verification Guide: Sources and Sinks using Kafka Connect

Revision History

Versions	Date	Author	Description
0.1	May 16 2019	jwfbean@confluent.io	Merged content from prior integration verification guide, partner developer guide for connectors, and connector development process guide.
0.2	June 4 2019	jwfbean@confluent.io	Separation of checklist from verification guide.
0.5	March 11 2020	jwfbean@confluent.io	Cloud readiness criteria

Overview

This document describes the criteria that Confluent uses in verifying integrations that use the Kafka Connect framework. Examples of these integrations include:

- Streaming CDC data into Kafka from a database such as an RDBMS
- Streaming data from Kafka into a target datastore

Because of the standardization it offers, Kafka Connect is Confluent's preferred framework for producing and consuming data into Apache Kafka. Integrations built using Kafka Connect are referred to as "Connectors".

This guide defines the criteria Confluent employs to verify a connector with at the gold level.

Why Verification?

Verifying connectors is a mutually beneficial relationship between Confluent and its partners, as well as their direct customers. The verification of a connector not only provides assurances of a level of compatibility and functionality with the Confluent Platform ecosystem, but it signals to customers that there is an established mutually supported integration between Confluent Platform and the partner product.

What does it mean to be Verified?

Confluent maintains a set of best practices for connector development. Verification measures a connector's design and implementation against those practices. A gold verification indicates that Confluent has determined whether and how the connector adheres to each practice. Confluent publishes the best practice checklist in an integration document for each verified connector.

Verification Criteria Detail

Implement the Kafka Connect API? (REQUIRED)

In order to qualify for gold verification as a source or a sink, the partner must implement the Kafka Connect API. Kafka Connect is designed to handle many of the common integration requirements with Apache Kafka, including offset tracking, data serialization, schema management, etc.

Source or Sink (REQUIRED)

Kafka Connect Connectors are designated as either source or sink connectors, which either produce or consume data with respect to Apache Kafka. The partner is required to designate whether their connector is a source, sink or both.

Fully supported (REQUIRED)

By verifying a connector with us, the partner pledges to fully support the connector in conjunction with Confluent.

Technical and Business Contacts (REQUIRED)

In order to facilitate ongoing support and go-to-market activities, the partner is asked for contact information. We'll reach out to the technical contact if the connector is referenced in a support or pre-sales context. We'll reach out to the business contact for other activities.

License (REQUIRED)

We ask the partner to share the licensing and distribution model under which they're releasing the connector. This helps us to better position and understand the offering in the context of our ecosystem.

Functionality

Configurable? (REQUIRED)

Connectors should properly and completely define a ConfigDef to include all configuration items, including documentation and default values, organized into meaningful groups. We also follow some best practices for designing the configuration properties for a connector, including:

1. Expose the fewest configuration properties required to configure a connector, keeping it as simple as possible but not too simple.
2. Always define a short, human readable display name in title case (e.g., “Database Name”).
3. Always define a meaningful, human readable documentation string that is short enough but also complete enough for most users to understand what the configuration does.
4. Always use the appropriate and restrictive type for the configuration property (e.g., “int”, “string”, “double”, “list”, etc.). Always use “PASSWORD” type for secrets and credentials. (Some property values, like JDBC URLs, can contain passwords or secrets. In this case, it’s better if the different segments within JDBC URL were represented as separate configuration properties. This doesn’t always work.)
5. Provide a validator for most configuration properties to provide the user immediate feedback when creating a configuration. Ideally, a connector whose configuration validates will always run, barring any transient problems communicating with the backend. Where possible, the value of a configuration property should not be case-sensitive (e.g., “avro” and “Avro” and “AVRO” might all be valid).
6. Provide a recommender when the configuration property has a limited number of choices for values. Always provide a recommender for enumerated literals. The recommender can communicate with the external service, but be careful that this never takes more than a few seconds, since this may block the connector and the Connect worker thread.
7. Use dependencies between configuration properties to dynamically “show” or “hide” configuration properties based upon the value of another configuration property. For example, consider a connector property “mode” that can be set to either “default” or “exact”, and another property “mode.exact.range” should apply only for the “exact” mode. In this case, the “mode.exact.range” can be dependent upon the “mode” property, such that “mode.exact.range” is hidden when “mode=default” and unhidden when “mode=exact”. This is used in Control Center and other tools driven entirely by ConfigDef.

8. Use appropriate importance, which generally correspond to:
    a. HIGH for configuration properties that are important enough to always consider,
    b. MEDIUM for configuration properties that have defaults that are somewhat advanced, and whose defaults should be acceptable for users who aren't sure what to pick, and
    c. LOW for configuration properties that have defaults and are typically needed in advanced cases.
9. Use defaults wherever possible, especially on MEDIUM and LOW importance properties.
10. Organize related configuration properties into groups, with human-readable group names. Specify the order of properties within each group, starting with those properties that have a HIGH importance, then MEDIUM, then LOW importance.
11. Prefer enumerations over boolean properties, or sets of boolean properties. Boolean properties are not as flexible for future expansion, and sometimes lead to deprecating older boolean properties in favor of other newer properties. Properties with enumerated literals for values can easily be expanded over time. For example, consider the JDBC source connector's older `numeric.precision.mapping=true|false` property, which couldn't be expanded to handle additional mapping strategies, and which was superseded by `numeric.mapping=none|best_fit|precision_only`.
12. If a connector has pluggable/extensible components (e.g., like HDFS and S3 connectors' `format.class` and `partitioner.class` properties, make sure the interface for those classes extends the `org.apache.kafka.common.Configurable` interface and defines a `config():ConfigDef` method. Then, the ConfigDef for the connector can dynamically discover the ConfigDef and configuration properties of these pluggable components, when they are chosen by the user. This is the only way to provide a good user experience in Control Center and other ConfigDef-driven tools, where the ConfigDef must always describe all of the configuration properties that are needed for a connector.

13. When pluggable components have their own configuration options, always use prefixes for the properties passed to that component. For example, the Connect worker configuration allows the user to choose converters for record keys and values, via the `key.converter` and `value.converter` properties. All properties that begin with the `key.converter.` prefix are passed (after removing the prefix) to the converter when it is configured. This style of prefixing makes it easy for the connector to determine which properties should be passed to the component, and then pass only those properties to that component. This is also a pattern used more recently in Apache Kafka.

Connectors should always parse and use properties via connector-specific custom subclasses of Apache Kafka's AbstractConfig, which at a minimum ensures that password-type properties are never logged and tracks which properties are used and unused by the connector. The AbstractConfig class has methods for obtaining subsets of the original properties as Maps, optionally removing prefixes, that still will track usage. See the "JdbcSourceConnectorConfig" class for an example of a subclass of "AbstractConfig" that follows most of the above best practices.

We consider configuration and ConfigDefs as public APIs, so it's critical that we maintain backward compatibility when releasing new versions. Any configuration created by a customer for an older version should work with all newer versions of the connector, with no changes. Yes, the user might want to edit their configuration to utilize more recently-added configurations, and it's possible some configurations were deprecated and any of these in the user's connector configuration might not be used in newer versions of the connector. Some configurations can even be removed (preferably after a deprecation process) once the connector no longer uses them to control behavior.

Data Types

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.

Usable with Avro or JSON converters?

For optimal utilization with the Confluent platform including the schema registry, support for Avro or Json is suggested. 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"},{"ty
pe":"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).

Uses 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.

Exactly-once support?

Kafka Connect allows sink connectors to optionally implement exactly once behavior by tracking in the external system the offsets for each topic partition. Most users would prefer exactly once if given the choice, so where possible and where feasible sink connectors should implement this behavior.

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.

Internals and Supportability

Error handling (REQUIRED)

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.

Logging (REQUIRED)

Connectors are expected to provide adequate and useful logging at ERROR, WARN, INFO, DEBUG, and TRACE levels. The default logging level for Confluent Platform is INFO, and by default customers should be able to understand when the connector has started and/or stopped, and is processing records, including with messages describing important activities and changes in state or behavior. WARN log messages should be used only when a potential problem exists and should include actions the user can take to correct the problem. ERROR log messages should be used for all problems and errors, ideally with suggested actions that the user can take to correct the problem. Connectors must never log any record data at ERROR, WARN, INFO, or DEBUG level. Connectors may log schemas at any level.

When a customer is having a problem with the connector, ideally they could enable DEBUG and learn more about what the connector is and is not doing, including the number of records being processed (or not processed) and the operations against the external system. TRACE level should not be required for normal problem-solving activities.

Metrics

Kafka Connect introduced framework-level metrics, and has yet to provide an API so connectors can easily log metrics using the same framework. It is possible, however, for connectors to use the same low-level metrics library that Kafka ships with to provide their own connector-specific metrics. We have yet to make use of this, and need to decide whether to move in this direction or to add to Kafka Connect a simple metrics API for connectors.

Graceful back-off

Connectors should always deal with connectivity and other transient problems without failing or stopping the connector. Instead, Connectors should simply try to re-establish connectivity using a backoff technique. The developer should also consider adding randomness to the timeouts (e.g., jitter), to prevent a large number of connectors talking to the same backend system forming a thundering herd when all lose connectivity at about the same time and then, with a fixed backoff interval, all try to re-establish connectivity at about the same time and potentially causing another outage of the backend system. Introducing random jitter also means that the specifics of the retries do not need to be configured by the user, thus simplifying the connector configuration at least a bit.

A simpler option is to throw a RetriableException, which is a special subtype of ConnectException, and the Connect framework will retry the same connector operation again. This may work well for source connectors, since they've not yet returned any records for the current poll. For sinks, however, this means they'd be passed the same set of records again, even if the sink connector had successfully written some of these to the backend system. Therefore, for sink connectors, it's almost always best for the connector to retry on its own.

Cloud Readiness

To be considered for hosting in Confluent Cloud, the following criteria needs to be met:

1. **Rich Validations:** The Kafka Connect validation API needs to be fully and completely implemented so as to ensure proper integration with the Confluent Cloud UI. Any input provided by the user should be validated not only for correctness but also for completeness. For example, a validation should not just check that a host URL for an external system is well-formed, but also that the system is live and can be accessed with the user-supplied credentials. .
2. **ConfigProvider:** The connector should support injecting Credentials via the Apache Kafka ConfigProvider class. Confluent Cloud stores credentials in a secret store and injects it dynamically.

Packaging (REQUIRED)

All connectors are also packaged and made available via Confluent Hub. The packaging specification is available in our documentation, and we have created a Maven Packaging Plugin that makes it easy for Maven-based builds to specify the package metadata and automatically build the artifacts.

Testing (REQUIRED)

Although we require visibility into how the connector was tested we don't specifically require everything listed below in order to achieve verification. Treat this as a comprehensive guide for completely testing most connectors.

Verification Tests (REQUIRED)

The process of verifying a connector entails making claims about the connector and verifying that those claims are in fact true via reproducible tests. For every criteria marked REQUIRED, a test for that criteria should be documented. Taken in total, this set of tests serve as "verification tests". The document should provide everything required to reproduce these tests and should indicate the expected results. See the document "Verification Checklist and Results: Gold verifications using Kafka Connect" for an example template.

Unit tests

Confluent runs unit tests as part of our Maven builds and use the JUnit framework. You can get fairly good coverage using JUnit and it's good to design for testability. 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.

Confluent Platform integration tests

Some manual testing is expected to verify the connector works properly with Control Center, and that all properties can be set and connectors managed only via the Control Center user interface.

Performance and Bound Tests

It's good to define the bounds of a connector by running the following workload. You can optionally record the number of topics, tasks, and topic partitions set during the test scenario and submit it as part of your verification.

Number of Workers	3
CPU	8 cores
Heap	6 GB

Message Size	Throughput (messages/sec)		Number of Topics	Number of Topic partitions per topic	Number of Tasks	Comments
100 bytes	25,000	2.5 MB/sec				
100 bytes	50,000	5 MB/sec				
100 bytes	100,000	10 MB/sec				
100 bytes	250,000	25 MB/sec				
100 bytes	500,000	50 MB/sec				
100 bytes	1,00,0000	100 MB/sec				
5 kb	500	2.5 MB/sec				
5 kb	1000	5 MB/sec				
5 kb	2000	10 MB/sec				

5 kb	5000	25 MB/sec				
5 kb	10,000	50 MB/sec				
5 kb	20,000	100 MB/sec				
50 kb	50	2.5 MB/sec				
50 kb	100	5 MB/sec				
50 kb	200	10 MB/sec				
50 kb	500	25 MB/sec				
50 kb	1000	50 MB/sec				
50 kb	2000	100 MB/sec				

Documentation

The following information should be provided as part of any connector's standard documentation.

Product Brief (REQUIRED)

A two page document detailing the contact info and high level design / architecture of the connector, including:

- Connect API version used
- Confluent Platform supported versions
- Partner Product supported versions

Additional Documentation

Supported data types

Documentation should include all the specifics about the data types supported by your connector and the expected message syntax.

Schema evolution and compatibility policies

Confluent schema registry supports a variety of different compatibility policies as detailed here:

https://docs.confluent.io/current/schema-registry/avro.html

Documentation should state how the connector behaves with each compatibility policy and whether that compatibility policy is officially supported by the connector.

Configurations

In addition to the self-describing configuration settings, a configuration reference is a good idea. Additionally, this can be a tuning and performance guide, including suggested connector, worker, producer, and consumer settings.

Quickstart guide / Tutorial (REQUIRED)

Covering getting started with the connector and its basic usage.

Development Resources

- Kafka Connect overview, concepts and architecture
- Kafka Connect developer guide
- Sample sink connector: S3
- Sample sink connector: Elasticsearch
- Sample source connector: JDBC
- Sample source and sink connector: Couchbase

Verification Process

The verified integration program is broken up into the following phases. This details the meaning of each phase and the criteria for exiting it.

Initiated

A connector initiative has been identified and the partner and Confluent have agreed to work together on it.
Deliverable: A 2 page product brief describing the connector and the partner information.

Guidance

The connector is actively being developed. Confluent can provide both development (Q&A) support as well as process support.
Deliverable: A connector for verification with the checklist below.

Testing

The connector development is complete and verification testing is underway by the partner or by Confluent.
Deliverable: A detailed test document illustrating how the criteria are met. See the document entitled “Verification Checklist and Results: Gold verifications using Kafka Connect” for an example.

Submitted

The connector is being evaluated by Confluent. There may be some revision and further discussion in this process.
Deliverable: A completed verification report including a filled out checklist of verification criteria.

Verified

The connector has been verified by Confluent.
Deliverable(s): A card on the Confluent hub. An entry in the Confluent internal repository for field enablement. Blogs, co-marketing, etc.

Questions? Email vip@confluent.io