Installing and Configuring Kafka Connect

This section describes how you can install and configure a Kafka Connect instance.

Getting Started

The quick start describes how to get started in standalone mode. It demonstrates an end-to-end job, importing data from one system (the filesystem) into Apache Kafka®, then exporting that same data from the Kafka topic to another system (the console). This section is for users comfortable with the concepts and quick start.

Refer to additional resources to configure Kafka Connect withsecurity.

See also

You may run end-to-end demos with Kafka Connect on-prem, with Confluent Cloud, or with Confluent Operator.

This page covers the following:

- Planning a Kafka Connect installation
- Running workers in standalone and distributed modes
- Installing connector plugins
- Configuring workers

Planning for Installation

When getting going with Kafka Connect, there are a few considerations to be aware of to help your environment scale to the long term needs of your data pipeline. This section aims to provide some context around those decisions.

Prerequisites

Kafka Connect has only one hard prerequisite in order to get started: a set of Kafka brokers. However, as your cluster grows, there are a couple of items that are helpful to consider ahead of time:

Internal Topic Creation

As we will talk about in more detail it is important to create Kafka Connect's required internal topics ahead of time with a high replication factor, a compaction cleanup policy, and correct number of partitions. This helps avoid recalibrating these topics later on.

Schema Registry

Although Schema Registry is not a required service for Kafka Connect, it enables you to easily use Avro as the common data format for all connectors. This keeps the need to write custom code at a minimum and standardizes your data in a flexible format. Additionally, you get the benefits of enforced compatibility rules for schema evolution.

Standalone vs. Distributed

As we discussed in the concepts section, workers can be run in two different modes. It is useful to identify which mode works best fo environment before getting started. For development or environments that lend themselves to single agents (e.g. sending logs from v to Kafka), standalone mode is well suited. In use cases where a single source or sink may require heavy data volumes (e.g. sending d

Kafka to HDFS), distributed mode is more flexible in terms of scalability and offers the added advantage of a highly available service to minimize downtime. In the end, the choice is up to the installer, but knowing what you are moving and where you are moving it with Kafka Connect will help inform this decision. We recommend distributed mode for production deployments for ease of management and scalability.

Deployment Considerations

Kafka Connect workers can be deployed in a number of ways, each with their own benefits. Workers lend themselves well to being run in containers in managed environments such as Kubernetes, Mesos, Docker Swarm, or YARN as all state is stored in Kafka, making the local processes themselves stateless. We provide Docker images and documentation for getting started with those images is here. By design, Kafka Connect does not automatically handle restarting or scaling workers which means your existing clustering solutions can continue to be used transparently.

Additionally, Kafka Connect workers are simply JVM processes and as such can be run on shared machines that have sufficient resources. The resource limit depends heavily on the types of connectors being run by the workers, but in most cases users should be aware of CPU and memory bounds when running workers concurrently on a single machine.

Hardware requirements for Kafka Connect workers are similar to that of the standard Java producers and consumers. For those deployments that are expected to send large messages, more memory will be required. Those that make heavy use of compression will require more powerful CPUs. We recommend starting with the default heap size setting and monitoring the JMX metrics and the system to be sure CPU, memory, and network (recommend 10GbE and up) are sufficient for load.

Installing Plugins

Confluent Platform ships with commonly used connectors, transforms, and converters that have been tested with the rest of the platform. Kafka Connect is designed to be extensible so that it is easy for other developers to create new plugins with custom connectors, transforms, and/or converters, and easy for users to use them with minimal effort.

A Kafka Connect plugin is simply a set of JAR files where Kafka Connect can find an implementation of one or more connectors, transforms, and/or converters. Kafka Connect isolates each plugin from one another so that libraries in one plugin are not affected by the libraries in any other plugins. This is very important when mixing and matching connectors from multiple providers.

A **Kafka Connect plugin** is:

1. an **uber JAR** containing all of the classfiles for the plugin and its third-party dependencies in a single JAR file; or
2. a **directory** on the file system that contains the JAR files for the plugin and its third-party dependencies.

However, a plugin should never contain any libraries that are provided by Kafka Connect's runtime.

Kafka Connect finds the plugins using its *plugin path*, which is a comma-separated list of directories defined in the Kafka Connect's worker configuration. To install a plugin, place the plugin directory or uber JAR (or a symbolic link that resolves to one of those) in a directory listed on the plugin path, or update the plugin path to include the absolute path of the directory containing the plugin.

For example, we might choose to create a `/usr/local/share/kafka/plugins` directory on each machine and then place in each of them all of our plugin uber JARs or plugin directories. We tell each Kafka Connect worker about these plugins by defining the plugin path property in the workers' configuration file:

```
plugin.path=/usr/local/share/kafka/plugins
```

Now, when we start our Kafka Connect workers with that configuration, Kafka Connect will discover all connectors, transforms, and/or converters defined within those plugins. When we use a connector, transform, or converter, the Kafka Connect worker loads the classes from the respective plugin first, followed by the Kafka Connect runtime and Java libraries. Kafka Connect explicitly avoids all of the libraries in *other* plugins and prevents conflicts, making it very easy to use connectors and transforms developed independently by different providers.

Earlier versions of Kafka Connect required a different approach to installing connectors, transforms, and converters. All the scripts for running Kafka Connect recognized the `CLASSPATH` environment variable, which you would use to define the list of paths for your connectors' JARs files:

```
export CLASSPATH=/path/to/my/connectors/*
bin/connect-standalone standalone.properties new-custom-connector.properties
```

Using the `CLASSPATH` still works, but can easily result in conflicts that break Kafka Connect or individual connectors. That's because the scripts combine *all* of the libraries you define on the `CLASSPATH` with *all* of the libraries for Kafka Connect itself*and* libraries for all of the connectors, transforms, and converters that come with the platform. Some third party libraries are fairly common, so as you install connectors developed by different providers via the same `CLASSPATH`, the chance increases that multiple versions of the same library will appear on the classpath. And when that happens, only the first version to appear on the classpath is what gets used, likely resulting in a failure of one of the connectors or even Kafka Connect itself when it uses one of the versions of the library.

The new *plugin path* mechanism works around these problems by isolating each plugin from the other plugins and libraries. So whenever possible, install Kafka Connect plugins using the *plugin path* as described above.

Important

You must install the plugins and update the `CLASSPATH` on all of the machines where Kafka Connect is running. Kafka Connect can be run as a distributed cluster that will run a connector's tasks on any of the workers. Every connector, transform, and converter that you use on a cluster must be available on all workers.

Running Workers

Standalone Mode

To execute a worker in standalone mode, run the following command:

```
bin/connect-standalone worker.properties connector1.properties [connector2.properties connector3.properties ...]
```

The first parameter is always a worker configuration file as described below. Note that `worker.properties` is an example file name. You can use any valid file name for your properties file. This file gives you control over settings such as the Kafka cluster to use and serialization format. For an example configuration file that uses Avro and Schema Registry in a standalone mode configuration file, see `etc/schema-registry/connect-avro-standalone.properties`. You can copy and modify this file for use as your standalone worker properties file.

All additional parameters are connector configuration files. Each file contains a single connector configuration.

If you run multiple standalone instances on the same host, there are a couple of settings that must be unique between each instance:

- `offset.storage.file.filename` - storage for connector offsets, which are stored on the local filesystem in standalone mode; using the same file will lead to offset data being deleted or overwritten with different values
- `rest.port` - the port the REST interface listens on for HTTP requests

Distributed Mode

The distributed workers are stateless and store connector and task configurations, offsets, and status within internal Kafka topics. Earlier versions of Kafka Connect required you to manually create these topics, but now Kafka Connect can do this automatically when it starts up. You can control the names, replication factor, and number of partitions for these topics in the Kafka Connect distributed worker configuration, and all topics are created with compaction cleanup policy.

You may still want to manually create these Kafka topics before starting Kafka Connect if, for example, your broker may be configured to not

allow clients such as Kafka Connect to create topics, or you require other advanced topic-specific settings that are not set by Kafka Connect and that are different than the broker's auto-created topic settings. In these cases, we recommend you use the following commands to create these compacted and replicated Kafka topics manually before starting Kafka Connect, following the guidelines described below:

```
# config.storage.topic=connect-configs
  bin/kafka-topics --create --zookeeper localhost:2181 --topic connect-configs --replication-factor 3 --partitions 1 -
-config cleanup.policy=compact

# offset.storage.topic=connect-offsets
  bin/kafka-topics --create --zookeeper localhost:2181 --topic connect-offsets --replication-factor 3 --partitions 50
--config cleanup.policy=compact


# status.storage.topic=connect-status
  bin/kafka-topics --create --zookeeper localhost:2181 --topic connect-status --replication-factor 3 --partitions 10 -
-config cleanup.policy=compact
```

Create a worker configuration file just as you would with standalone mode, except specifying the options for distributed workers. Then, start the worker process with the worker configuration file you created. For an example configuration file that uses Avro and Schema Registry in a distributed mode configuration file, see `etc/schema-registry/connect-avro-distributed.properties`. You can make a copy of this file, modify it, use it as the new `worker.properties` file, and start it exactly like a worker in standalone mode is started. Note that `worker.properties` is an example file name. You can use any valid file name for your properties file.

```
bin/connect-distributed worker.properties
```

Distributed mode does not have any additional command line parameters other than a worker configuration file. New workers will either start a new group or join an existing one based on the worker properties provided. Workers then coordinate similarly to consumer groups to distribute the work to be done. This is different from standalone mode where users may optionally provide connector configurations at the command line as only a single worker instance exists and no coordination is required in standalone mode. Use the REST API to deploy and manage the connectors when running in distributed mode as described here.

In distributed mode, if you run more than one worker per host (for example, if you are testing distributed mode locally during development), the following settings must have different values for each instance:

- `rest.port` - the port the REST interface listens on for HTTP requests

---

Configuring Workers

Whether you're running standalone or distributed mode, Kafka Connect workers are configured by passing a properties file containing any required or overridden options as the first parameter to the worker process. Some example configuration files are included with Confluent Platform to help you get started. We recommend using the files `etc/schema-registry/connect-avro-[standalone|distributed].properties` as a starting point because they include the necessary configuration to use Confluent Platform's Avro converters that integrate with Schema Registry. They are configured to work well with Kafka and Schema Registry services running locally and do not require running more than a single broker, making it easy to test Kafka Connect locally. These example configurations can also be easily adapted for production deployments by using the correct hostnames for Kafka and Schema Registry and acceptable (or default) values for the internal topics' replication factors.

Many more configuration properties are provided inReferences. This section describes a few of the more-commonly updated properties.

Common Worker Configs

`bootstrap.servers`

A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping. The list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).

- Type: list
- Default: [localhost:9092]
- Importance: high

`key.converter`

Converter class for key Connect data. This controls the format of the data that will be written to Kafka for source connectors or read from Kafka for sink connectors. Popular formats include Avro and JSON.

- Type: class
- Default:
- Importance: high

`value.converter`

Converter class for value Connect data. This controls the format of the data that will be written to Kafka for source connectors or read from Kafka for sink connectors. Popular formats include Avro and JSON.

- Type: class
- Default:
- Importance: high

`internal.key.converter`

Converter class for internal key Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.

- Type: class
- Default:
- Importance: low

`internal.value.converter`

Converter class for offset value Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.

- Type: class
- Default:
- Importance: low

`rest.host.name`

Hostname for the REST API. If this is set, it will only bind to this interface.

- Type: string
- Importance: low

`rest.port`

Port for the REST API to listen on.

- Type: int
- Default: 8083
- Importance: low

`plugin.path`

The comma-separated list of paths to directories that containKafka Connect plugins.

- Type: string
- Default:
- Importance: low

Standalone Worker Configuration

In addition to the common worker configuration options, the following are available in standalone mode.

`offset.storage.file.filename`

The file to store connector offsets in. By storing offsets on disk, a standalone process can be stopped and started on a single node and resume where it previously left off.

- Type: string
- Default: ""
- Importance: high

Distributed Worker Configuration

Distributed workers that are configured with matching `group.id` values automatically discover each other and form a cluster. All workers in the cluster must also have access to and use the same three Kafka topics to share connector configurations, offset data, and status updates, and so all worker configurations must have matching `config.storage.topic`, `offset.storage.topic`, and `status.storage.topic` properties. As each distributed worker starts up, it will simply use these topics if they already exist; if not, it will attempt to create the topics using these and other topic-specific options described below. This allows you to manually create these topics before starting Kafka Connect if you require other advanced topic-specific settings or when Kafka Connect does not have the ACL privileges to create the topics If you do create the topics manually, be sure to follow the guidelines listed below.

In addition to the common worker configuration options, the following options may be specified in the properties file passed to the Connect worker when started in distributed mode as shown here:

`group.id`

A unique string that identifies the Connect cluster group this worker belongs to.

- Type: string
- Default: connect-cluster
- Importance: high

`config.storage.topic`

The name of the topic where connector and task configuration data are stored. This *must* be the same for all workers with the same `group.id`. Kafka Connect will upon startup attempt to automatically create this topic with a single-partition and compacted cleanup policy to avoid losing data, but it will simply use the topic if it already exists. If you choose to create this topic manually, **always** create it as a compacted topic with a single partition and a high replication factor (3x or more).

- Type: string
- Default: ""
- Importance: high

`config.storage.replication.factor`

The replication factor used when Kafka Connects creates the topic used to store connector and task configuration data. This should **always** be at least 3 for a production system, but cannot be larger than the number of Kafka brokers in the cluster.

- Type: short
- Default: 3
- Importance: low

`offset.storage.topic`

The name of the topic where connector and task configuration offsets are stored. This *must* be the same for all workers with the same `group.id`. Kafka Connect will upon startup attempt to automatically create this topic with multiple partitions and a compacted cleanup policy to avoid losing data, but it will simply use the topic if it already exists. If you choose to create this topic manually, **always** create it as a compacted, highly replicated (3x or more) topic with a large number of partitions (e.g., 25 or 50, just like Kafka's built-in `__consumer_offsets` topic) to support large Kafka Connect clusters.

- Type: string
- Default: ""
- Importance: high

`offset.storage.replication.factor`

The replication factor used when Connect creates the topic used to store connector offsets. This should **always** be at least 3 for a production

system, but cannot be larger than the number of Kafka brokers in the cluster.

- Type: short
- Default: 3
- Importance: low

`offset.storage.partitions`
The number of partitions used when Connect creates the topic used to store connector offsets. A large value (e.g., 25 or 50, just like Kafka's built-in `__consumer_offsets` topic) is necessary to support large Kafka Connect clusters.

- Type: int
- Default: 25
- Importance: low

`status.storage.topic`
The name of the topic where connector and task configuration status updates are stored. This *must* be the same for all workers with the same `group.id`. Kafka Connect will upon startup attempt to automatically create this topic with multiple partitions and a compacted cleanup policy to avoid losing data, but it will simply use the topic if it already exists. If you choose to create this topic manually, **always** create it as a compacted, highly replicated (3x or more) topic with multiple partitions.

- Type: string
- Default: ""
- Importance: high

`status.storage.replication.factor`
The replication factor used when Connect creates the topic used to store connector and task status updates. This should **always** be at least 3 for a production system, but cannot be larger than the number of Kafka brokers in the cluster.

- Type: short
- Default: 3
- Importance: low

`status.storage.partitions`
The number of partitions used when Connect creates the topic used to store connector and task status updates.

- Type: int
- Default: 5
- Importance: low

Security

For information about security, see Kafka Connect Security.

Configuring Converters

The `key.converter` and `value.converter` properties in the common worker configurations are where you specify a converter to use. The converters you can specify are listed below:

- `AvroConverter` (recommended): use with Confluent Schema Registry
- `JsonConverter`: great for structured data
- `StringConverter`: simple string format
- `ByteArrayConverter`: provides a "pass-through" option that does no conversion

Each converter has its own associated configuration requirements. To configure a converter-specific property, you prepend the connect property (where the converter has been specified) to the converter property.

The following worker property file snippet shows that the `AvroConverter` bundled with Schema Registry requires the URL for Schema Registry to

be passed as a property:

```
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
```

We recommend using the `AvroConverter` for your Kafka Connect data. Those with a need to use JSON for Connect data can use the `JsonConverter` supported with Kafka. An example of using the `JsonConverter` without schemas for converting keys looks like:

```
key.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=false
```

Important

These converters are used by *all* connectors running on the worker, except for any connectors whose configurations override these configurations.

Overriding Producer and Consumer Settings

Internally, Kafka Connect uses the standard Java producer and consumers to communicate with Kafka. Kafka Connect configures these producer and consumer instances with good defaults. Most importantly, it is configured with important settings that ensure data from sources will be delivered to Kafka in order and without any loss. The most critical site-specific options, such as the Kafka bootstrap servers, are already exposed via the standard worker configuration.

Occasionally, you may have an application that needs to adjust the default settings. One example is a standalone process that runs a log file connector. For the logs being collected, you might prefer low-latency, best-effort delivery and minor data loss in the case of connectivity issues might be acceptable for this application in order to avoid any data buffering on the client, keeping the log collection as lean as possible.

All new producer configs and new consumer configs can be overridden by prefixing them with `producer.` or `consumer.`, respectively. For example:

```
producer.retries=1
consumer.max.partition.fetch.bytes=10485760
```

would override the producers to only retry sending messages once and increase the default amount of data fetched from a partition per request to 10 MB.

Note that these configuration changes are applied to *all* connectors running on the worker. You should be especially careful making any changes to these settings when running distributed mode workers.

Upgrading Kafka Connect Workers

Documentation for upgrading your Kafka Connect workers is found in the platform upgrade section. To upgrade individual connectors, please see our documentation on upgrading connector plugins.

Adding Connectors (Docker)

Refer to Adding Connectors or Software for steps and examples for adding additional Kafka Connect connectors or customizing the connectors contained in a Docker image.

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

26 Votes

Last updated on Oct 17, 2019.