O'REILLY®

Kafka Connect

Build Data Pipelines by Integrating Existing Systems

Early Release
Raw & Unedited
Compliments of
Red Hat Developer

Mickael Maison
& Kate Stanley

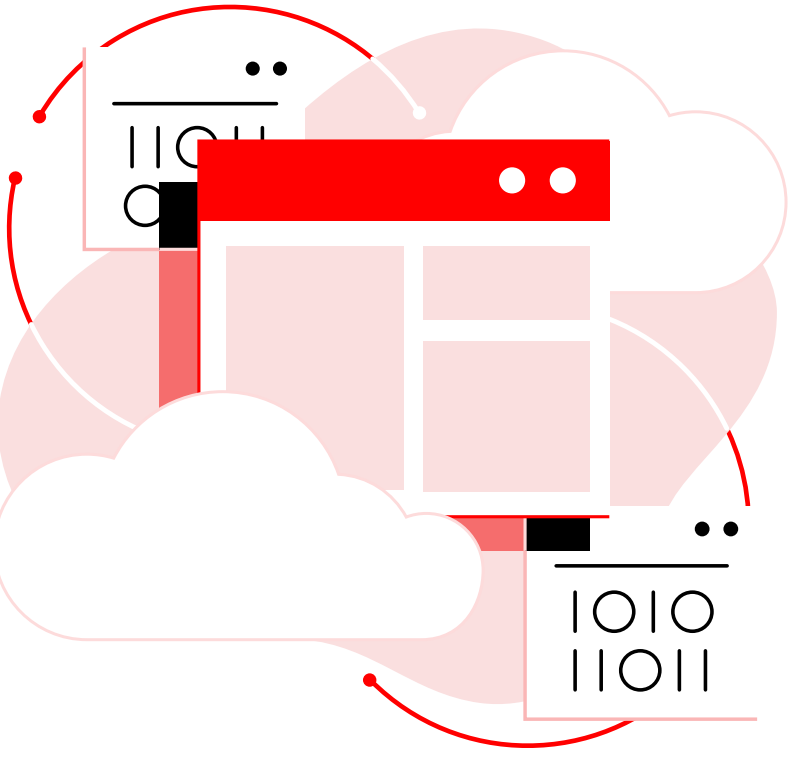
Connect data streams

Say goodbye to isolated data

Connect data in real-time across applications, APIs, devices, and edge computing using cloud-native technologies. Red Hat® OpenShift® Streams for Apache Kafka provides:

- Kubernetes-native application development, connectivity, and data streaming.
- A unified experience across different clouds.
- An ecosystem to streamline real-time data initiatives.

Start your data streaming journey



Red Hat

Kafka Connect

Build Data Pipelines by Integrating Existing Systems

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Mickael Maison and Kate Stanley

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kafka Connect
by Mickael Maison and Kate Stanley

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman
Development Editor: Jeff Bleiel
Production Editor: Gregory Hyman
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

October 2023: First Edition

Revision History for the Early Release

2022-02-18: First Release
2022-04-15: Second Release
2022-05-24: Third Release
2022-07-07: Fourth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098126537* for release details.

978-1-098-12653-7

Table of Contents

CHAPTER 1
Apache Kafka Basics

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the second chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *KafkaConnectBook@gmail.com*.

Connect is one of the components of the Apache Kafka project. While you don't need to be a Kafka expert to use Connect, it's useful to have a basic understanding of the main concepts in order to build reliable data pipelines.

In this chapter, we will give a quick overview of Kafka and you will learn the basics in order to fully understand the rest of this book. (If you already have a good understanding of Kafka, you can skip this chapter and go directly to Chapter 3.) We will explain what Kafka is, its use cases and briefly introduce some of its inner workings. Finally we will discuss the different Kafka clients, including Kafka Streams, and show you how to run them against a local Kafka cluster.

If you want a deeper dive into Apache Kafka, we recommend you take a look at the book "Kafka, the Definitive Guide".

A Distributed Event Streaming Platform

On the official website, Kafka is described as an "open-source distributed event streaming platform". While it's a technically accurate description, for most people it's not immediately clear what that means, what Kafka is and what you can use it for. Let's first look at the individual words of that description separately and explain what they mean.

Open Source

The project was originally created at LinkedIn where they needed a performant and flexible messaging system to process the very large amount of data generated by their users. It was released as an open source project in 2010 and it joined the Apache Foundation in 2011. This means all the code of Apache Kafka is publicly available and can be freely used and shared as long as the Apache License 2.0 is respected.

The Apache Foundation is a nonprofit corporation created in 1999 whose objective is to support open source projects. It provides infrastructure, tools, processes and legal support to projects to help them develop and succeed. It is the world's largest open source foundation and as of 2021, it supports over 300 projects totalling over 200 million lines of code.

The source code of Kafka is not only available, but the protocols used by clients and servers are also documented. This allows third parties to write their own compatible clients and tools. It's also noteworthy that the development of Kafka happens in the open. All discussions (new features, bugs, fixes, releases) happen on public mailing lists and any changes that may impact users have to be voted on by the community.

This also means Apache Kafka is not controlled by a single company that can change the terms of use, arbitrarily increase prices or simply disappear. Instead it is managed by an active group of diverse contributors. To date, Kafka has received contributions from over 800 different contributors. Out of this large group, a small subset (~50) are committers that can accept contributions and merge them into the Kafka codebase. Finally there's an even smaller group of people (25-30) called Project Management Committee (PMC) members that oversee the governance (they can elect new Committers and PMC members), set the technical direction of the project and ensure the community around the project stays healthy. You can find the current Committer and PMC member roster for Kafka on the website: https://kafka.apache.org/committers.

Distributed

Traditionally, enterprise software was deployed on few servers and each server was expensive and often used custom hardware. In the past 10 years, there has been

a shift towards using "off the shelf" servers (with common hardware) that are cheaper and easily replaceable. This trend is highly visible with the huge popularity of cloud infrastructure services that allow you to provision standardized servers within minutes whenever needed.

Kafka is designed to be deployed over multiple servers. A server running Kafka is called a *broker*, and interconnected brokers form a *cluster*. Kafka is a distributed system as the system workload is shared across all the available brokers. In addition, brokers can be added to or removed from the cluster dynamically to increase or decrease the capacity. This horizontal scalability enables Kafka to offer high throughput while providing very low latencies. Small clusters with a handful of brokers can easily handle several hundreds of megabytes per second and several Internet giants, such as LinkedIn and Microsoft, have large Kafka clusters handling several trillion events per day (LinkedIn: https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages; Microsoft: https://azure.microsoft.com/fr-fr/blog/processing-trillions-of-events-per-day-with-apache-kafka-on-azure/).

Finally distributed systems offer resilience to failures. Kafka is able to detect when brokers leave the cluster, due to an issue, or for scheduled maintenance. With appropriate configuration, Kafka is able to keep fully functional during these events by automatically distributing the workload on remaining brokers.

Event Streaming

An event stream is an unbounded sequence of events. In this context, an event captures that something has happened. For example, it could be a customer buying a car, a plane landing, or a sensor triggering.

In real life, events happen constantly and, in most industries, businesses are reacting to these events in real time to make decisions. So event streams are a great abstraction as new events are added to the stream as they happen. Event streaming systems provide mechanisms to process and distribute events in real time and store them durably so you can replay them later.

Kafka is not limited to handling "events" or "streams". Any arbitrary data, unbounded or finite, can be handled by Kafka and equally benefit from the processing and replay capabilities.

Platform

The final part of the definition is platform. Kafka is a platform because it provides all the building blocks to build event streaming systems.

As shown in Figure 1-1, the Apache Kafka project consists of the following components:

Cluster
: Brokers form a Kafka cluster and handle requests for Kafka clients.

Clients

Producer
: Sends data to Kafka brokers.

Consumer
: Receives data from Kafka brokers.

Admin
: Performs administrative tasks on Kafka resources.

Connect
: Enables building data pipelines between Kafka and external systems. This is the topic of this book!

Streams
: A library for processing data in Kafka.

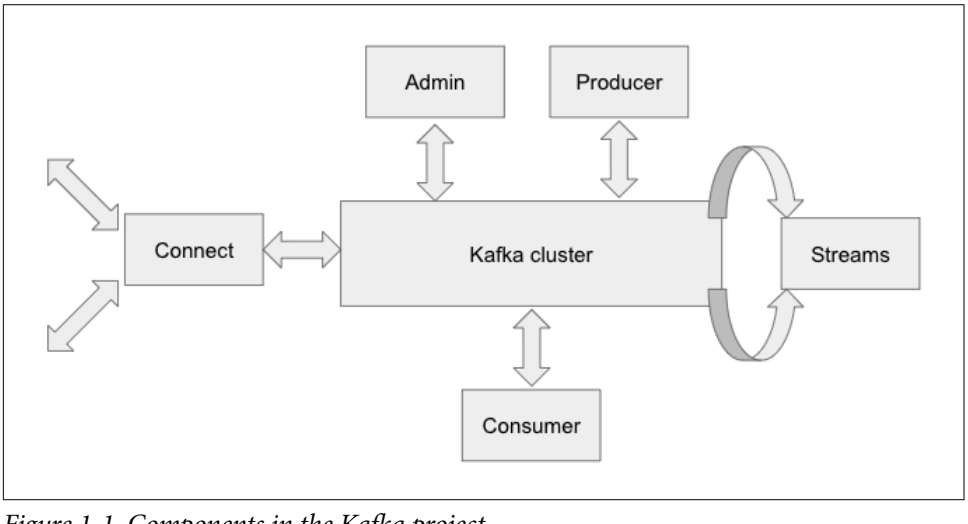


Figure 1-1. Components in the Kafka project

Due to its openness, many third party tools and integrations have been created by the ever growing Kafka community.

Putting it all together, we see that Kafka is an *open source project* with an open governance under the Apache foundation. Because it is *distributed*, it is scalable and able to handle very high throughput but also provides high availability. It provides low latency and unique characteristics make it ideal for handling *streams of events*.

Finally the various components of the project create a robust and flexible *platform* to build data systems.

Now that you understand what Kafka is, let's go over some of the use cases it excels at.

Use Cases

Here we will only explore the most common use cases. But integrations with a multitude of other data systems and flexible semantics make Kafka versatile in practice.

Log/Metrics Aggregation

This type of use case requires the ability to collect data from hundreds or thousands of applications in real time. In addition, strong ordering guarantees, especially for logs, and the capacity to handle sudden bursts in volume are key requirements. Kafka is a great fit for log and metrics aggregation as it's able to handle large volumes of data with very low latency.

Kafka can be configured as an appender by logging libraries like log4j2 to send logs directly from applications to Kafka instead of writing them to files on storage.

Stream Processing

Stream processing has emerged as a differentiating feature in many industries. It allows users to process and analyze data in real time and hence see results and make decisions as soon as possible. This is in contrast with batch processing where data is processed in large chunks for example, once each day.

Kafka is designed for handling streams of data and has tools and APIs specifically for this paradigm. Kafka Streams is a library for building stream processing applications. It provides high level APIs that hide the complexity of handling unbounded data streams, and it enables building complex processing applications that are reliable and scalable.

Messaging

Kafka is also great at generic messaging use cases. Because it is performant, offers strong delivery semantics, and allows decoupling the sending and receiving ends, it can fulfil the role of a message broker. Kafka clients are built into many application frameworks, making it a popular choice for connecting applications in event driven or microservices architectures.

How Kafka Works

Kafka uses the *Publish-Subscribe* messaging pattern to enable applications to send and receive streams of data. *Publish-Subscribe*, or *PubSub*, is a messaging pattern that decouples senders and receivers. The utility of this pattern is easy to understand with a simple example. Imagine you have two applications that need to exchange data. The obvious way to connect them is to have each application send data directly to the other one. This works for a small number of applications but as the number of applications increases the number of connections grows even more. This makes connecting applications directly impractical, even if most applications only need to talk to a few of the others as you can see in Figure 1-2.

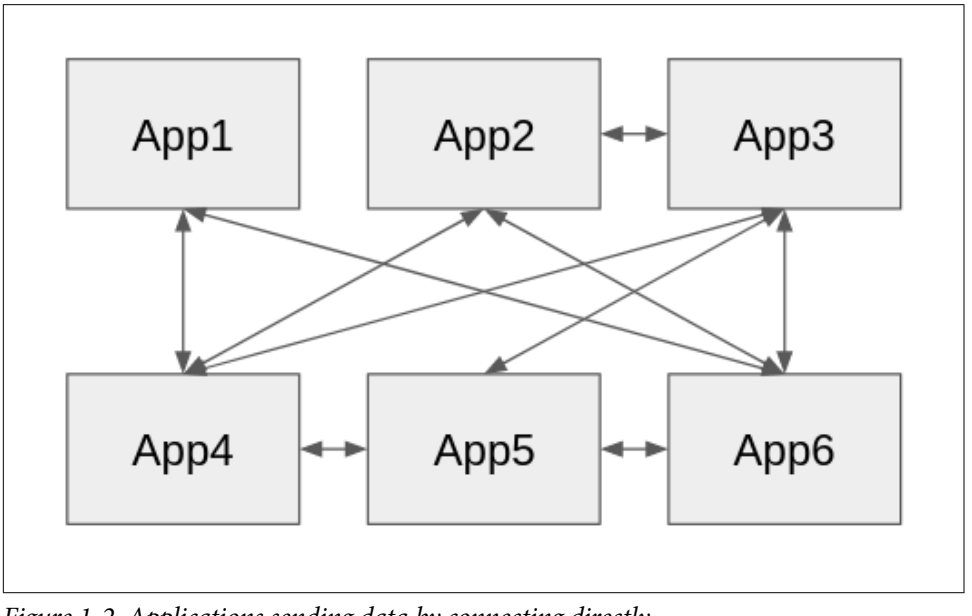


Figure 1-2. Applications sending data by connecting directly

Such tight coupling also makes it very hard to evolve applications and a single failing application can bring the whole system down if others depend on it. PubSub instead introduces the concept of a system that acts as the buffer between senders and receivers. As shown in Figure 1-3, Kafka provides this buffer.

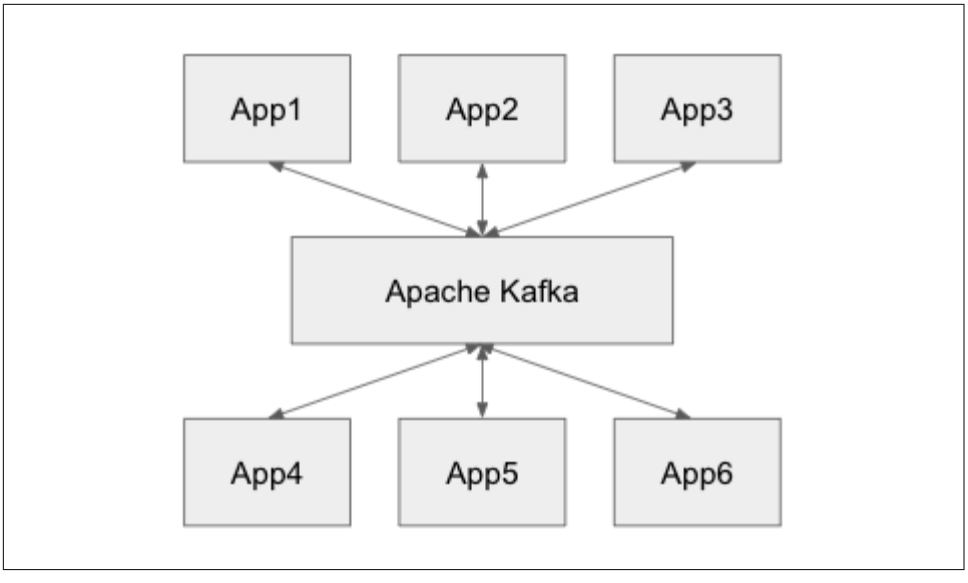


Figure 1-3. Apache Kafka provides a buffer between applications

The PubSub model makes it easy to add or remove applications without affecting any of the others. For example, it is easy to add more receivers if a piece of data is interesting for multiple applications. Similarly, if an application that is sending data is offline for any reason, the receiving applications aren't affected and will just wait for more data to arrive in the buffer. In Kafka, applications sending data are called *producers* and applications receiving data are called *consumers*.

Although there are many other technologies that use PubSub, very few provide a durable PubSub system. Unlike other messaging systems, Kafka does not remove data once it has been received by a particular consumer. Other systems have to make copies of the data so that multiple applications can read it, and the same application cannot read the same data twice. This is not the case with Kafka as applications can read a piece of data as many times as they like, and since it doesn't have to create new copies, adding new consuming applications has very little overhead.

In the rest of this section we will cover some common terms which will help you understand how Kafka provides this durable event streaming platform.

Brokers and Records

As discussed previously, a deployment of Apache Kafka, normally referred to as a Kafka cluster, is made up of one or more brokers. Each Kafka broker is deployed separately but they collaborate together to form the distributed event streaming platform that is at the core of Kafka. When a Kafka client wants to add data to or read data from the stream, they connect to one of these brokers.

The data in the event stream is stored on the Kafka brokers. However, to support the different use cases we discussed earlier, the brokers need to be able to handle lots of different types of data, no matter what format it is in. Kafka does this by only dealing with Kafka records. A *record* is made up of a *key*, a *value*, a timestamp and some *headers*. Records are sent to and read from Kafka as raw bytes. This means Kafka does not need to know the specific format of the underlying data, it just needs to know which parts represent the key, value, timestamp and headers.

Typically, the value is where you put the bulk of the data you want to send. The data in the value can be in any shape that is needed for the use case. For example, it could be a string, a JSON document or something more complex. The data can represent anything, from a message for a specific application, to a broadcast of a change in state. The record key is optional and is often used to logically group records and can inform routing. We will look at how keys affect routing later in this chapter.

The headers are also optional. They are a map of key/value pairs that can be used to send additional information alongside the data. The timestamp is always present on a record, but it can be set in two different ways. Either the application sending the record can set the timestamp, or the Kafka runtime can add the timestamp when it receives the record.

The record format is important as it brings on a few specificities that make Kafka extremely performant. First clients send records in the exact same binary format that brokers write to disk. Upon receiving records, brokers only need to perform some quick sanity checks such as Cyclic Redundancy Checks (CRC) before writing them to disk. This is particularly efficient as it avoids making copies or allocating extra memory for each record. Another technique used to maximize performance is batching, which consists in grouping records together. The Kafka record format is actually always a batch so when producers send multiple records grouped in a batch, this results in a smaller total size sent over the network and stored on brokers. Finally to reduce sizes further, batches can be compressed using a variety of data compression libraries, such as gzip, lz4, snappy or zstd.

Now let's look at where records are stored in Kafka.

Topics and Partitions

Kafka doesn't store one single stream of data, it stores multiple streams and each one is called a *topic*. When applications send records to Kafka they can decide which topic to send them to. To receive data, applications can choose one or more topics to consume records from. There is no right or wrong way to decide which records should go on which topic. It depends how you want those records to be used and your specific system. For example, suppose you are collecting temperature readings from sensors all over the world. You could put all those records into one big topic, or have a topic for each country. It depends how you want to use the topics later.

Kafka is designed to handle a high volume of data flowing through it at any one time. It uses partitions to help achieve this by spreading the workload across the different brokers. A *partition* is a subset of a particular topic. When you create the topic you decide how many partitions you want, with a minimum of one. Partitions are numbered starting from 0.

Figure 1-4 shows 2 topics spread across 3 brokers. The topic called `mytopic` has 3 partitions and the topic called `othertopic` has 2 partitions.

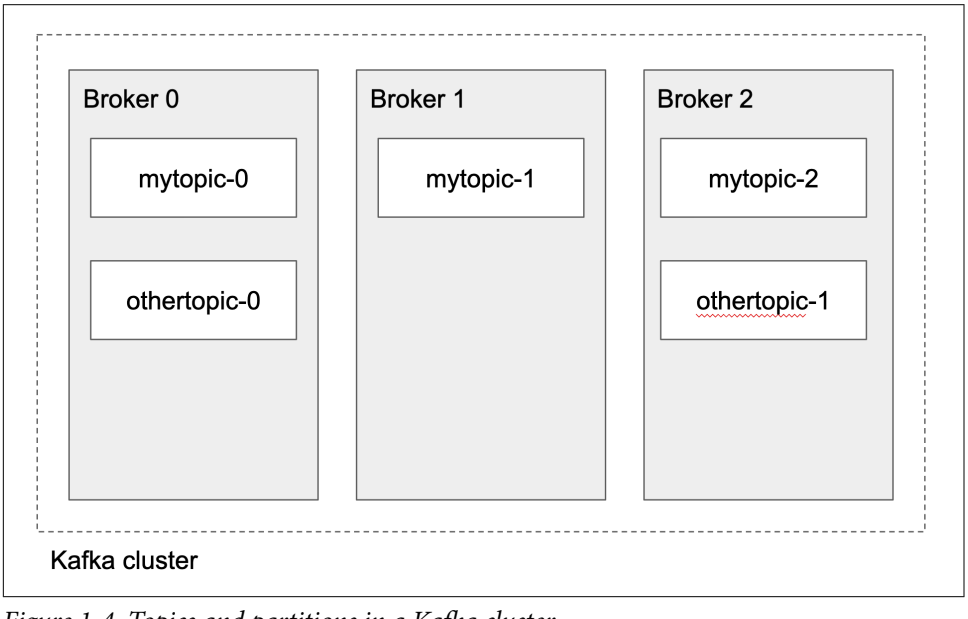


Figure 1-4. Topics and partitions in a Kafka cluster

If you only have one partition, every application that wants to send data to the topic has to connect to the same broker. This puts a lot of load on this single broker. Creating more partitions allows Kafka to spread a topic out across the brokers in the cluster. We will talk more about how the partitions affect both producing and consuming applications in the section on Kafka clients.

Figure 1-5 shows records on a partition. Each record in a partition can be identified by its *offset*. The first record added to the topic gets an offset of `0`, the second `1` etc. To uniquely identify a record in Kafka you need the topic, the partition and the offset.

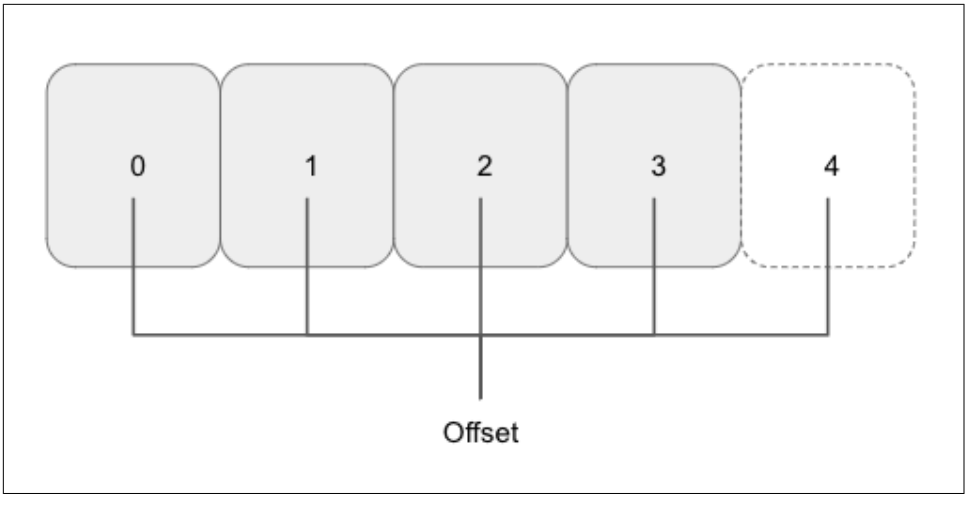


Figure 1-5. Records on a partition denoted by their offset

New records are always added to the end of the partition and Kafka records are ordered. This means Kafka will guarantee that the records are placed onto the partition in the order that it received them. This is a useful feature if you need to maintain the order of data in your stream. However, since the guarantee is only per partition, if you absolutely need all events in a topic to be strictly kept in the order they were received, you are forced to use a single partition in your topic.

Replication

In most systems it is sensible to assume that something will go down at some point. You should always be planning for failure. In Kafka, brokers can be taken offline for many reasons, whether it's the underlying infrastructure failing or the broker being restarted to pick up a configuration change. Because it's a distributed system, Kafka is designed for high availability and can cope with a broker going down. It does this using replication.

Replication means Kafka can maintain copies or *replicas* of partitions on a configurable number of brokers. This is useful because if one of the brokers goes down, the data in partitions on that broker isn't lost or unavailable. Applications can continue sending and receiving data to that partition, they just have to connect to a different broker.

Applications are told which broker to contact based on which one is the *leader* for the partition they are interested in. For each partition in a topic one broker is the leader and the brokers containing replicas are called *followers*. The leader is responsible for receiving records from producers and followers are responsible for keeping their copies of the partition up to date. Followers that have up to date copies

are called *in-sync replicas* (ISR). Consumers can connect to either the leader or ISR to receive records from the partition. If the leader goes offline for some reason Kafka will perform a leader election and choose one of the ISR as the new leader. Kafka applications are designed to handle leadership changes and will automatically reconnect to an available broker. This is what provides the high availability because applications can continue processing data even when Kafka brokers go down.

You configure the number of replicas for a topic by specifying the replication factor. If you have a replication factor of 3 then Kafka will make sure you have 1 leader broker and 2 follower brokers. If there are multiple partitions in the topic Kafka will aim to spread out the leaders amongst the brokers.

One broker within a Kafka cluster has the additional role of being the *controller*. This is the broker that is responsible for managing the leaders of each partition. If the controller goes down, Kafka will automatically select a new controller from the remaining eligible brokers.

Retention and Compaction

As mentioned earlier, Kafka topics contain unbounded streams of data, and this data is stored on the Kafka brokers. Since machine storage isn't unlimited this means at some point Kafka needs to delete some of the data.

When a topic is created you can tell Kafka the minimum amount of time to wait or the minimum amount of records to store, before it can start deleting records. There are various configuration options you can use to control deletion, such as `log.retention.ms` and `log.retention.bytes`.

Kafka won't delete your records as soon as it hits the specified time or size. This is due to the way Kafka stores the records on disk. The records in a partition are stored in multiple, ordered files. Each file is called a log segment. Kafka will only delete a record when it can delete the entire log segment file. Log segments are built up sequentially, adding one record at a time and you can control when Kafka starts writing to a new log segment using the `log.segment.ms` and `log.segment.bytes` settings.

Configuring clean up based on time or size doesn't always make sense. For example, if you are dealing with orders you might want to keep at least the last record for each order, no matter how old it is. Kafka enables such use cases by doing log compaction. When enabled on a topic, Kafka can remove a record from a partition when a new record has been produced that contains the same key. To enable compaction, set the `cleanup.policy` to `compact`, rather than the default `delete`.

In order to delete all records with a specific key you can send a *tombstone* record. That is a record with null value. When Kafka receives a tombstone record it knows it can clean up any previous records in that partition that had the same key. The tombstone record will stay on the partition for a configurable amount of time, but will also eventually be cleaned up.

Compaction keeps the overall partition size proportional to the number of keys used rather than to the total number of records. This also makes it possible for applications to rebuild their state at startup by reprocessing the full topic.

Similarly to record deletion, compaction doesn't happen immediately. Kafka will only compact records that are in the non-active segment files. That means if the segment file is currently still having records sent to it, it won't be compacted.

Kafka Clients

To get data into and out of Kafka you need a Kafka client. As we mentioned in section 2.1 the Kafka protocol is open, so there are plenty of clients to choose from or you can write your own. In this section we will introduce the Kafka clients that are shipped with the Kafka distribution. They are all Java clients and provide everything you need to get started with Kafka. The advantage of using one of these provided clients is they are updated when new versions of Kafka are released.

The configuration options we will cover are usually available within third-party clients as well. So even if you don't plan to use the included clients, the next few sections will help you understand how Kafka clients work. If you decide to use a third-party client, keep in mind that it can take some time for them to release a new version that supports the features in the latest Kafka.

Producers

Producers are applications that send records to topics. The class `org.apache.kafka.clients.producer.KafkaProducer<K,V>` is included as part of the Kafka distribution for producer applications to use. The `K` and `V` indicate the type of the key and value in the record.

When you are writing a Kafka producer application you don't have to specify the partition you want to send records to, you can just specify the topic. Then the Kafka client can determine which partition to add your record to using a *partitioner*. You can provide your own partitioner if you want to but to start off you can make use of one of the built-in partitioners.

As shown in Figure 1-6, the default partitioner decides which partition the record should go to based on the key of the record.

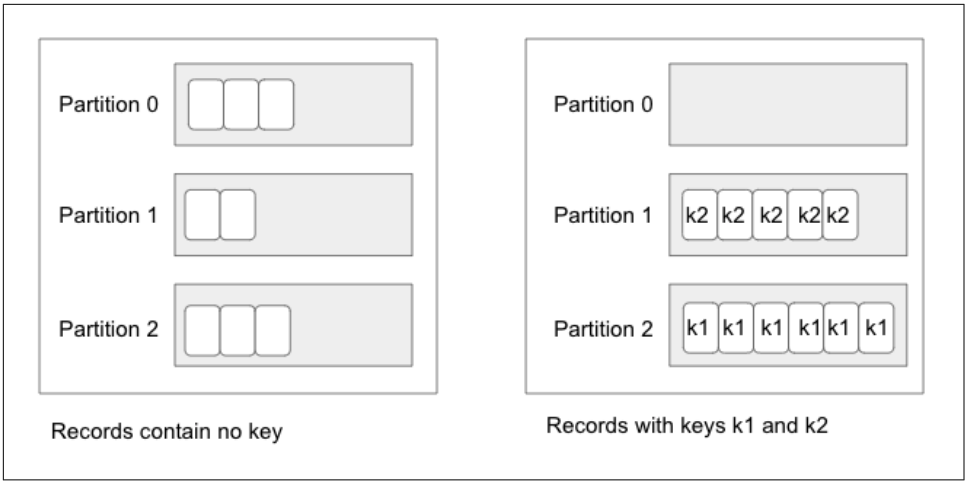


Figure 1-6. Records distributed on partitions by the default partitioner

For records without a key, the partitioner will roughly spread them out across the different partitions. Otherwise, the partitioner will send all the records with the same key to the same partition. This is useful if you care about ordering of specific records. For example if you are tracking status updates to a specific order, it makes sense to have these updates in the same partition so that you can take advantage of Kafka's ordering guarantees.

There are a few different configuration options that you should be aware of when writing a producer application. These are:

- `bootstrap.servers`
- `key.serializer`
- `value.serializer`
- `acks`
- `retries`
- `delivery.timeout.ms`
- `enable.idempotence`
- `transactional.id`

The `bootstrap.servers` configuration is a list of the broker endpoints the application will initially connect to. This can be a single broker, or include all the brokers. It is recommended to include more than one broker so that applications are still able to access the cluster even if a broker goes down.

The `key.serializer` and `value.serializer` configuration options specify how the client should convert records into bytes before sending them to Kafka. Kafka includes some serializers for you to use out of the box, for example `StringSerializer` and `ByteArraySerializer`.

You should configure `acks` and `retries` based on the message delivery guarantees you want for your specific use case. The `acks` configurations option controls whether the application should wait for confirmation from Kafka that a record has been received. There are three possible options: `0`, `1`, `all/-1`. If you set acks to `0` your producer application won't wait for confirmation that Kafka has received the record. Setting it to `1` means the producer will wait for the leader to acknowledge the record. The final option is `all`, or `-1` and this is the default. This acks setting means the producer won't get acknowledgement back from the leader broker until all followers that are currently in-sync (have an up-to-date copy of the partition) have replicated the record. This allows you to choose the delivery semantics you want, from maximum throughput (`0`) to maximum reliability (`all`).

If you have acks set to `1` or `all` you can control how many times the producer will retry on failure using the `retries` and `delivery.timeout.ms` settings. It is normally recommended to leave the `retries` setting as the default value and use the `delivery.timeout.ms` to set an upper bound for the time a producer takes trying to send a particular record. This determines how many times the producer will try to send the record if something goes wrong. Using the acks and retries settings, you can configure producers to provide at least once or at most once delivery semantics.

Kafka also supports exactly once semantics via the idempotent and transactional producers. You can enable the idempotent producer via the `enable.idempotence` setting and is enabled by default from Kafka 3.0 onwards. In this mode, a single producer instance sending records to a specific partition can guarantee they are delivered exactly once and in order. You enable the transactional producer using the `transactional.id` setting. In this mode, a producer can guarantee records are either all delivered exactly once to a set of partitions, or none of them are. While the idempotent producer does not require any code changes in your application to be used, in transactional mode, you need to explicitly start and end transactions in your application logic, via calls to begin and commit or abort.

If you don't use an idempotent or transactional producer and have retries enabled you might get reordering of records. This is because the producer could try to send multiple batches before the first batch is acknowledged. You can control the number of in-flight requests using `max.in.flight.requests.per.connection`.

Consumers

Consumer applications fetch records from Kafka topics. Similarly to producers there is a class included in the Kafka distribution that you can use: `org.apache.kafka.clients.consumer.KafkaConsumer<K,V>`. The `K` and `V` represent the type of key and value. Consumer applications can consume from a single topic, or multiple topics at the same time. They can request data from specific partitions, or use consumer groups to determine which partitions they receive data from.

Consumer groups are useful if you want to share the processing of records on a topic amongst a set of applications. All applications in a consumer group need to have the same `group.id` configuration. As shown in Figure 1-7, Kafka will automatically assign each partition within the topic to a particular consumer in the group.

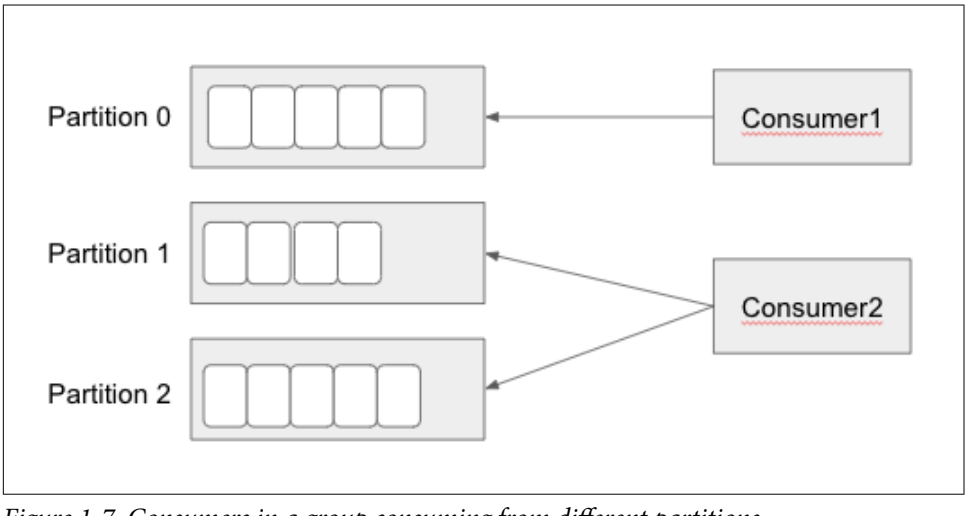


Figure 1-7. Consumers in a group consuming from different partitions

This means a single consumer could be assigned to multiple partitions, but for a single partition there is only one consumer processing it at a time. If a new consumer joins the group, or a consumer leaves the group, Kafka will rebalance the group. During rebalancing, the remaining consumers in the group coordinate with Kafka to assign the partitions amongst themselves.

If you want more control over the partition assignments you can assign them yourself in the application code. In Java applications you use the `assign` function to handle partition assignment manually or `subscribe` to let Kafka do it with a consumer group.

For each group, one broker within a Kafka cluster takes on the role of group coordinator. This broker is responsible for triggering rebalancing and coordinating the partition assignments with the consumers.

These are some of the configuration options you should be familiar with to write a consumer application:

- `bootstrap.servers`
- `group.id`
- `key.deserializer`
- `value.deserializer`
- `isolation.level`
- `enable.auto.commit`
- `auto.commit.interval.ms`

The `bootstrap.servers` configuration works the same as the matching configuration for producers. It is a list of one or more broker endpoints the application can initially connect to. The `group.id` determines which consumer group the application will join.

Configure the `key.deserializer` and `value.deserializer` to match how you want the application to deserialize records from raw bytes when it receives them from Kafka. This needs to be compatible with how the data was serialized in the first place by the producer application. For example if the producer sent a string you can't deserialize it as JSON. Kafka provides some default deserializers, for example `ByteArrayDeserializer` and `StringDeserializer`.

The setting `isolation.level` enables you to choose how consumers handle records sent by transactional producers. By default, this is set to `read_uncommitted` which means consumers will receive all records, including those from transactions that have not yet been committed by producers or have been aborted. Set it to `read_committed` if you want consumers to only see the records that are part of committed transactions. With this setting the consumers still receive all records that were not sent as part of a transaction.

Committing offsets

The `enable.auto.commit` and `auto.commit.interval.ms` configuration options are related to how consumer applications know which record to read next. Kafka persists records even after a consumer has read them and doesn't keep track of which consumers have read which records. This means it is up to the consumer to know which

record it wants to read next and where to pick up from if it gets restarted. Consumers do this using offsets.

We mentioned previously that a record can be uniquely identified by its topic, partition and offset. Consumers can either keep track of which offsets they have read themselves or use Kafka's built-in mechanism to help them keep track. It isn't recommended for consumer applications to keep track of offsets in memory. If the application is restarted for some reason it would lose its place and have to start reading the topic from the beginning again or risk missing records. Instead consumer applications should save their current position in the partition somewhere external to the application.

Applications can use Kafka to store their place in the partition by *committing offsets*. Most consumer clients provide a mechanism for a consumer to automatically commit offsets to Kafka. In the Java client this is the `enable.auto.commit` configuration option. When this is set to true the consumer client will automatically commit offsets based on the records it has read. It does this on a timer based on the `auto.commit.interval.ms` setting. Then if the application is restarted it will first fetch the latest committed offsets from Kafka and use them to pick up where it left off.

Alternatively you can write logic into your application to tell the client when to commit an offset. The advantage of this approach is you can wait until a record has finished being processed before committing offsets. Whichever approach you choose it's up to you to decide how to best configure it for your use case.

Streams

Kafka Streams is a Java library that gives you the building blocks to create complex stream processing applications. This means Streams applications process data on client-side, so you don't need to deploy a separate processing engine. Being a key component of the Kafka project, it takes full advantage of Kafka topics to store intermediate data during complex processes. We will give a brief overview of how it works here, but if you want to do a deeper dive the Kafka website goes into more detail.

Streams applications follow the read-process-write pattern. One or more Kafka topics are used as the stream input. As it receives the stream from Kafka, the application applies some processing and emits results in real time to output topics. The easiest way to explain the architecture of a Kafka Streams application is through an example. Consider a partition containing records that match those in Figure 1-8. The top word is the key and the bottom is the value, so the first record has a key of `choose_me` and a value of `Foo`.

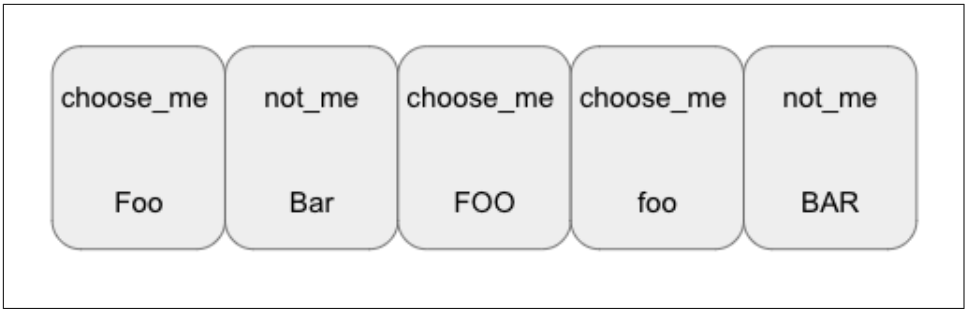


Figure 1-8. Records in a partition with a key and value

Imagine you want to only keep the records with a key of `choose_me` and you want to convert each of the values to lowercase. You do this by constructing a processor topology. The topology is made up of a set of stream processors or nodes that each connect together to create the topology. The stream processors perform operations on the stream. So for our example we would need a topology that looks similar to Figure 1-9.

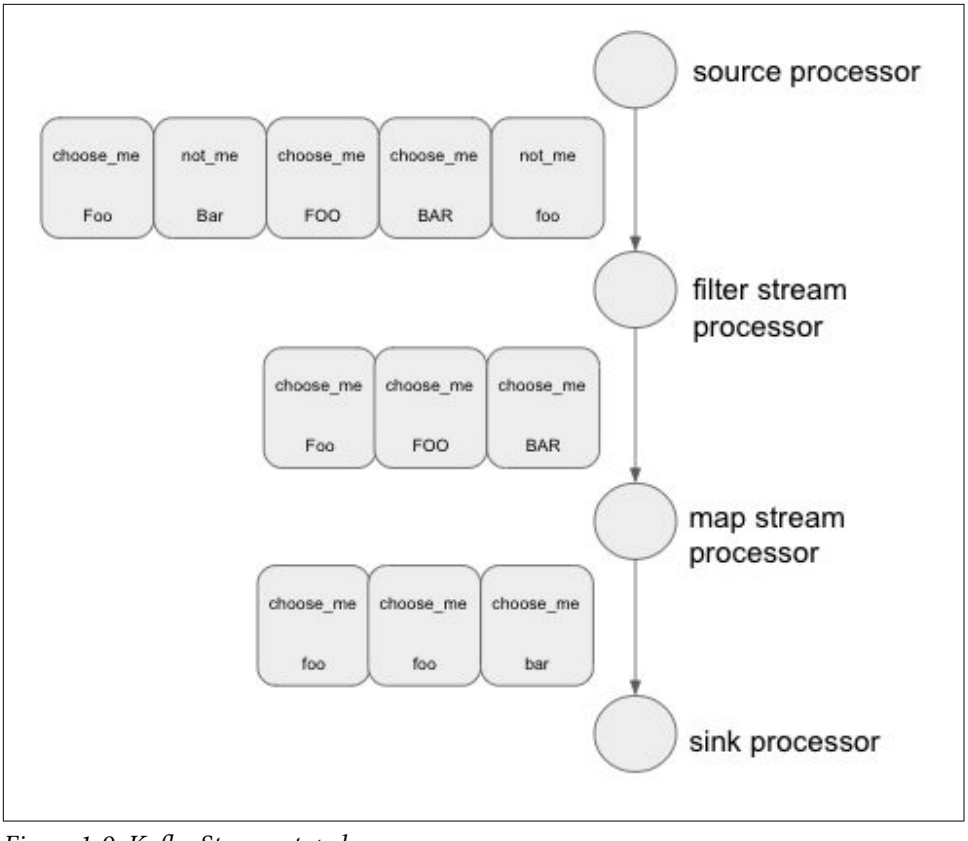


Figure 1-9. Kafka Streams topology

The first node reads from the input topic and is called a source processor. This passes the stream to the next node, which in our case is a filter that removes any record from the stream that doesn't have the key `choose_me`. The next node in the topology is a map that converts the record values to lowercase. The final node writes the resulting stream to an output topic and is called a sink processor.

To write our example in code you would need something similar to the following:

```
KStream<String, String> source = builder.stream("input-topic")
    .filter((key, value) -> key.equals("choose_me"))
    .map((key, value) -> KeyValue.pair(key, value.toLowerCase()))
    .to("output-topic")
```

This example only used stream processors that are part of the Kafka Streams DSL. The DSL provides stream processors that are useful for many use cases, such as map, filter and join. Using just the DSL you can build very complex processor topologies. Kafka Streams also provides a lower-level processor API that developers can use to extend Streams with stream processors that are specific to their use case. Kafka

Streams makes it easy to build processor topologies that contain many nodes and interact with many Kafka topics.

In addition to the basic stream processors used in the example, Kafka Streams also provides mechanisms to enable aggregation, or combining, of multiple records, windowing and storing state.

Getting Started with Kafka

Now that you understand the main concepts of Kafka, it's time to get it running. First you need to make sure you have Java installed in your environment. You can download it from https://java.com/en/download/.

Then, you need to download a Kafka distribution from the official Kafka website. We recommend you grab the latest version. Note that different versions of Kafka may require different Java versions. The supported Java versions are listed in https://kafka.apache.org/documentation/#java. Kafka releases are built for multiple versions of Scala, for example, Kafka 3.0.0 is built for Scala 2.12 and Scala 2.13. If you already use a specific Scala version, you should pick the matching Kafka binaries, otherwise it's recommended to pick the latest.

Once you've downloaded the distribution, extract the archive. For example, for Kafka 3.0.0:

```
$ tar zxvf kafka_2.13-3.0.0.tgz
$ cd kafka_2.13-3.0.0
```

Kafka distributions have scripts for Unix based systems under the *bin* folder and Windows systems under *bin/windows*. We will use the commands for Unix based systems, but if you are using Windows, replace the script names with the Windows version. For example, *./bin/kafka-topics.sh* would be *.\bin\windows\kafka-topics.bat* on Windows.

Starting Kafka

As described previously, Kafka initially required ZooKeeper in order to run. The Kafka community is currently in the process of removing this dependency. We'll cover both ways to start Kafka, you can follow one or the other.

Kafka in KRaft mode (without ZooKeeper)

In this mode, you can get a Kafka cluster running by starting a single Kafka broker.

You first need to generate a cluster ID:

```
$ ./bin/kafka-storage.sh random-uuid
RAtwS8XJRYywwDNBQNB-kg
```

Then you need to format the Kafka storage directory. By default, the directory is */tmp/kraft-combined-logs* and can be changed to a different value by changing `log.dirs` in *./config/kraft/server.properties*. To format the directory, run the following command, replacing `<CLUSTER_ID>` with the value returned by the previous command:

```
$ ./bin/kafka-storage.sh format -t <CLUSTER_ID> -c ./config/kraft/server.proper-
ties
Formatting /tmp/kraft-combined-logs
```

Finally you can start a Kafka broker:

```
$ ./bin/kafka-server-start.sh ./config/kraft/server.properties
```

Look out for the `Kafka Server started (kafka.server.KafkaRaftServer)` line to confirm the broker is running.

Kafka with ZooKeeper

If you don't want to run in KRaft mode, before starting Kafka, you need to start ZooKeeper. Fortunately, the ZooKeeper binaries are included in the Kafka distribution so you don't need to download or install anything else. To start ZooKeeper you run:

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

To confirm ZooKeeper is successfully started, look for the following line in the logs:

```
binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxn-
Factory)
```

Then in another window, you can start Kafka:

```
$ bin/kafka-server-start.sh config/server.properties
```

You should see the following output in the Kafka logs:

```
[KafkaServer id=0] started (kafka.server.KafkaServer)
```

Sending and receiving records

Before exchanging records, you first need to create a topic. To do so, you can use the kafka-topics tool:

```
$ ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-
first-topic --partitions 1 --replication-factor 1
Created topic my-first-topic.
```

The `--partitions` flag indicates how many partitions the topic will have. The `--replication-factor` flag indicates how many replicas will be created for each partition.

Let's send a few records to your topic using the kafka-console-producer tool:

```
$ ./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-
first-topic
> my first record
> another record
```

When you are done, you can stop the producer by pressing CTRL + C.

You can now use the kafka-console-consumer tool to receive the records in the topic

```
$ ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-
first-topic --from-beginning
my first record
another record
```

Note that we added the `--from-beginning` flag to receive all existing records in the topic. Otherwise, by default, the consumer only receives new records.

Running a Streams app

To conclude this quick overview, you can also run a small Kafka Streams application which is included in the Kafka distribution. This application consumes records from a topic and counts how many times each word appears. For each word, it produces the current count into a topic called `streams-wordcount-output`.

In order to run the application, you need to create the topic that it will use as its input:

```
$ ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic
streams-plaintext-input --partitions 1 --replication-factor 1
```

Once you've created the topic, start the Streams application:

```
$ ./bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.Word-
CountDemo
```

Here you need to leave this running and open a new window to run the remaining commands.

In a new window, you can produce a few records to the input topic:

```
$ ./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-
first-topic
> Running Kafka
> Learning about Kafka Connect
```

Again press CTRL + C to stop the producer once you're done.

Finally you can see the output of the application by consuming the records on the `streams-wordcount-output` topic:

```
$ ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic streams-wordcount-output \
  --from-beginning \
  --formatter kafka.tools.DefaultMessageFormatter \
```

```
  --property print.key=true \
  --property print.value=true \
  --property key.deserializer=org.apache.kafka.common.serialization.StringDeser-
ializer \
  --property value.deserializer=org.apache.kafka.common.serialization.LongDeser-
ializer
Running  1
Kafka    1
Learning 1
about    1
Kafka    2
Connect  1
```

For each word, the Streams application has emitted a record that has the word as the key and the current count as the value. For this reason, we configured the kafka-console-consumer command to have the appropriate deserializers for the key and value.

Summary

In this chapter we covered some of the basics of Apache Kafka. We introduced the open source project and explained some common use cases. After reading this chapter you should understand the following key terms that you will likely encounter elsewhere in this book:

- Broker
- Record
- Partition
- Topic
- Offset

We also looked at the different clients that interact with Kafka, from producers and consumers that write to and read from partitions, to Kafka Streams that process streams of data.

Finally we walked through how to start Kafka, create a topic, send and consume a record and run a Kafka Streams application. You can refer back to the steps in the getting started section as you progress through this book. You will need Kafka running before you start Kafka Connect and the producer and consumer sections can be used to either send test data for Kafka Connect to read, or check the data that Kafka Connect has put into Kafka.

CHAPTER 2

Components in a Connect Data Pipeline

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the third chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *KafkaConnectBook@gmail.com*.

A Kafka Connect pipeline involves multiple components, such as the runtime, connectors, converters and transformations. You can combine and configure these pluggable components in many different ways to get the best pipeline for your use case. To get the most out of Connect it's important to understand the purpose of each component and how to configure them.

In this chapter we will cover each of the core Kafka Connect components: the runtime, connectors, converters and transformations. We will introduce some key concepts you should understand, give a high-level overview of how each component works and how to use them together. People often use the term Connect to refer to one component, or the whole pipeline, so we will introduce the correct terms for each component so you can differentiate them. By the end of this chapter you will know how to build, configure and run a basic Kafka Connect pipeline using the standard Kafka distribution.

Kafka Connect Runtime

At its core Kafka Connect is a runtime that runs and manages the components that make up the pipeline. You can either deploy the runtime in "standalone" or "distributed" mode. In standalone mode you run a single Kafka Connect process and it stores its state on the filesystem as shown in Figure 2-1.

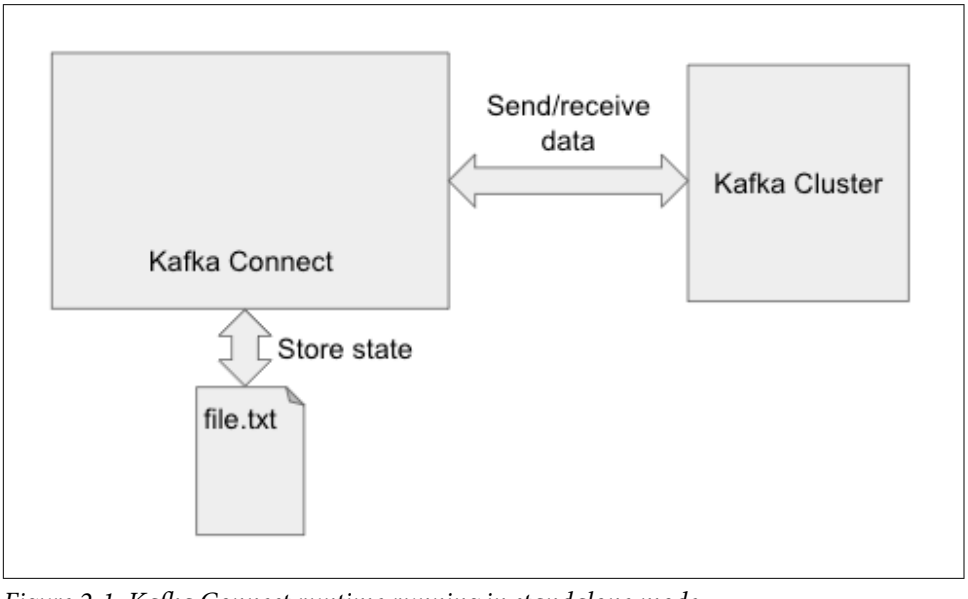


Figure 2-1. Kafka Connect runtime running in standalone mode

In this mode data only flows through the Connect pipeline as long as this single process is up, and you can't make any changes to the pipeline once it is started.

For production deployments it is preferable to instead run in the distributed mode. As shown in Figure 2-2, in this configuration you start one or more Kafka Connect runtimes. Each one runs independently and we refer to them as "workers". The workers collaborate with each other to spread out workload and store joint state in Kafka topics.

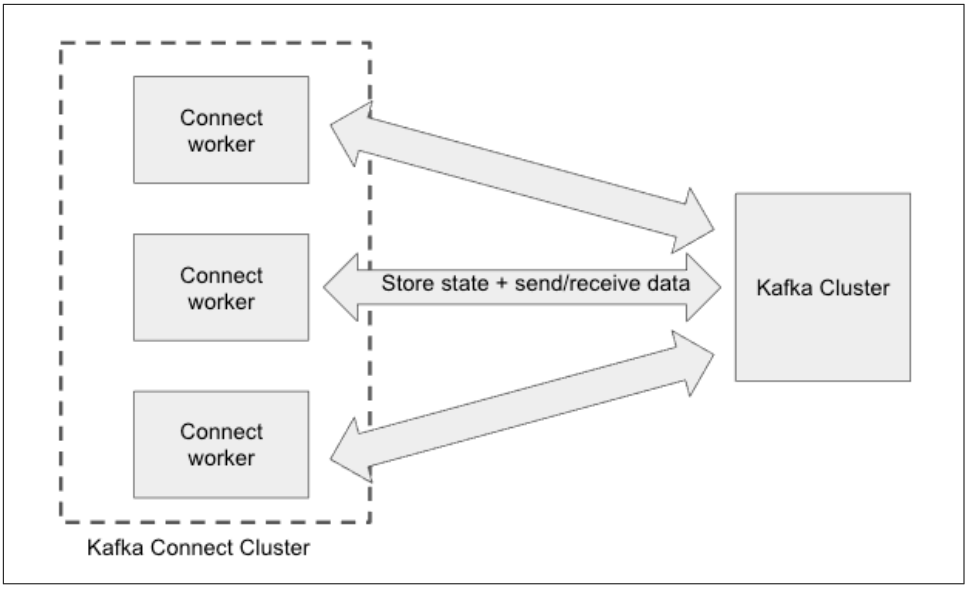


Figure 2-2. Kafka Connect runtime running in distributed mode

In distributed mode Kafka Connect distributes the workload amongst the workers and the pipeline can be reconfigured while Connect is running. Having multiple workers means Kafka Connect can continue flowing data even if a worker goes down and you can add more workers to the cluster if needed. This means the system is resilient and scalable.

A Kafka Connect cluster, both in standalone and distributed mode, flows data between a single Kafka cluster and one or more external systems. However, for a single Kafka cluster, there is no limit to the number of Kafka Connect clusters that you can have connected to it.

If you want to run more than one Kafka Connect cluster in the same environment make sure you consider where the state will be stored. If running Connect in standalone mode, make sure each cluster has its own file. If running in distributed mode, change the topics that are used so the different clusters don't interfere with each other.

We will now discuss the four steps you need to take to get a Kafka Connect runtime up and running. These are, getting the binaries and scripts, starting the runtime, customising the runtime with plugins and managing the runtime once it is started using the REST API.

Binaries and Scripts

You can easily run Kafka Connect on a laptop using the scripts, jar files and configuration files provided in the Kafka distribution. For example Kafka 3.0.0 includes the following scripts in the *bin* directory for Unix based operating systems:

- *connect-distributed.sh*
- *connect-standalone.sh*

The equivalent scripts for Windows operating systems are under *bin/windows* in the Kafka distribution:

- *connect-distributed.bat*
- *connect-standalone.bat*

These two scripts start the distributed and standalone versions of Kafka Connect respectively.

The libs directory contains the following jar files:

connect-api-3.0.0.jar
: Api jar for writing a new connector

connect-basic-auth-extension-3.0.0.jar
: Library to allow you to add basic authentication to the Kafka Connect REST API

connect-file-3.0.0.jar
: File connectors for writing to and reading from a file

connect-json-3.0.0.jar
: Converter for writing data into Kafka using a json format

connect-runtime-3.0.0.jar
: The Kafka Connect runtime

connect-transforms-3.0.0.jar
: API for writing transformations

This directory is automatically added to the Connect classpath and together these jar files include everything you will need for both writing custom components and running them on Kafka Connect.

Finally the *config* directory contains the following properties files:

- *connect-console-sink.properties*
- *connect-console-source.properties*
- *connect-file-sink.properties*

- *connect-file-source.properties*
- *connect-log4j.properties*
- *connect-standalone.properties*
- *connect-distributed.properties*

The files *connect-console-sink.properties*, *connect-console-source.properties*, *connect-file-sink.properties* and *connect-file-source.properties* all contain example configurations you can use to deploy the file connectors that can be used to build a data pipeline that includes writing to and reading from a file.

The *connect-log4j.properties* file is an example of how to adjust the logging levels of Kafka Connect.

You can use the *connect-standalone.properties* and *connect-distributed.properties* to start Kafka Connect in standalone and distributed mode respectively.

Running Kafka Connect in Distributed Mode

Before starting up Kafka Connect make sure you have a Kafka cluster running. We will use the *connect-distributed.sh* script from the bin directory to start Kafka Connect in distributed mode. The script requires a configuration file, so we will use the *connect-distributed.properties* file from the config directory.

The configuration file must provide the following values:

`bootstrap.servers`
: A comma-separated list of addresses for Kafka Connect to use for Kafka.

`group.id`
: A unique name for the Kafka Connect cluster which is used for workers to identify the others in their cluster.

`key.converter` *and* `value.converter`
: The format of keys and values of events in Kafka. Kafka connect uses these as defaults to convert between that format and the Kafka Connect internal format.

`offset.storage.topic`
: The topic Kafka Connect will use to store offsets.

`config.storage.topic`
: The topic Kafka Connect will use to store connector configuration.

`status.store.topic`
: The topic Kafka Connect will use to store its status.

You can override the converter settings with individual connector configuration. For example you might set JSON as the default converter but run a connector that should produce using the String format.

In addition the file can include overrides to the default configuration, such as the replication factor for the Kafka Connect topics, the host and port used for the REST API and the location of connector jar files. The full list of configuration options is available in the Apache Kafka documentation.

We are using a configuration file that assumes there is a Kafka broker accessible on *localhost:9092* and that there is only a single broker. If your environment is different then you need to edit the connect-distributed.properties file before you use it.

To start Kafka Connect:

1. Navigate to the directory containing the Kafka distribution.
2. Start ZooKeeper (if applicable).
3. Start Kafka.
4. Start Kafka Connect by running `./bin/connect-distributed.sh config/connect-distributed.properties`.

Your terminal will tail the Kafka Connect logs. Take a look at the logs and note Kafka Connect prints out the configuration it's using. You can also make sure there are no errors. If you list the topics in your Kafka cluster you can see the new topics created by Kafka Connect:

```
$ ./bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

You will have one topic for each of configs, offsets and status:

```
connect-configs
connect-offsets
connect-status
```

Plugins

The Kafka Connect runtime is the starting point for all Connect pipelines, then you add additional components specific to your use case. These additional components are called "plugins". Connectors, converters and transformations are all types of plugins you can load into Kafka Connect. Some plugins are included in the Kafka distribution and are already available on the Kafka Connect classpath. There are two ways to add new plugins, either by adding them to the Kafka Connect classpath, or by using the `plugin.path` configuration option. If possible, we recommend that you

use the `plugin.path` so that Kafka Connect only loads the libraries for the required plugin. This prevents classpath clashes between plugins.

Your `plugin.path` should be configured to point to a list of one or more directories. Each directory can contain a combination of JAR files and directories that in turn contain the assets (JAR files or class files) for a single plugin. For example the contents of a directory listed in the `plugin.path` could look like:

```
+-- custom-plugin-1.jar ❶
+-- custom-plugin-2 ❷
|     +-- custom-plugin-2-lib1.jar
|     +-- custom-plugin-2-lib2.jar
```

❶ A single jar file containing the plugin and all its dependencies

❷ A directory containing a set of jar files that include the jar file for the plugin and the jar files for all its dependencies

Whether you use the `plugin.path` approach or the classpath approach, when Kafka Connect starts up it lists out the plugins it has loaded:

```
INFO Added plugin 'org.apache.kafka.connect.converters.ByteArrayConverter'
INFO Added plugin 'org.apache.kafka.connect.file.FileStreamSourceConnector'
INFO Added plugin 'org.apache.kafka.connect.transforms.TimestampRouter'
```

We will cover the different plugin types (connectors, converters, transformations) in detail in the other sections of this chapter.

Kafka Connect REST API

When running distributed mode, Kafka Connect includes a REST API to allow you to query the cluster for the current state. By default this endpoint is not secured and uses the HTTP protocol, but Kafka Connect can be configured to use HTTPS instead. You can also configure the port that Kafka Connect is listening on. The default value is 8083.

With Kafka Connect up and running try the following curl command:

```
$ curl localhost:8083
```

It gives you basic information about the cluster:

```
{"version":"3.0.0","commit":"8cb0a5e9d3441962","kafka_cluster_id":"SXu4poDjQ-
ZyzQ84eB4Asjg"}
```

You can use jq to print the responses from the REST API in a more readable format:

```
$ curl localhost:8083 | jq
{
  "version":"3.0.0",
  "commit":"8cb0a5e9d3441962",
  "kafka_cluster_id":"SXu4poDjQZyzQ84eB4Asjg"
}
```

The REST API supports two different base paths: */connectors* and */connector-plugins*. For more details on how to use the REST API to manage connectors see Chapter 8.

You can use the REST API with the */connector-plugins* path to verify which connector plugins are currently installed into your Kafka Connect cluster:

```
$ curl localhost:8083/connector-plugins
```

By default, you will have available the `FileStreamSink`, `FileStreamSource` and the MirrorMaker2 connectors:

```
[
  {
    "class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type":"sink",
    "version":"3.0.0"
  },
  {
    "class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
    "type":"source",
    "version":"3.0.0"
  },
  {
    "class":"org.apache.kafka.connect.mirror.MirrorCheckpointConnector",
    "type":"source",
    "version":"1"
  },
  {
    "class":"org.apache.kafka.connect.mirror.MirrorHeartbeatConnector",
    "type":"source",
    "version":"1"
  },
  {
    "class":"org.apache.kafka.connect.mirror.MirrorSourceConnector",
    "type":"source",
    "version":"1"
  }
]
```

Once you have started Kafka Connect and verified that it's running, you can start a connector plugin.

Source and Sink Connectors

Connectors are plugins you can add to a Kafka Connect runtime. They serve as the interface between external systems and the Connect runtime and encapsulate all the external system specific logic. A connector consists of one or more JAR files that implement the Connect API.

As shown in Figure 2-3, there are two types of connectors:

Sink connectors
: To export records from Kafka to external systems

Source connectors
: To import records from external systems into Kafka

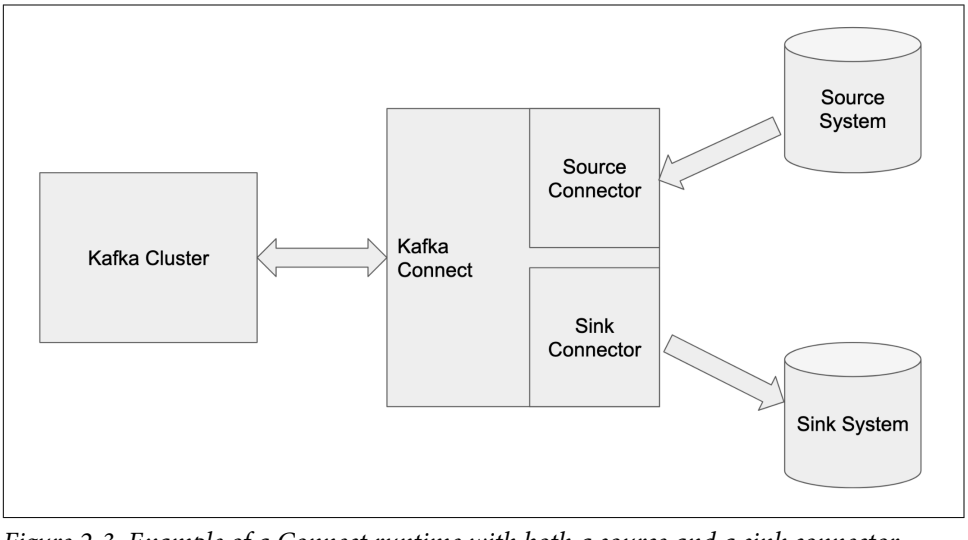


Figure 2-3. Example of a Connect runtime with both a source and a sink connector

A connector typically targets a single system or protocol. For example, you can have a S3 Sink connector that is able to write records into S3, or a JDBC Source connector that is able to retrieve records from various databases via JDBC. Often connectors come in pairs, a sink and a source, for the same system but this is not required. If you only want to sink records into S3, you only need the S3 sink connector.

How Do Connectors Work?

The Connect runtime runs and manages connectors by calling the various methods on the connector API. It exchanges data with connectors using `ConnectRecord` objects. `ConnectRecord` is an abstract Java class that encapsulates records flowing through Connect. It has fields for the record's topic, partition, key and its schema,

value and its schema, timestamp and headers. There are two concrete record classes that are specific to each connector type:

`SinkRecord`
: Sink connectors receive `SinkRecord` objects from the runtime. In this case, all fields are populated with the Kafka details from this record.

`SourceRecord`
: Source connectors build `SourceRecord` objects to pass to the runtime. They can use the fields to provide some information about the record in the source system.

Individual connectors are responsible for translating data between these `ConnectRecord` formats and the format used by the external system. It allows the runtime to stay generic and not know any details of the connector's external system.

The role of a connector is to create and manage tasks. Tasks do the actual work of exchanging data with external systems.

When a connector starts up, it computes how many tasks to start. For each task, the connector also computes a configuration and assigns the task a share of the workload. Different connectors compute the number of tasks and assignments in different ways. This can depend on the connector internal logic, the user provided connector configuration as well as the state of the external system it's interacting with.

In Connect, multiple tasks can run in parallel and they can also be spread across multiple workers when running in distributed mode. This works very much like regular Kafka consumer groups that distribute partitions across consumers. The work is split across tasks and it can be dynamically rebalanced when resources change. At runtime, connectors can detect if the resources they interact with have changed and trigger a reconfiguration. This allows adjusting the number of tasks to match the current workload. For example, a sink connector can create new tasks, up to the configured maximum, if you have added partitions to the topics it's exporting data from. This makes tasks the unit of scalability in Connect.

In order to start a connector, you first need to define its configuration. Some configuration options apply to all connectors such as the maximum number of tasks, `tasks.max`. Others are specific to source or sink connectors, for example, the configuration `topics` apply to all sink connectors and it indicates which topics the runtime will consume data from. Finally each connector has its own specific configuration options that depend on its implementation, features and on the system it's targeting. This means you can start multiple copies of the same connector with different configurations that run independently in parallel. For example, you can start two instances of the S3 sink connector to export different topics into different S3 buckets.

In a connector configuration, it is also possible to override some runtime configurations. This is useful to adjust Kafka client configurations and change key, value or header converters.

Adjust Kafka client configurations

The runtime creates dedicated Kafka clients for each connector. For example, for a sink connector, it creates consumers retrieving data from the specified topics. By default, these clients use the runtime configuration. Some connectors may benefit from some client specific configuration changes such as different fetch sizes, isolation level or timeouts.

Key, value, and header converters

The runtime is configured with default converters to control the format of records sent to/from Kafka by Connect. Individual connectors can override this setting to provide the converters required for their use case. We'll cover converters in more detail in the next section.

Finding Connectors for Your Use Case

Out of the box, Apache Kafka only provides a handful of ready-made source and sink connectors. There are two file connectors, a sink and a source, for interacting with files on disks. These ready-made connectors mostly serve as examples to demonstrate how both source and sink connectors work and for quick demos. The other included connectors are for connecting Kafka to … Kafka! These are the MirrorMaker source connectors used for mirroring Kafka clusters, we'll cover those in detail in Chapter 6.

Fortunately, the Kafka community has implemented connectors for hundreds of popular technologies ranging from messaging systems, to databases, to storage and data warehouse systems. To find connectors for your use case, you can use repositories like Confluent Hub or the Event Streams connector catalog that reference the most used and tested connectors. Finally, in case there is not already a connector available, as Connect is a pluggable API, you can implement your own connectors! We will cover how to do so in Chapter 12.

How Do You Run Connectors?

In the previous section, you started a Connect runtime in distributed mode, let's now use it to start a connector. For example, we can start the file sink connector, `FileStreamSinkConnector`, to write records from a Kafka topic into a file. To start a connector, you use the Connect REST API.

First you create a file named *sink-config.json* that contains the desired configuration for the connector:

```
{
  "name": "file-sink", ❶
  "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector", ❷
  "tasks.max": 1, ❸
  "topics": "topic-to-export", ❹
  "file": "/tmp/sink.out", ❺
  "value.converter": "org.apache.kafka.connect.storage.StringConverter" ❻
}
```

❶ `name` specifies the name we're giving to this connector instance. When managing connectors or looking at logs, we will use this name.

❷ `connector.class` is the fully qualified Java class name of the connector we want to run. You can also provide the short name, `FileStreamSink` in this case.

❸ `tasks.max` defines the maximum number of tasks that can be run for this connector.

❹ `topics` specifies which topics this connector will receive records from.

❺ `file` indicates where the connector will write Kafka records, you can change it to your preferred path. This configuration is specific to this connector.

❻ `value.converter` overrides the runtime's `value.converter` configuration. We use the `StringConverter` here so we can produce raw text via the console producer.

Create a topic called `topic-to-export` using:

```
$ bin/kafka-topics.sh --bootstrap-server localhost:9092 \
  --create --replication-factor 1 --partitions 1 --topic topic-to-export
```

Then you use the Connect REST API to start the connector with the configuration you created:

```
$ curl -X PUT -H "Content-Type: application/json" \       http://local-
host:8083/connectors/file-sink/config --data "@sink-config.json"
```

Once it has started, you can check the state your connector via the Connect REST API:

```
$ curl http://localhost:8083/connectors/file-sink | jq
 {
  "name": "file-sink",
  "config": {
        "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnec-
tor",
        "file": "/tmp/sink.out",
        "tasks.max": "1",
```

```
            "topics": "topic-to-export",
            "name": "file-sink",
            "value.converter": "org.apache.kafka.connect.storage.StringConverter"
        },
        "tasks": [
            {
                "connector": "file-sink",
                "task": 0
            }
        ],
        "type": "sink"
    }
```

You can see a `FileStreamSinkConnector` instance is running and it has created one task.

Let's now insert some records into the `topic-to-export` topic. To do so, you can use the console producer:

```
$ bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic topic-to-export
> First record
> Another record
> Third record
```

The connector appends the records you produced to the file */tmp/sink.out*:

```
$ cat /tmp/sink.out
First record
Another record
Third record
```

Connectors do the actual work of moving data between Kafka and external systems. However, the connectors need to know what format to use when sending the data to and from Kafka. That's where the next plugin type, converters, comes in.

Converters

In this section we will discuss the mechanism that is used to translate between the Kafka Connect internal format and the format used by Kafka. That mechanism is called a converter. When data is sent from Kafka Connect and Kafka it is serialised and sent as a stream of bytes. When data is sent from Kafka to Kafka Connect it has to be deserialized. A converter is a library that implements the Converter API and describes how to convert a `ConnectRecord` into bytes and back again.

For source connectors, as shown in Figure 2-4, converters are invoked after the connector.

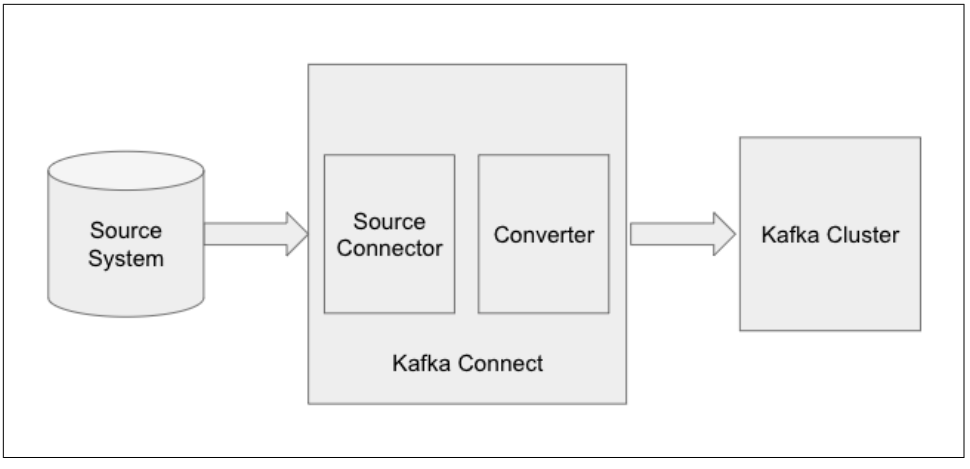


Figure 2-4. Source Connect pipeline

On the other hand, for sink connectors, converters are invoked before the connector, as shown in Figure 2-5.

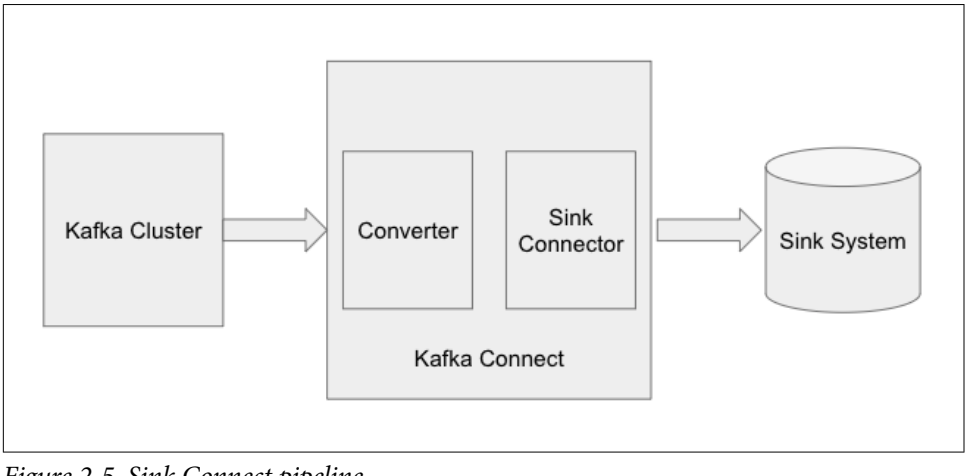


Figure 2-5. Sink Connect pipeline

Why Is Data Format Important?

To understand why converters are important we need to consider the data that is flowing through the system. Kafka records are made up of a key, value and headers. When an application sends data to or from Kafka it has to be serialized into raw bytes, as that is how Kafka stores it.

The way the data is serialized, and then deserialized, completely depends on the format of the data. For example the key might be a String, but the value could be JSON.

You can inadvertently deserialize JSON into a String, but trying to deserialize some data into JSON when it's actually a String can cause exceptions in your application. This can lead people to choose the String format for everything to reduce the chances of exceptions.

However, particularly for the value, a String often doesn't provide enough structure. For many use cases the data sent to and from Kafka needs structure, and perhaps even a schema so the structure can change and evolve as new capabilities are added to the system. As a result formats like JSON or Apache Avro are very common in data pipelines that use Kafka. Since these formats depend heavily on the use case, you need to think carefully about the best format for your specific system.

Producer and consumer applications use serializers and deserializers to configure how data should be translated before being sent to or when received from Kafka. If Connect is getting data out of Kafka, it needs to be aware of the serializers that were used to produce the data. If Connect is sending data to Kafka it needs to serialize it to a format that the consumers will understand. Often the format that the Kafka applications expect is different from the format in your external system. This is why Connect not only lets you configure the translation between the Kafka Connect internal format and the Kafka format, but also allows you to configure it completely independently of the connector you have chosen. So if your data is in a particular format in your external system, that doesn't mean it has to stay in that format when it reaches Kafka. Converters give you the flexibility to evolve the structure of the data as it flows through the system.

For Connect you don't configure a serializer and deserializer separately, instead you provide a single library, a converter, that can both serialise and deserialise the data for your chosen format. You will likely find you don't need to write one from scratch. There are a few provided as part of Kafka and many more created by the community. However, if you do need to write your own, we cover that in Chapter 13. To make a custom converter available to Kafka Connect, you add it as a new plugin.

For simple data types, converters manipulate data to conform to that type in the same way every time. For example, IntegerConverter is able to write and read as an integer. For complex data types such as JSON or Avro records, in order to manipulate data, converters need to know the exact schema the data is put in.

Converters and Schemas

Schemas describe the shape of your data. For example, the data could contain multiple fields with different types. For example, for the following record:

```
{
  "title": "Kafka Connect",
  "isbn": "123-45-67890-12-3",
```

```
  "authors": ["Kate Stanley", "Mickael Maison"]
}
```

The JSON schema could look like the following:

```
{
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "isbn": {"type": "string"},
    "authors": {
      "type": "array",
      "items": {"type": "string"}
    }
  },
  "required": ["title"]
}
```

Since a schema can evolve over time you need a mechanism to tell any consumers what schema a specific record is using. One way to do this is to include the schema alongside each record. This adds overhead to each record, so it is very common to use a schema registry instead. A schema registry allows you to store schemas in a central place and just include a reference to the schema with each record. Popular schema registries that work with Kafka are the Apicurio Registry and the Confluent Schema Registry.

Schema registries that are designed to be used with Kafka usually provide custom converters that you can use with Connect. The purpose of these custom converters is to perform the task of getting the schema from the schema registry and using it to correctly interpret the data. This also includes storing the schema id somewhere in the record before sending it to Kafka so that consuming applications can retrieve it when they read the message. If you are adding Kafka Connect to an existing system that already uses a schema registry, check if that registry provides a converter that implements the Connect converter API.

Configuring Connect with Converters

In Kafka Connect there are three different converters you can configure, one for the record key, the record value and finally the headers. The properties are called `key.converter`, `value.converter` and `header.converter` respectively. The `key.converter` and `value.converter` do not have a default value and must be configured when you start Kafka Connect.

By default, the `header.converter` is set to use the class `org.apache.kafka.connect.storage.SimpleHeaderConverter`. The class only serializes and deserializes header values, not the keys. The header values are serialized as strings, then when deserializing the class makes a best guess at what object to choose, for example boolean, array or map.

The converters that you specified as part of the runtime configuration are the default converters for every connector the Kafka Connect runtime starts. However, you can override this in your connector configuration. This allows you to set sensible defaults and then be very prescriptive about how data is structured as it flows through your system. This also means you can start multiple instances of the same connector, but running with a different converter. By allowing you to mix and match both the connectors and converters you can build a complex data pipeline without writing any new code.

Some converters have additional configuration options that you can apply to them. For example let's consider the JSON converter that is included in Connect. The JSON converter serializes and deserializes to and from JSON and has a configuration option called `schemas.enable`. If you enable this option the converter will include a JSON schema inside the JSON it creates, and look for a schema when it is deserializing data.

Let's say you want to use the JSON converter for your record keys, and to enable the schema. You will already have the configuration:

```
key.converter=org.apache.kafka.connect.JsonConverter
```

To enable a specific configuration option you add a line to your properties that specifies the name of the converter you want to configure, followed by the configuration option you want to set. So to set the `schemas.enable` configuration option to `true` you add the following:

```
key.converter.schemas.enable=true
```

Kafka Connect comes with some built-in converters to save you needing to write your own. These are:

- `org.apache.kafka.connect.json.JsonConverter`
- `org.apache.kafka.connect.storage.StringConverter`
- `org.apache.kafka.connect.converters.ByteArrayConverter`
- `org.apache.kafka.connect.converters.DoubleConverter`
- `org.apache.kafka.connect.converters.FloatConverter`
- `org.apache.kafka.connect.converters.IntegerConverter`
- `org.apache.kafka.connect.converters.LongConverter`
- `org.apache.kafka.connect.converters.ShortConverter`
- `org.apache.kafka.connect.storage.SimpleHeaderConverter`

Most of these are included in the Connect runtime or api jar files. The `JsonConverter` is included as a separate jar called *connect-json-*.jar* in the *libs* directory of

the Kafka distribution. This means you can configure your Kafka Connect runtime to use these converters without needing to put the jar files anywhere special.

Example

Let's see how using different converters can change the way data can appear in an external system. We are going to start two copies of the `FileStreamSink` connector, one using the `StringConverter` and the other using the `JsonConverter`. Make sure you have Kafka Connect running in distributed mode and a new topic called `topic-to-export-with-converters`. Create a file called *json-sink-config.json* with the following contents:

```
{
  "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
  "tasks.max": 1,
  "topics": "topic-to-export-with-converters",
  "file": "/tmp/json-sink.out",
  "value.converter": "org.apache.kafka.connect.json.JsonConverter",
  "value.converter.schemas.enable": "false"
}
```

Create a second file called *string-sink-config.json* that contains:

```
{
  "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
  "tasks.max": 1,
  "topics": "topic-to-export-with-converters",
  "file": "/tmp/string-sink.out",
  "value.converter": "org.apache.kafka.connect.storage.StringConverter"
}
```

Now run the following commands to start the two connectors:

```
$ curl -X PUT -H "Content-Type: application/json" \       http://local-
host:8083/connectors/file-json-sink/config --data "@json-sink-config.json"

$ curl -X PUT -H "Content-Type: application/json" \       http://local-
host:8083/connectors/file-string-sink/config --data "@string-sink-config.json"
```

Try sending some JSON messages to the topic-to-export topic:

```
$ bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic topic-to-export
> {"key":"value"}
```

Compare the contents of the two files:

```
$ cat /tmp/json-sink.out
{key=value}

$ cat /tmp/string-sink.out
{"key":"value"}
```

Both connectors read the same message from Kafka, however the one configured with the `JsonConverter` deserialized it as a JSON object when converting it to the Kafka Connect internal format. The `FileStreamSink` connector has then written the data into the file differently because it knew it was a JSON type.

Now try sending a message that isn't JSON. The message only appears in the *string-sink.out* file appear in the file and if you check the Kafka Connect logs you will see an exception from the `JsonConverter`:

```
Error converting message value in topic 'topic-to-export'
...
Caused by: com.fasterxml.jackson.core.JsonParseException: Unrecognized token
'foo'
...
```

One of the most common mistakes to make when using Kafka Connect is to not consider message serialization and deserialization. By thinking carefully about the converter you use you can avoid exceptions in your consuming applications and connectors as they try to consume from Kafka.

If all the data flowing through Connect is already in the right format with the right content you don't need to transform it! But on the other hand, if you want to slightly alter some data, this is where transformations come to the rescue.

Transformations

Transformations, often referred to as Single Message Transformations (SMT) or transforms, are also plugins you can add to a Kafka Connect runtime. As the name indicates, they allow you to transform messages that flow through Connect. This helps you get the data in the right shape for your use case before it gets to either Kafka or the external system, rather than needing to manipulate it later. A transformation is a Java class that implements the Transformation interface from the Connect API.

Contrary to connectors and converters, transformations are optional components in a Connect pipeline.

While it's possible to perform complex logic in transformations, it's best to stick to fast and simple logic. As a rule of thumb, transformations should not store states nor interact with remote APIs. A slow or heavy transformation can significantly affect the performance of a Connect pipeline. For advanced logic, it's best to use Kafka Streams.

In a Connect pipeline, a transformation is always associated with a connector. Transformations are always applied on `ConnectRecord` objects. For source connectors, as shown in Figure 2-6, transformations are invoked after the connector and before the converter.

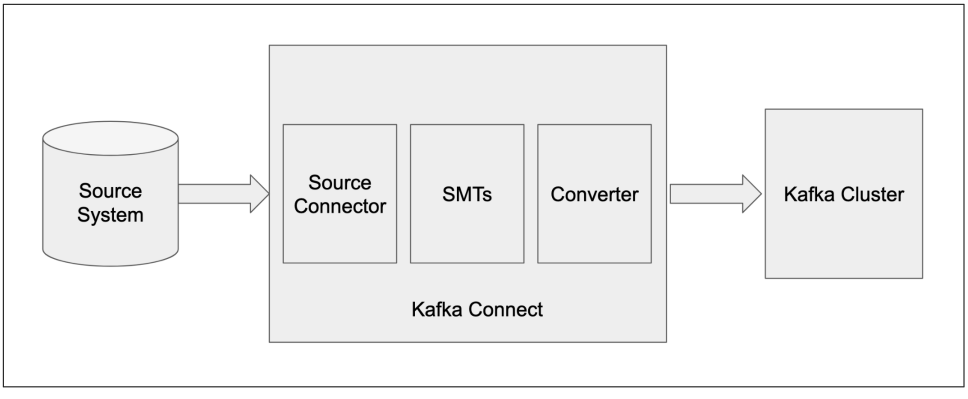


Figure 2-6. Source Connect pipeline

On the other hand, for sink connectors, transformations are invoked after the converter and before the connector, as shown in Figure 2-7.

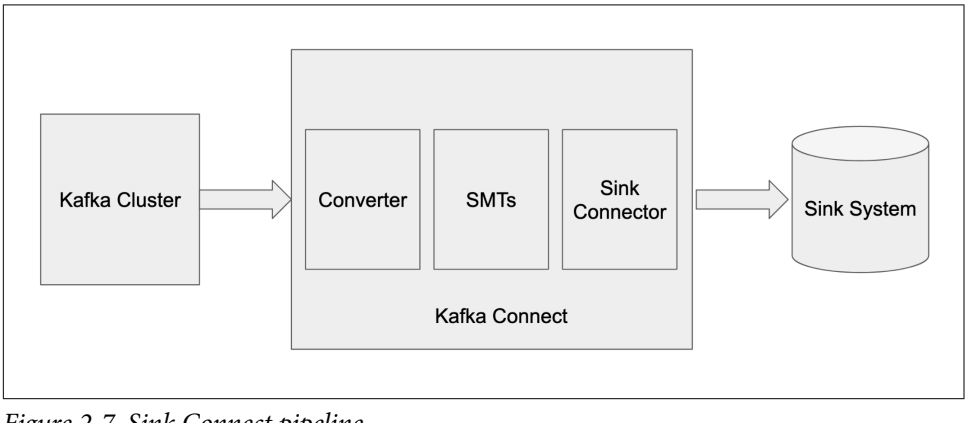


Figure 2-7. Sink Connect pipeline

It's possible to chain multiple transformations together in a specific order to perform several modifications. To enable transformations, you need to declare them in the configuration of a connector. Then these transformations will be applied to all the `ConnectRecord` objects this connector handles. Different connectors running in the same Connect cluster can have different transformations associated with them.

Single Message Transformations were initially introduced in Kafka 0.10.2.0 via KIP-66 in February 2017. Transformations related to headers shipped with 2.4.0 via KIP-440 and finally predicates in 2.6.0 via KIP-585.

What Can Transformations Do?

The main use cases for transformations are:

- Routing
- Sanitizing
- Formatting
- Enhancing

For each category, we will list the built-in transformations that enable the use case.

Routing

A routing transformation typically does not touch the key, value or headers of `ConnectRecord` but instead can change its topic and partition fields. This type of transformation is used to dynamically decide where each record will be written to.

This is useful when you want to split a stream of data into multiple streams, so let's look at a scenario. If you have a Kafka topic containing user interactions events, it may have different types of events like `UpdateAddress` and `MakeOrder`. When sinking this topic to a Cloud Object Storage system, a routing transformation enables you to send `UpdateAddress` and `MakeOrder` records to different stores or indexes.

Kafka comes with the following built-in transformations that allow routing records:

`RegexRouter`
: If the topic name matches a configurable Regular Expression (regex), the topic in the `ConnectRecord` is replaced by a configurable value.

`TimestampRouter`
: Injects the record timestamp, with a configurable format, into the topic name.

Sanitizing

Sanitizing transformations allow you to remove data that you don't want to flow downstream in your Connect pipeline. This type of transformation involves directly altering the content of `ConnectRecord` objects or completely discarding them.

This is useful for removing sensitive data such as credentials, personally identifiable information (PII), or simply data that is of no-use to downstream applications.

The following transformations are built-in Kafka and allow sanitizing records:

`DropHeaders`
: Removes headers whose keys match a configurable list.

`MaskField`
: Given a field in the content, replaces its value with its default null value or a configurable replacement.

`Filter`
: Drops the record. This is always used with a Predicate.

Formatting

Formatting transformations allow you to change the schema of `ConnectRecord` objects. It can be useful to move fields around or change the type of some fields to make the data easier to consume downstream. This can also be used to shape `ConnectRecord` objects into the format the converter is expecting.

For example, `ConnectRecord` objects may come using this JSON schema:

```
{
  "type": "struct",
  "fields": [
    {"type": "string","field": "item"},
    {"type": "string","field": "price"}
  ]
}
```

It would be preferable to have the `price` field as a number instead of as a string. In this case, you can use a transformation to change the type of this field.

The following transformations are built-in in Kafka and allow formatting records:

`Cast`
: Casts a field into a different configurable type

`ExtractField`
: Extracts a configurable field and throws away the rest of the record

`Flatten`
: Flattens the nested structure of the record and renames fields accordingly

`HeaderFrom`
: Copies or moves a field from the record value to its headers

`HoistField`
: Wraps the record's fields with a new configurable field

`ReplaceField`
: Renames fields using a configurable mapping

`SetSchemaMetadata`
: Sets the key or value schema to configurable values

`ValueToKey`
: Replaces the record's key with configurable fields from the record's value

Enhancing

Enhancing transformations allow you to add fields and headers or improve the data in some fields. In many cases, it is useful to inject additional data to records passing through a pipeline. This can be used for data lineage, tracing or even debugging.

For example, you can use an enhancing transformation to inject a new field with the record timestamp. Even though the `ConnectRecord` object has a dedicated timestamp field, when it is exported to an external system, some connectors might not include it. To solve this issue, you can use `InsertField` to inject a field with the timestamp value.

The following transformations are built-in in Kafka and allow enhancing records:

`InsertField`
: Inserts fields with configurable values

`InsertHeader`
: Inserts headers with configurable values

`TimestampConverter`
: Converts timestamp fields using a configurable format

Note that while these categories are helpful to identify use cases, it's possible for a single transformation to perform several of these. Each transformation is not limited to perform a single modification. Sometimes, you can chain multiple single purpose transformations and in other cases you may prefer using a single transformation that does multiple modifications.

To see the complete list of the Apache Kafka built-in transformations and their associated configuration options, see the Transformations section on the Kafka website.

As with connectors and converters, the Kafka community has built transformations for many use cases. But again if you can't find one for your use cases, as it's a pluggable component, you can write your own transformations. This is covered in detail in Chapter 13.

Configuring Transformations

Before detailing how to configure transformations, let's look at predicates.

Predicates

Predicates allow you to apply a transformation only if a configurable condition is met. Some transformations are intended to be used with a predicate, such as `Filter` which otherwise would apply to all records and result in all records being dropped. But this is also really useful with many other transformations. For example if a stream contains several types of events you can apply certain transformations to certain events.

These are the built-in predicates in Kafka:

`HasHeaderKey`
: Is satisfied if the record has a header with a configurable name

`RecordIsTombstone`
: Is satisfied if the record is a tombstone, i.e., the value is null

`TopicNameMatches`
: Is satisfied if the record's topic name matches a configurable regex

Predicates are also pluggable and if the built-in predicates don't satisfy your use case, you can implement your own. This is also covered in Chapter 13.

In Kafka, a record is called a tombstone if its value is `null`. The name comes from compacted topics where a record with a null value acts as a delete marker and causes all previous records with the same key to be deleted during the next compaction cycle.

Configuration syntax

You specify transformations (and predicates) alongside the connector configuration. The syntax to define transformations is a bit convoluted so let's look over a simple example to see how it works.

```
{
  "name": "my-connector",
  "config": {
    [...] ❶
    "transforms": "addSuffix", ❷
    "transforms.addSuffix.type": "org.apache.kafka.connect.transforms.Regex-
Router", ❸
    "transforms.addSuffix.regex": "(.*)", ❹
    "transforms.addSuffix.replacement": "$1-router" ❺
```

```
  }
}
```

❶ This is the regular connector configuration, like you have seen in previous sections.

❷ You first need to list the transformations you are going to use. The `transforms` field uses a comma separated value of names for the transformations. Transformations will be applied in the order they are specified in this field.

❸ You define the actual transformation class to use for each name listed above. In this case, we defined a single name `addSuffix` so here we specify the fully qualified class name we want to use using the `type` field.

❹ Then you define the configurations specific for this transformation. The first configuration `RegexRouter` uses is the `regex` field that is set to `(.*)`.

❺ The remaining configuration of `RegexRouter` is `replacement` that defines the suffix to add.

With this transformation, all `ConnectRecord` objects emitted by your connector will have the `-router` suffix added to their topic.

Predicates are defined using the same syntax but use the `predicates` prefix. Let's look at an example mixing both transformations and predicates:

```
{
  "name": "my-connector",
  "config": {
    [...]
    "transforms": "filterTombstones",
    "transforms.filterTombstones.type": "org.apache.kafka.connect.trans-
forms.Filter",
    "transforms.filterTombstones.predicate": "isTombstone", ❶

    "predicates": "isTombstone", ❷
    "predicates.isTombstone.type": "org.apache.kafka.connect.transforms.predi-
cates.RecordIsTombstone" ❸
    [...] ❹
  }
}
```

❶ All transformations accept a single predicate field to specify which predicate must be satisfied for it to be applied.

❷ Like for transformations, you start by listing predicates you are going to use. The predicates field uses a comma separated value of names for the predicates.

❸ You define the actual predicate class to use for each name listed above. In this case, we defined a single name `isTombstone` so here we specify the fully qualified class name we want to use using the `type` field.

❹ Predicates can also have specific configurations. Like for transformations, the syntax is:

```
predicates.<predicate_name>.<configuration>=<value>
```

With this transformation, all `ConnectRecord` objects emitted by your connector that are tombstones will be dropped and not passed to the rest of the Connect pipeline.

Predicates can also be negated, if you want to test for the opposite condition. This allows using the same small set of predicates for both conditions. To do so, you set the `negate` field on the connector to `true`. For example:

```
{
[...]
"transforms.myTransformation.predicate": "topicMatch",
"transforms.myTransformation.negate": "true",

"predicates": "topicMatch", (2)
        "predicates.topicMatch.type": "org.apache.kafka.connect.transforms.pred-
icates.TopicNameMatches" (3)
        "predicates.topicMatch.pattern": "mytopic.*",

[...]
}
```

In this case, the `myTransformation` transformation is only applied if the `topicMatch` predicate is not satisfied because the record topic name does not start with `mytopic`.

Key and value transformations

In Kafka, the record's key and value can contain arbitrary data. In fact, in `ConnectRecord`, both the key and value fields are defined as `Object` and each has a `Schema` field (keySchema and valueSchema) associated.

This means that many transformations can be applied on the value or on the key. If that's the case, there are often two different classes, one that applies to the key and one to the value. You need to make sure you specify the correct class in the `type` configuration of the transformation.

For example, the transformation `Cast` exposes two classes:

`org.apache.kafka.connect.transforms.Cast$Key`
: For casting a field in the key

`org.apache.kafka.connect.trans forms.Cast$Value`
: For casting a field in the value

Enabling Transformations in Your Pipeline

Let's now enable some transformations in your Connect pipeline. Make sure you have Kafka Connect running in distributed mode and a new topic called `topic-to-export-with-transformations`.

First we need to update the connector configuration in a file called *file-sink.json*:

```
{
  "name": "file-sink",
  "config": {
    "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": 1,
    "topics": "topic-to-export-with-transformations",
    "file": "/tmp/sink.out",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false"
    "transforms": "replaceSource,addTimestamp", ❶
    "transforms.replaceSource.type": "org.apache.kafka.connect.transforms.Repla-
ceField$Value", ❷
    "transforms.replaceSource.renames": "source:origin", ❸
    "transforms.addTimestamp.type": "org.apache.kafka.connect.transforms.Insert-
Field$Value", ❹
    "transforms.addTimestamp.timestamp.field": "ts" ❺
  }
}
```

❶ You define two transformations you want to apply: `replaceSource` and `addTimes tamp`.

❷ The first one is `ReplaceField.` Note that we specified the `ReplicaField$Value` class to apply the transformation on the value.

❸ The `source` field will be replaced by `origin`.

❹ The second transformation is `InsertField`, again on the value.

❺ It will insert the timestamp into a new field named `ts`.

Then you produce another record to topic-to-export

```
$ ./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic topic-to-export-with-transformations
> {"source": "kafka-console-producer", "type": "event"}
```

Now start the connector:

```
$ curl -X PUT -H "Content-Type: application/json" \       http://local-
host:8083/connectors/file-sink/config --data "@file-sink.json"
```

The connector processes that record and append the following to your file:

```
$ cat /tmp/sink.out
{"origin":"kafka-console-producer","type":"event","ts":"Mon Nov 22 10:38:05 CET
2021"}
```

Summary

In this chapter we explored the main components of a Connect data pipeline and built a simple pipeline exporting records from a topic to a file.

We first looked at the Connect runtime including its artifacts and modes of operations. To recap, the Connect runtime can run in the following modes:

Standalone
: Only suitable for basic development due to its lack of resiliency and limitations managing connectors

Distributed
: Suitable for both development and production since it provides strong resiliency, scalability and management capabilities

We would recommend that you use distributed mode wherever possible, as it is easy to run on any system and allows you to run with the same mode in both development and production.

We then introduced connectors and described how they import and export data between Kafka and external systems. The two types of connectors are:

Sink
: Used to export data from Kafka

Source
: Used to import data into Kafka

We also covered converters and understood their role in translating data between formats and ensuring data stays consistent for applications consuming it.

Finally, we looked at transformations and predicates and explained how they can be used to fully control the content and format of data flowing through Connect.

CHAPTER 3
Building Effective Data Pipelines

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *KafkaConnectBook@gmail.com*.

In this chapter you will learn how to build resilient and effective data pipelines using Connect. We will explain the key concepts and decision points data engineers and architects have to understand when assembling the different components we introduced in Chapter 3. We will start by looking at important factors you need to take into account when selecting connectors from the hundreds available in the community and then describe how to model data effectively to satisfy your production requirements.

We will also dive into the resiliency characteristics of Connect. By detailing some of its inner workings, we'll explain why Connect is a robust environment, able to handle failures. We will show you how to understand the end to end delivery semantics that sink and source pipelines can achieve and the different configuration options and trade offs available to target your specific use cases.

Connect can run in Standalone or Distributed mode. Standalone only provides bare bones resiliency features and it is not suitable for production environments. In this chapter we will focus on Distributed mode which is able to handle many types of

failures at many levels in the stack, from the machine that's running Connect workers, down to individual tasks.

Choosing a Connector

When building a data pipeline that uses Connect, you first need to decide which connectors to deploy. Since Kafka is a very popular technology there are many existing connectors for you to choose from. Rather than reinventing the wheel it is better to use an existing connector if you can, but only if it fulfills your requirements. Here are some things to consider when choosing whether to use a specific connector as part of your pipeline:

- Flow direction - source or sink
- Licensing and support
- Connector features

Flow Direction

First verify that the connector flows data in the right direction, i.e. is it a source connector that sends data to Kafka or a sink connector that consumes from Kafka. Most connectors include this detail as part of the name, and it is usually clear from the documentation. In case it is not, you can install the connector in a Connect environment and use the REST API to retrieve its type.

```
$ curl localhost:8083/connector-plugins
[
  {
    "class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type":"sink",
    "version":"3.0.0"
  }
]
```

The `type` field indicates the type of the connector.

Some projects provide a single download that includes both a source and sink connector but some other projects may provide only one or the other.

Licensing and Support

Before using a connector make sure to check what its license permits. Just because a connector is open source or freely available to download doesn't mean the license is permissive. You should also consider the level of support you need. The Kafka community works hard to make sure that older clients can work with newer versions of the runtime, however it is still preferable for your connector to be updated regularly.

Whatever connector you choose, whether it's open source or proprietary, make sure you know how often the connector is updated with the latest Kafka APIs and how the developers address security vulnerabilities.

Since a single connector is often used for many different use cases you may find there isn't one perfectly suited to your needs. If that is the case instead of writing one from scratch we would encourage you to see if there is an open source connector that you could contribute to. You still need to get your changes accepted but most open source projects are very welcoming of new contributors.

Connector Features

Once you have identified potential connectors for your pipeline, you will need to take a closer look at the features offered by those connectors. To start with, does the connector support the type of connection you need? For example, your external system might require an encrypted connection, some form of authentication or for the data to be in a specific format. You should also check if the connector is suitable for production use. For example, does it provide metrics for monitoring its status and logging to help you debug problems? Look over the documentation and, for an open source connector, the code, to see how the connector works and assess the features it provides.

In Chapter 3 we introduced the common configuration options for all connectors; `topics` for sink connectors and `tasks.max` for both source and sink. Most connectors offer additional options to configure their specific features. For a specific connector you can use the REST API to list all of the available configuration options and validate your configuration before starting the connector.

Using the REST API is especially useful if the code is not available, but be aware that this relies on the developer documenting their configuration correctly. Some fields might be incorrectly marked as optional or required.

To list the configuration options for `FileStreamSinkConnector`:

```
$ curl http://localhost:8083/connector-plugins/org.apache.kafka.con-
nect.file.FileStreamSinkConnector/config
[
  {
    "name": "file",
    "type": "STRING",
    "required": false,
    "default_value": null,
    "importance": "HIGH",
    "documentation": "Destination filename. If not specified, the standard out-
put will be used",
    "group": null,
    "width": "NONE",
    "display_name": "file",
```

```
    "dependents": [],
    "order": -1
  }
]
```

To validate a specific configuration JSON object:

```
$ curl -X PUT -d '{"connector.class":"org.apache.kafka.connect.file.FileStream-
SinkConnector",...}' http://localhost:8083/connectors/MyConnector/config/vali-
date
{
  "configs": [{
    "definition": {"name": "topics", "importance": "HIGH", "default_value":
null, ...},
    "value": {
    "errors": ["Missing required configuration \"topics\" which has no default
value."],
    ...
}
```

Defining Data Models

No two pipelines are identical. Even if they fulfill a similar use case or use the same components, the actual data and how that data evolves varies from pipeline to pipeline. When you are designing your pipeline you need to consider when and how each individual data entry will change, but also how the individual entries relate to each other. How you group or split (shard) your data will affect how well you can scale your pipeline as the amount of data it is processing increases. To examine these ideas in more detail we will first discuss when to apply data transformation in Connect and techniques for mapping data between Connect and other systems.

Data Transformation

There are two common patterns that are used to evolve data as it flows through a pipeline: ETL (Extract, Transform, Load) (Extract, Transform, Load) and ELT (Extract, Load, Transform) (Extract, Load, Transform). In these patterns the word "Transform" doesn't just refer to updating the format. Transformation could include cleaning the data to remove sensitive information, collating the data with other data streams or performing more advanced analysis.

Both approaches have their advantages and disadvantages. In systems where storage is restricted it is better to use the ETL approach and use the ETL approach and transform the data before loading it into storage. This makes it easy to query the data because it has already been prepared for analysis. However, it can be difficult to update the pipeline if a new use case that requires a different transformation is discovered. In contrast, ELT keeps the data as generic as possible for as long as possible, giving the opportunity for the data to be reused for other purposes. The

ELT pattern has been gaining popularity and there are now many dedicated data processing and analysis tools that are built to support this pattern. Some examples of these tools are Kafka Streams, Apache Spark, Apache Flink, Apache Druid and Apache Pinot.

So where do Connect transformations fit in this flow? In Connect there is a rich set of transformations that you can perform on your data while it is in flight, so this fits naturally into the ETL pattern. Using Connect for your transformations removes the need for a separate tool to transform the data before loading it. Since you choose the specific set of transformations to apply and Connect allows you to plug in custom ones, the possibilities are endless. However Connect transformations do have their limitations because they are applied to each piece of data independently. This means you can't perform more advanced processing like merging two streams of data or aggregating data over time. Instead you should use one of the dedicated technologies for these kinds of operations.

Even if you decide to use a dedicated technology for the bulk of your data processing and analysis, you can still make use of Connect transformations. The particular transformations you might want to consider are the ones that remove or rename fields and can drop records. These are very useful for ensuring that sensitive data isn't sent further down the pipeline and for removing data that could cause processing problems later. If you have multiple different sources that need to be aggregated in subsequent steps you can also use Connect transformations to first align the data to have common fields. Figure 3-1 shows this sort of flow.

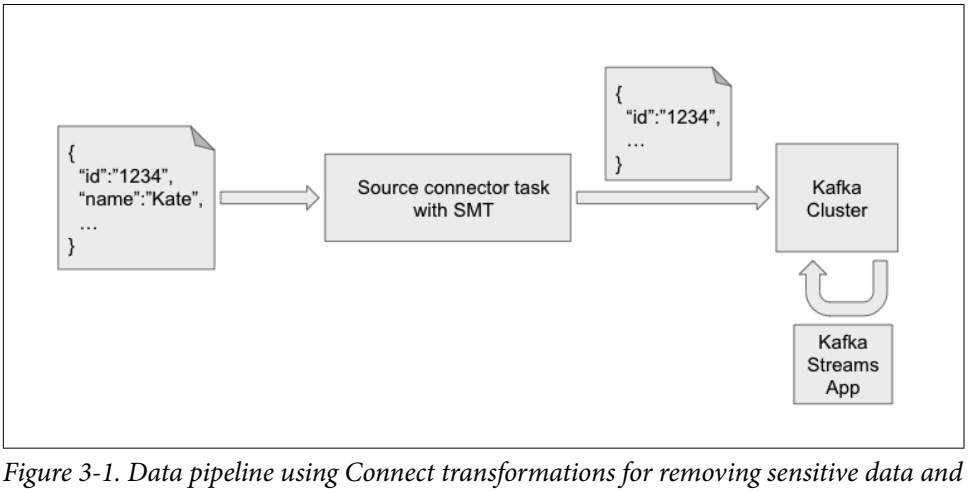


Figure 3-1. Data pipeline using Connect transformations for removing sensitive data and Kafka Streams to perform further processing.

We have discussed how you can transform individual data entries, but what about the grouping of your data as a whole?

Mapping Data Between Systems

One of the hardest things to reason about in a data pipeline is how to map the data structures between different systems. Here we mean more than just the format of an individual entry, but how the data should be grouped and stored, what ordering is required and what happens when the pipeline needs to be scaled. In Connect a lot of these decisions are made for you by the developer who wrote the connector. However, you should still be aware of the mechanisms that are available for connectors to use when mapping data between Kafka and other systems. If you understand these mechanisms, you are better equipped to assess a connector you want to use and then configure the connector correctly for your pipeline.

To understand the way connectors can group and map data, you need to consider the interaction between Connect tasks and Kafka partitions. In Chapter 3 we introduced tasks as the mechanism that Connect uses to do the actual work of transferring data from one place to another. In Chapter 2 we talked about partitions and highlighted the fact that Kafka provides ordering guarantees within a single partition. Both mechanisms provide a way to shard data.

Let's first look at the impact of tasks. When a source connector reads data from an external system, each task is reading data in parallel. It is up to the connector to decide how to split this data amongst the available tasks to ensure there are no duplicates. A simple connector could run a single task and avoid the problem of sharding the data that's in the external system. This is actually how `FileStreamSourceConnector` that is packaged with Kafka works. See Figure 3-2 for an example.

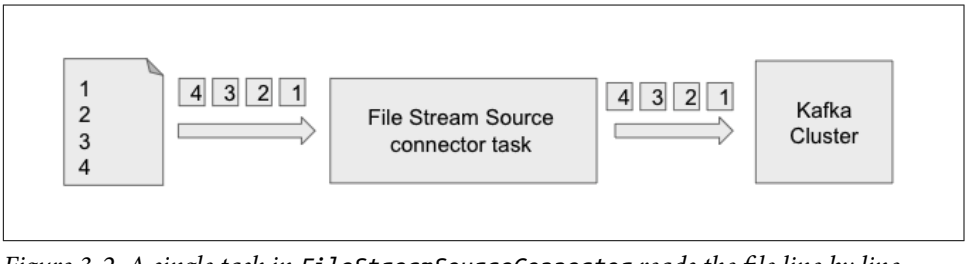


Figure 3-2. A single task in `FileStreamSourceConnector` reads the file line by line.

Even if you increase the `max.tasks` setting, it will still only run a single task, because it doesn't have a sensible mechanism to shard the data. Most connectors are more advanced than `FileStreamSourceConnector` and have built in mechanisms to assign the data across the tasks. Figure 3-3 shows an example of such a connector that allows different tasks to read different lines of a table.

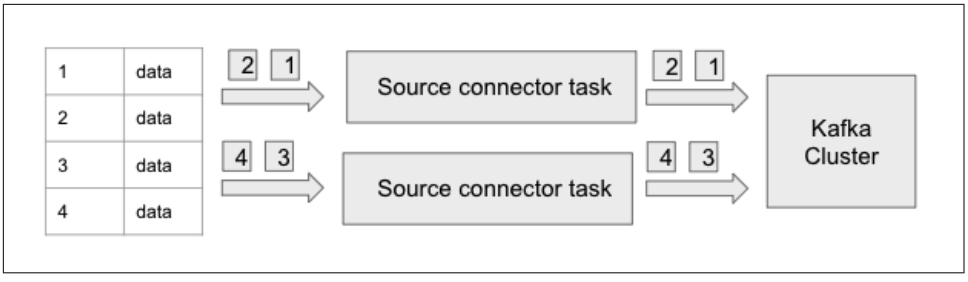


Figure 3-3. Multiple tasks that each read a subset of the data, preventing duplicates in Kafka.

In sink connectors, tasks also run parallel. This can affect the order that data is sent to the external system. For example if you had a `FileStreamSourceConnector` with two tasks, there is no way to determine the final order that the data entries would end up in the file. You can be sure that each task will write its own data in order, but there isn't any order coordination between tasks.

Now let's consider partitions. An individual source connector can either choose which records should go to which partitions or rely on the configured partitioning strategy. Many connectors use keys to identify the data that needs to be sent to the same partitions, for example status updates that apply to a particular entity might use the entity id as the key. Figure 3-4 shows an example of tasks sending data to partitions.

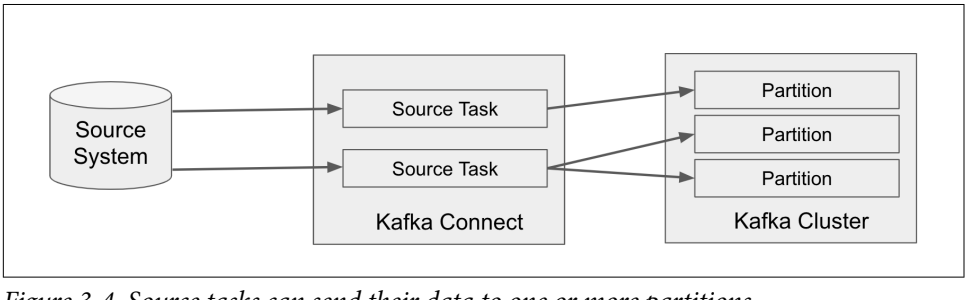


Figure 3-4. Source tasks can send their data to one or more partitions.

How a source connector partitions its data will affect the next stage of the pipeline, whether that next stage is a sink connector, or just a Kafka consumer. That is due to the way Kafka distributes partitions among both sink tasks and consumers from a group. Each partition can only be assigned to a single sink task of a particular connector and similarly a single consumer within a particular group. So any data that needs to be read by a single task or consumer, needs to be sent to the same partition by the source connector.

The way sink tasks interact with partitions also impacts the number of sink tasks you can run. If you have one partition and two tasks, only one task will receive any data. So when creating a data pipeline with a sink connector, make sure you are mindful about the number of partitions on the topics the connector is reading from. Figure 3-5 shows two sink tasks reading data from three partitions.

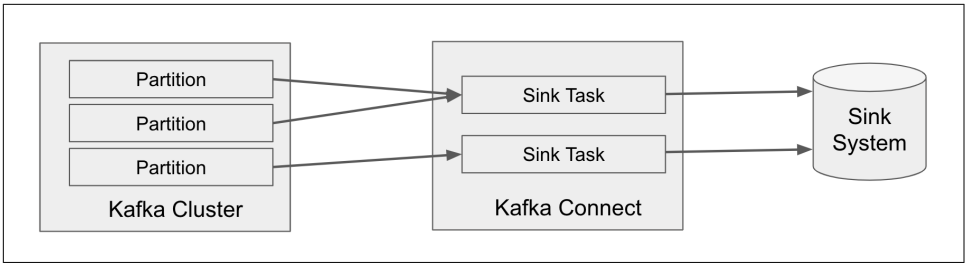


Figure 3-5. Each partition can only be read by one sink task for a specific connector.

As you can imagine, the combination of tasks and partitions means there are multiple ways that the data can be grouped and ordered as it flows through the system. When you are designing your Connect data pipeline make sure you think about these options and don't leave the `max.tasks` and partitions configuration options as an afterthought.

Now that we have looked at how data at a high level can be transformed and mapped between systems, let's look at how you can control the specific format of data in a Connect pipeline.

Formatting Data

In Chapter 3 we talked about converters and how they serialize and deserialize data as it goes into and out of Kafka. We also briefly covered why you need to align your converters with the serializer and deserializer of your producers and consumers that are also interacting with the data. Here we will discuss in more detail the difference between converters, transformations and connectors and how they impact the data format throughout the pipeline. We will also look at how you can enforce this structure with schemas and a schema registry.

Data Format

In a Connect pipeline the format of the data and how it evolves depends on the connector, any configured transformations, and the converter. Let's look at each of these in turn and how they affect the data format.

First let's consider the connector. In a source flow the connector runs first, it reads the data from the external system and creates a `ConnectRecord`. The connector decides

which parts of the data should be kept and how to map them to the `ConnectRecord`. The specifics of this mapping can differ between connectors, even if they are for the same system. So make sure the connector you choose keeps the parts of the data that are important to you.

In a sink flow the connector is run last rather than first. It takes `ConnectRecord` objects and translates them into data objects that it can send to the external system. This means that a sink connector has the last say on what data makes it to the external system. Keep this in mind when you are adding transformations and converters. Make sure you aren't wasting processing time on fields that will be ignored by the sink connector.

Next we'll look at the difference between converters and transformations when it comes to their input and output:

- Transformations have `ConnectRecord` objects as both their input and output.
- Converters convert between `ConnectRecord` objects and the raw bytes that Kafka sends and receives.

Figure 3-6 shows the different data types that are passed between connectors, transformations and converters.

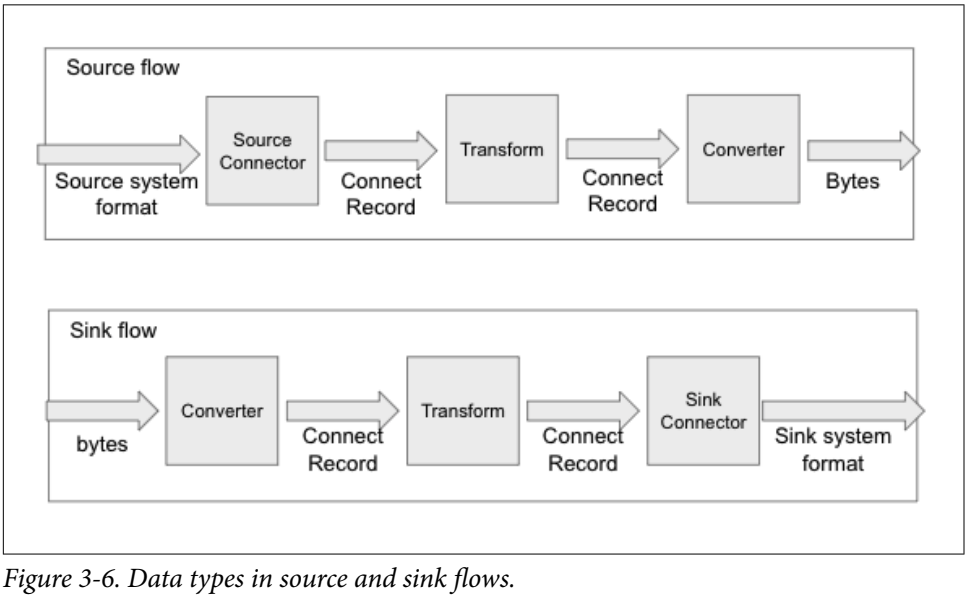


Figure 3-6. Data types in source and sink flows.

The reason transformations and converters are separate steps is to enable the composability that Connect offers. You could write a JSON converter that manipulates the contents of the record before sending it to Kafka. This would work fine, but if you

also wanted a pipeline that manipulated the data in the same way, but instead used a format like Avro, you would need a new converter. It would be much better to create a transformation that manipulates the data and then have two converters, one for JSON and one for Avro.

This is also true for transformations. Transformations are designed to be stacked, so it is better to run multiple simple transformations that together fulfill your requirements, than to write a custom one that only works for your very specific use case. If you can't find a transformation or converter that fits your requirements you can write your own (see Chapter 13), but try to write them in a way that will allow future reuse.

Now that you understand the individual roles of connectors, transformations and converters, and the order they run in, you can make a better informed decision of which libraries to use for your pipeline. This will make it easier to get the exact data format you need at each stage in the pipeline.

Schemas

A schema provides a blueprint for the shape of the data. For example, a schema can specify which fields are required and the expected types that should be present. When you are building a data pipeline it is important to use schemas because most data is complex and contains multiple fields of different types. Without schemas to give context to the data, it is very difficult for applications to reliably perform the steps to process and analyze it.

Almost all systems that deal with data provide a mechanism to define schemas. Although the specific schemas of your systems will vary, here's how Connect pipelines make use of schemas. As we saw in the previous section, data transitions between two different formats while passing through Connect, the `ConnectRecord` and raw bytes. Each format has a different mechanism that is provided to configure the schema.

A `ConnectRecord` contains a `Schema` object for both the key and the value. `Schema` is a Java class that is part of the Connect API and is used by connectors, transformations and converters as the data travels through Connect. Let's look at how the schema is used in both a source and sink flow.

A source connector is responsible for constructing the initial `ConnectRecord` object and has control over the schema that is added. The way the schema is defined depends on the connector. `FileStreamSourceConnector` always uses the `STRING_SCHEMA` no matter what format the file is using. You can see this in the source code:

```
private static final Schema VALUE_SCHEMA = Schema.STRING_SCHEMA;
@Override
public List<SourceRecord> poll() throws InterruptedException {
  ...
  records.add(new SourceRecord(offsetKey(filename), offsetValue(streamOffset),
```

```
    topic, null, null, null, VALUE_SCHEMA, line, System.currentTimeMillis()));
      ...
    }
```

Most connectors are more complex than `FileStreamSourceConnector` and will make use of schemas provided by the system. For example the Debezium connectors that read database change logs will take note of schema changes and use that information to construct the `ConnectRecord`. The `ConnectRecord` and included schemas are then passed to any transformations and to the converter. Transformations and converters use the schema to parse the `ConnectRecord` and do their respective work.

In a sink flow, it is the converter that constructs the `ConnectRecord` and therefore the `Schema`. Again the transformations use this information for parsing the contents. Sink connectors use the `ConnectRecord` to construct the object that is sent to the external system. This means they can choose how to interpret the schema that is included in the `ConnectRecord`. For example `FileStreamSinkConnector` ignores the schema completely, however that is only because it is writing to a file. Most sink connectors will use the schema information to construct the external system data.

The second type of schemas in a Connect flow is the one used for the raw bytes that are sent to Kafka. These schemas are used by converters to understand how to serialize and deserialize the data that is sent to and from Kafka. In a sink flow, the schema Connect uses to deserialize data is the same one that is used by the applications that originally produced the data. On the other hand, in a source flow, the schema Connect uses to serialize the data is also used by consuming applications or sink connectors, to deserialize it further down the pipeline.

If you are using a schema for your data in Kafka you should also use an external schema registry as part of your Connect pipeline. The next section will look at why a schema registry is important and the most commonly used ones for Kafka.

Schema Registry

Before explaining what a schema registry is, let's consider the impact of schemas on your pipeline without one. Say you have some JSON data that you want to send to Kafka and have consumed later:

```
{
  "productId": "1234",
  "productName": "Connect Book"
}
```

As long as the applications that are sending and receiving this kind of data agree on the schema, the pipeline will work fine. However, what if you decide to add a new field to your JSON, such as price. Without an updated schema to define the latest expected format, applications might reject the new data and as a result your pipeline would be unstable.

So the next question is how do you make sure that consuming applications know what schema to validate against. Well you could send the schema with every message. This is actually what the `JsonConverter` does by default. If you run `FileStreamSourceConnector` against a file with the following contents:

```
This is a string
Another string
A third string
The final string
```

The `JsonConverter` will use the `String` schema that the connector provides and construct Kafka records with the values as:

```
{"schema":{"type":"string","optional":false},"payload":"This is a string"}
{"schema":{"type":"string","optional":false},"payload":"Another string"}
{"schema":{"type":"string","optional":false},"payload":"A third string"}
{"schema":{"type":"string","optional":false},"payload":"The final string"}
```

Although this makes it easy to pass a schema along for consumers it means that every single record has to include the schema. The example here is simple, so the schema is small, but the more complex the schema the bigger the overhead for each record. As you can imagine this quickly becomes unsustainable. Instead you should consider using a schema registry.

A schema registry is a central store for your schemas. If you use a schema registry with Kafka you only need to send the id of the schema with each record, not the entire schema. Then the converter or consumer can use the id to look up the schema in the registry. There are two schema registries that are most commonly used with Kafka: the Confluent Schema Registry and the Apicurio Registry.

Both of these registries allow you to use Kafka as the backing store for the registry, removing the need for a separate database or other storage system. They also both support the most common schema formats that are used with Kafka: Avro, JSON schema and Protobuf.

A detailed comparison of the available schema formats and schema registries for Kafka is outside the scope of this book, however we can give some pointers. To choose a format make sure you consider the tools and libraries that go along with each. For example, do they support the language you want and provide code generation options? The schema registry you choose will influence your converter and serializer/deserializer options. The Confluent schema registry will only work with Confluent libraries, whereas Apicurio Registry comes with a compatibility API which means you can use the dedicated Apicurio Registry libraries or the Confluent ones.

The last thing we will highlight around schema registries is how the schema id is sent with the record. This is important for Connect pipelines because different converters use different mechanisms when serializing Kafka data. Most converters use one of two mechanisms:

- The schema id is added as a record header.
- The schema id is included at the beginning of the serialized value.

If you are building a source flow make sure you choose a converter that not only works with your schema registry, but also will store the id in a place that is expected by the downstream applications that will consume the record. Similarly if you are building a sink flow, choose a connector that can contact your schema registry and knows where in the record to look for the id.

Exploring Connect Internals

In order to understand how Connect in distributed mode can withstand failures, you should first understand how it stores its state by using a mix of internal topics, and group membership. Secondly, you should be familiar with the rebalancing protocol Connect uses to spread tasks across workers and detect worker failures.

Internal Topics

As explained in Chapter 2, Connect automatically creates and uses 3 topics:

- Configuration topic, specified via `config.storage.topic`
- Offsets topic, specified via `offsets.storage.topic`
- Status topic, specified via `status.storage.topic`

In the configuration topic, Connect stores the configuration of all the connectors and tasks that have been started by users. Each time users update the configuration of a connector or when a connector requests a reconfiguration (for example it detects it can start more tasks), a record is emitted to this topic. This topic is compacted, so it always keeps the last state for each entity while ensuring it does not use a lot of storage.

In the offsets topic, Connect stores offsets of source connectors. Again this topic is compacted for the same reasons. By default, Connect will create this topic with several partitions, as each source task uses it regularly to write its position. Offsets for sink connectors are stored using regular Kafka consumer groups.

In the status topic, Connect stores the current state of connectors and tasks. This topic is used as the central place for the data that is queried by users of the REST API. It allows users to query any worker and still get the status of all running plugins. It is also compacted and should also have multiple partitions.

At startup, Connect automatically creates these topics if they don't already exist. All workers in a Connect cluster must use the same topics, but if you are running

multiple Connect clusters, each cluster needs its own separate topics. Data within all these 3 topics is stored in JSON so it can be viewed using a regular consumer.

For example, with the `kafka-console-consumer.sh` tool, here's how you can view the content of the status topic:

```
./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
                                --topic connect-status \
                                --from-beginning \
                                --property print.key=true
status-connector-file-source {"state":"RUNNING","trace":null,
"worker_id":"192.168.1.12:8083","generation":5}
```

In this example, the runtime has `status.storage.topic` set to `connect-status`, a connector named `file-source` and Connect that name to derive the key, `status-connector-file-source`, for records about this connector.

Group Membership

In addition to topics, Connect makes extensive use of Kafka's group membership API.

First, for each sink connector, the Connect runtime runs a regular consumer group that extracts records from Kafka. The groups are named after the connector name, for example for a connector named `file-sink`, the group will be `connect-file-sink`. Each consumer in the group is providing records to a single task. These groups and their offsets can be retrieved using regular consumer groups tools, such as `kafka-consumer-groups.sh`.

In addition, Connect uses the group membership API to assign tasks onto workers and ensure each partition is only consumed once. At startup, Connect creates a group using the `group.id` value from its configuration. This group is not directly visible by the consumer group tools as it's not a typical consumer group but it works in essentially the same way. This is why all workers with the same `group.id` value, become part of the same Connect cluster.

To be a member of a group, workers, just like regular consumers, have to *heartbeat* regularly. A heartbeat is just a request that contains the group name, the member ID and a few more fields to identify the sender. It is sent at regular intervals (specified by `heartbeat.interval.ms`, by default 3 seconds) by all workers to the broker that acts as the coordinator for their group. If a worker stops sending heartbeats, the coordinator will detect it, remove the worker from the group and trigger a rebalance. During a rebalance, tasks are assigned to workers using a rebalance protocol and once this is complete, workers resume their work.

Rebalancing Protocols

The specifics of rebalancing protocols are generally hard to comprehend. Thankfully, to use Connect effectively, it's enough to understand the high level process that is described in this section.

As mentioned, Connect wants to ensure that all tasks are running, that each task is run by a single worker and that tasks are spread evenly across all workers. A rebalance happens anytime the resources that are managed by Connect change, for example, when a worker joins or leaves the group, or when tasks from a connector are added or removed. When this happens Connect has to *rebalance* tasks across the workers.

The mechanism that Connect uses has changed over time. We will talk about the different approaches that have been taken over time and explain why the current approach has been chosen.

Until Kafka 2.3, during a rebalance Connect would simply stop all tasks, and reassign them all onto the available workers. This is called the `eager` rebalance protocol, also often referred to as "stop the world". The main issue with this protocol is that Connect can run a set of independent connectors, and each time one of them decides to create or delete tasks, all connector tasks are stopped, then they are reassigned to workers and they finally restart running. In a busy Connect cluster, this can lead to long and repetitive pauses in the processing. This approach also makes rolling restarts very expensive as each worker causes two rebalances to happen, one when it shuts down and another one when it restarts.

In Kafka 2.3, Connect introduced an incremental cooperative rebalancing protocol called `compatible`. The idea is to avoid stopping all connectors and tasks each time a rebalance happens, and instead only rebalance the resources that need to be rebalanced, and do it incrementally if possible. For example, if a worker disappears, Connect will first wait a short duration before rebalancing anything. This is because in most cases, workers don't experience destructive failures but instead restart immediately. If the worker rejoins quickly, it will keep the tasks that it owned before and no rebalance is needed. If the worker does not rejoin quickly enough (the duration is specified via `scheduled.rebalance.max.delay.ms`, 5 minutes by default) then the tasks it used to run are reassigned to available workers.

Since Kafka 2.4, the default rebalance protocol is `sessioned`. In terms of rebalancing behavior, it works exactly the same way as `compatible` but it also ensures that inter-worker communications are secured. Like `compatible`, `sessioned` is only active if all workers support it, otherwise it defaults to the common protocol shared by all workers.

The rebalance protocol Connect is using is specified by the `connect.protocol` configuration. Users should keep the default value for the version they use and only consider downgrading to `eager` in case they rely on its specific behavior.

For more details in the history behind each protocol, you can read the respective KIPs. The `compatible` rebalance protocol was introduced by KIP-415. The `sessioned` rebalance protocol was introduced by KIP-507.

Handling Failures in Connect

Now that you understand how Connect manages its state, let's take a look at the most common types of failures and see how to handle them.

In order to build a resilient pipeline it's key to understand how all components in your system handle failures. In this section we'll focus on Connect and how it handles failures and ignore other components (such as the operating system, execution and deployment environment or hardware).

We will cover the following failures:

- Worker Failure
- Connector/Task Failure
- Kafka/External Systems Failure

We will also discuss how you can use Dead Letter Queues to deal with unprocessable records.

Worker Failure

In distributed mode, Connect can run across multiple workers and it is recommended for users to use at least 2 workers to be resilient to a single worker failure.

For example, if we have 3 workers that are running 2 Connectors (C1 and C2), the different tasks could be spread like in Figure 3-7.

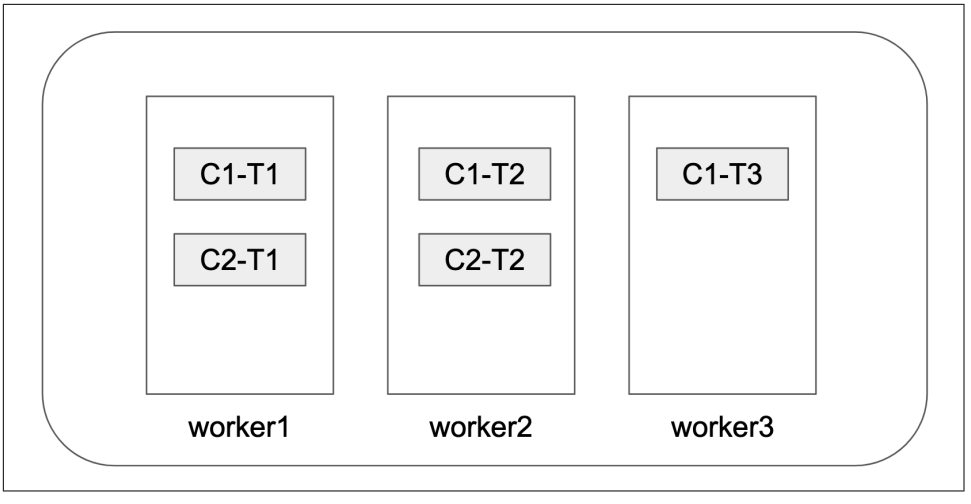


Figure 3-7. Example of a Connect cluster with 3 workers. Connector C1 has three tasks (T1, T2, T3) and C2 has two tasks (T1 and T2).

In this case, if worker2 is taken offline, either because it crashed or for maintenance, Kafka will not receive its heartbeat anymore and after a short interval, it will automatically kick worker2 out of the group. This will force Connect to rebalance all running tasks onto the remaining workers.

After the rebalance, the task assignment may look like Figure 3-8.

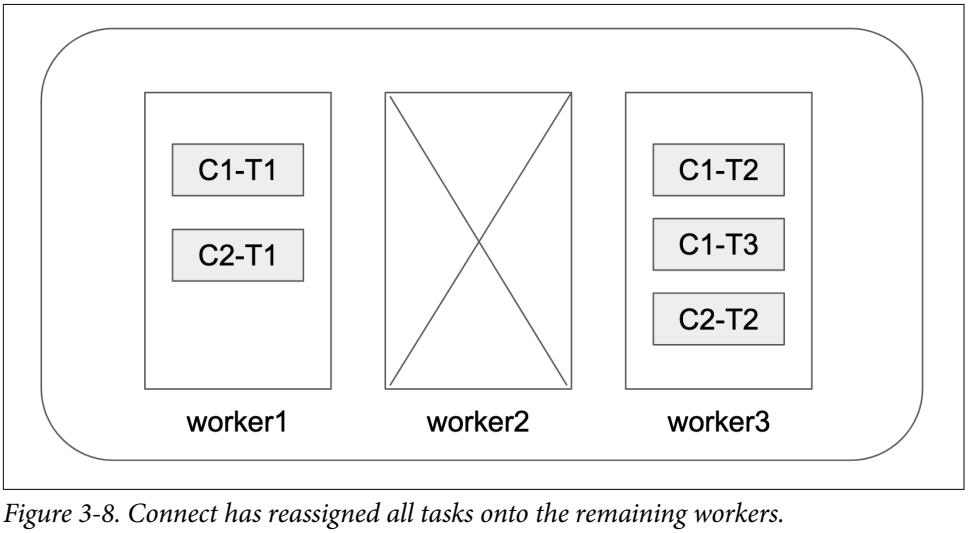


Figure 3-8. Connect has reassigned all tasks onto the remaining workers.

While the rebalance is happening, the tasks that were on worker2 are not run. This mechanism triggers and completes within a few minutes. It depends mostly on the following configurations:

`session.timeout.ms`
: The maximum duration between 2 consecutive heartbeats from workers

`rebalance.timeout.ms`
: The maximum duration workers can take to rejoin the group when a rebalance happens

`scheduled.rebalanc e.max.delay.ms`
: The maximum delay to schedule a rebalance

When a worker is not stopped cleanly, it's possible it did not commit offsets for all records it was processing. So upon restarting, some tasks may reprocess some records. We will discuss this problem and how it affects delivery semantics later in this chapter.

So in order for Connect to handle worker failures, you need to make sure you have enough capacity to accommodate tasks that were on these workers. Connect has no mechanism to limit the number of tasks that can be assigned to a worker during a rebalance. If a worker is assigned too many tasks, its performance will degrade and eventually tasks won't make any progress. At the minimum, you should at all times have enough capacity to handle a single worker being down to reliably handle rolling restarts of the workers.

Connector/Task Failure

Another common type of failure is a crash of one of the connectors or one of its tasks. Until now, we've simplified what happens exactly when Connect runs a connector. In reality, it has to run 1 instance of the connector and 1 or more instances of the task. Connect tracks the health of both and associates them with a state which can be:

`UNASSIGNED`
: A connector or task has not yet been assigned to a worker.

`RUNNING`
: A connector or task is correctly running on a worker.

`PAUSED`
: A connector or task has been stopped by a user via the REST API.

`FAILED`
: A connector or task has encountered an error and crashed.

`DESTROYED`
: A connector or task has just been deleted by a user via the REST API and is shutting down. This state is never exposed to the end user.

`RESTARTING`
: A connector or task has just been restarted by a user via the REST API after it was paused.

The state of connectors and tasks can be retrieved via the REST API. Figure 3-9 depicts the possible transitions between the different states.

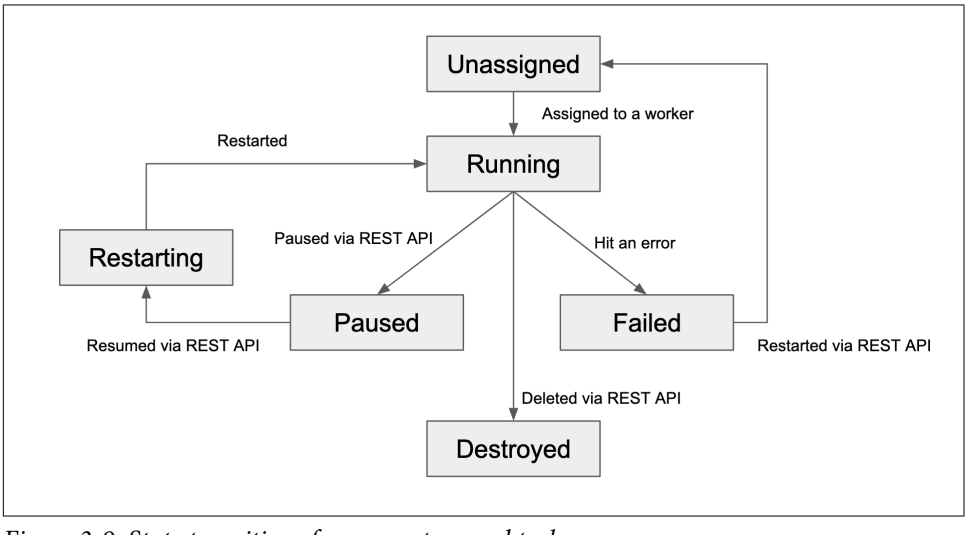


Figure 3-9. State transitions for connectors and tasks.

Connect emits detailed metrics tracking the time spent in each state by each connector. See Chapter 11 for details on how to retrieve and monitor metrics.

The connector class that implements the `Connector` interface is used by the runtime to configure and spawn tasks. In case any of its methods fail and throw an `Exception`, the Connect runtime will retry calling it automatically after a short backoff interval. Some connectors may perform extra logic such as connecting to their target system to discover resources, so this retry mechanism permits handling connectivity issues. While this happens, the connector will be in `RUNNING` state but none of the tasks will be created. One exception is the `start()` method which, contrary to the other methods, immediately puts the connector in the `FAILED` state if it throws an exception.

Each task can also encounter an error. By default Connect lets the task crash, marks it as `FAILED` and it does not attempt to restart it automatically. Connect emits metrics for the state of tasks which administrators have to monitor to quickly identify failures. A task failure does not trigger a rebalance.

In case of a one-off failure, administrators can restart `FAILED` tasks via the REST API. The REST API can also be used to retrieve the Exception that crashed the task and its stack trace. In case of a systematic failure, for example a record that is impossible to process, Connect offers the possibility to skip it and optionally emit a detailed log message instead of failing the task. This can be configured per connector using the `errors.tolerance` configuration.

Kafka/External Systems Failure

As Connect flows data between Kafka and external systems, failures in either can impact Connect.

As detailed in Chapter 2, Kafka can be configured to be a very resilient system. For production use cases, Kafka clusters must have multiple brokers and be configured to offer maximum availability. In addition, Connect must be configured to create its topics with multiple replicas so it's not negatively impacted by the failure of a single broker. This includes topics that are either the source or sink for connectors, the internal Connect topics and the `__consumer_offsets` topic. In this case, Connect will automatically reconnect to Kafka and it will keep running.

On the other hand, a failure on the external system has to be handled by the connector. Depending on the system and the implementation of the connector, it may be handled automatically or it may crash tasks and require manual intervention to recover.

Before building a pipeline, it's important to read the connector documentation and understand the failure modes of the external system to gauge the resiliency of a Connect pipeline. Keep in mind that sometimes there are multiple community implementations for the same connectors and you need to pick the one that satisfies your needs. Then, you need to perform resiliency testing to validate if the connector provides the required resiliency for your use cases. Finally it's important to monitor the appropriate metrics and logs from both the external system and the connector.

Dead Letter Queues

When dealing with an unprocessable record, for sink connectors, Connect can also use a dead letter queue. A dead letter queue, often abbreviated DLQ, is a concept from traditional messaging systems, it is basically a place to store records that can't be processed or delivered instead of simply ignoring them. In Connect, the dead letter queue is a topic (specified via `errors.deadletterqueue.topic.name` in the

connector configuration) where unprocessable records are written. Connect however does not provide a similar mechanism for source connectors because it can't convert the record from the external system into a Kafka record.

Support for dead letter queues for sink connectors was added in Kafka 2.6 via KIP-610.

Let's look at an example of using a dead letter queue. When running the S3 sink connector, the Connect runtime is reading records from a Kafka topic before passing them to the connector. As the topic is expected to contain Avro records, we configure the connector with an Avro converter. However, if a single record in the topic is not in the Avro format, for example an application emitted a JSON record, the connector will not be able to handle this record. Instead of failing the connector or losing this record, Connect can forward it to a dead letter queue and keep processing the other records in the topic. The connector configuration would contain the following settings:

```
{
      "connector.class":  "io.confluent.connect.s3.S3SinkConnector",
      "value.converter":  "io.confluent.connect.avro.AvroConverter",
      "errors.tolerance": "all",
      "errors.deadletterqueue.topic.name": "my-dlq"
}
```

This allows the contents of the dead letter queue topic to be processed by another mechanism, for example another connector or a consumer application.

Figure 3-10 shows an example of using a dead letter queue.

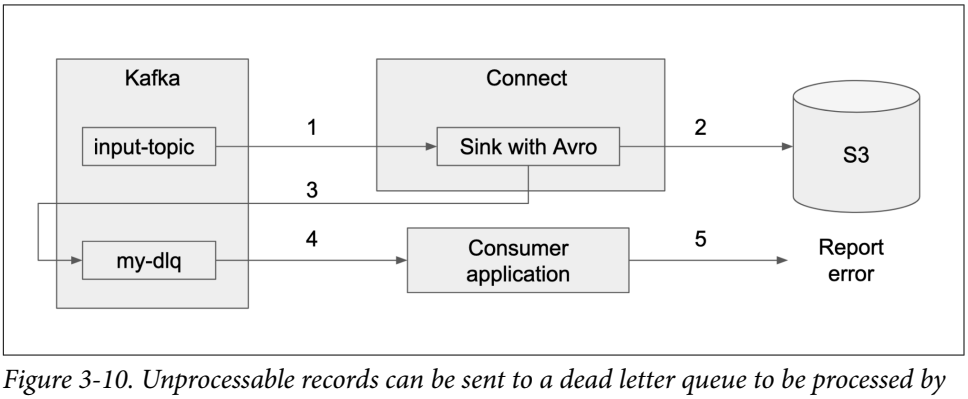


Figure 3-10. Unprocessable records can be sent to a dead letter queue to be processed by another mechanism.

The flow starts with the S3 sink connector configured with Avro receiving records from the input topic (1). Avro records are correctly processed and sent to S3 (2). In case a record can't be processed, it is sent to the dead letter queue configured for the connector (3). In this example, another application receives records from the dead letter queue (4), processes them and reports errors (5).

Understanding Delivery Semantics

Delivery semantics defines the type of guarantees that are made when sending a message. It can be one of these 3 types:

At most once
: When sending a message, it will either arrive or not on the receiver. Messages that don't arrive on the receiver are lost.

At least once
: When sending a message, it will always arrive on the receiver but it may arrive multiple times. Extra copies of a message are called duplicates.

Exactly once
: When sending a message, it will arrive once and only once on the receiver.

Achieving consistent exactly-once semantics is one of the hardest problems in computer science. In some cases, it might not be possible to achieve exactly-once semantics between Connect and an external system. Depending on the use case, relying on at least once semantics with some deduplication, or gracefully handling duplicates in the downstream system may be an acceptable compromise.

The exact delivery semantics that a Connect pipeline provides depends on several aspects, including:

- The connector that is being used and its configuration
- The configuration of the runtime and the way it's set up to handle errors.

Also because a pipeline is composed of multiple steps, the delivery semantic is typically not of the whole pipeline but often per message or batch of messages. This is because each step, due to its configuration, may drop records and result in at most once semantics or instead retry on failure and result in at least once semantics.

Let's look at each type of connector and see how to understand the semantics that can be provided.

Sink Connectors

As a quick recap, these are the steps that constitute a sink pipeline:

1. The runtime consumes records from the Kafka topic
2. Records are passed to the configured converter
3. Records are passed to the configured transformations
4. Records are passed to the sink connector that writes then to the sink system.

Figure 3-11 shows this flow.

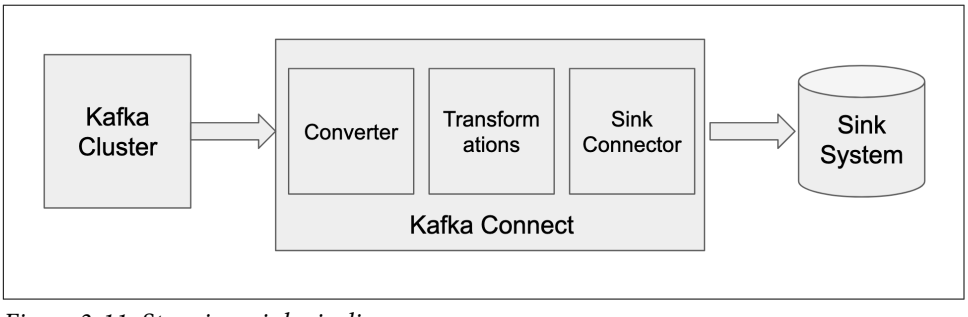


Figure 3-11. Steps in a sink pipeline.

In step 1, each time a task starts up it will restart from the last committed offset. So depending when offsets are committed, a task could re-consume or skip some records. Offsets are eligible to be committed once the connector `put()` method returns. So for example, if a connector sends new records synchronously to the target system and it crashes before returning, the runtime won't have committed the offsets and when it restarts it will re-consume these records. In this case, you have at least once semantics.

On the other hand, if a connector sends records asynchronously to the target system, there's a chance the runtime might commit those records offsets before they are sent. In this case if the send then fails, these records will effectively be skipped and you have at most once semantics.

In step 2 and 3, the main factor deciding the semantics is whether the connector is using a dead letter queue or not. In case it is not, failures will be ignored (or crash tasks) and records may be lost (at most once), based on `errors.tolerance`. With a dead letter queue, if there is a failure, the runtime ensures no records are lost and it automatically forwards the affected records to the dead letter queue. In case there are no failures, these steps effectively provide exactly once semantics.

In the last step, the runtime passes the records to the sink task via the `put()` method and the task is responsible for exporting the records to the target system. Any

failures at this step have to be handled explicitly by the connector and if an error is thrown, the runtime will either skip the records or fail the task based on the value of `errors.tolerance`. In case the connector is configured to not fail, connectors can explicitly forward records in flight to the dead letter queue by calling `report()` on the `ErrantRecordReporter` instance of the task otherwise they will be skipped.

Several of these steps may create duplicate records but in practice, it's still often possible to achieve exactly once delivery semantics. Because records in Kafka are immutable, if the same records are processed twice, the connector can emit the exact same records to the target system. In systems that offer *idempotent* writes, for example by storing records based on their key, they can remove duplicated records and only keep a single copy, effectively achieving exactly once delivery semantics.

To summarize, if you want to avoid losing any records and maximize the availability of a sink pipeline, you should use a dead letter queue. Also while some steps may cause duplicates, some external systems are able to handle them and effectively provide exactly once semantics end to end for sink pipelines.

Source Connectors

As a quick recap, these are the steps that constitute a source pipeline as shown in Figure 3-12:

1. The connector consumes records from the external system
2. Records are passed to the configured transformations
3. Records are passed to the configured converter
4. Records are passed to the runtime that produces them to a Kafka topic

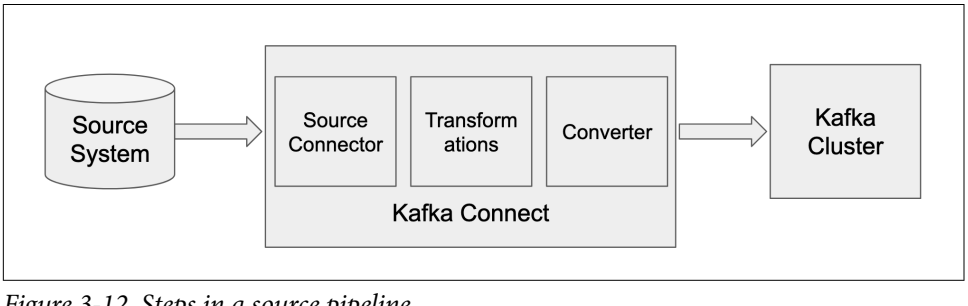


Figure 3-12. Steps in a source pipeline.

Similarly to the consumer fetching records in a sink pipeline, in a source pipeline the connector has to decide which data to retrieve from the external system. Not all external systems have a mechanism like offsets in Kafka that enable them to directly identify a record. For that reason, source connectors can associate an arbitrary mapping of keys to values, the `sourceOffset` field in `SourceRecord` objects, to express

their current position. This arbitrary object can be retrieved by tasks whenever needed. It is the responsibility of the connector to ensure this object contains the appropriate information to correctly retrieve records from the external system. The Connect runtime will automatically store this object in the offsets topic and it also calls `commit()` on tasks in case they want to store it themselves in the target system. This is always done after producing records to Kafka, so it's possible for a worker to successfully produce records to Kafka and fail before it's able to commit their offsets. So depending how the connector works, this step may cause some record reprocessing resulting in at least once semantics.

In case there are any errors in the transformations or converter steps, based on `errors.tolerance`, the task will be either marked as `FAILED` or the failing record will be skipped. Source tasks can't rely on dead letter queues, so this effectively makes these steps provide at most once semantics in case of errors.

Finally the record is pushed to Kafka via a producer from the runtime. As of Kafka 3.1.0, by default, producers are configured to offer at least once but this could be overridden in the connector configuration.

Overall, the achieved semantics of source pipelines are hard to summarize and depend very much on the connector implementation.

Support for exactly once delivery semantics for source connectors is being added to Kafka via KIP-618.

Summary

In this chapter, we looked at the different aspects that need to be taken into consideration in order to build resilient data pipelines with Connect.

We first looked at selecting the right connectors from the hundreds of connectors built by the Kafka community. You need to consider the flow direction, whether it fulfills your feature requirements and if it comes with an appropriate level of support.

Then we focused on data models and formats and the options you have for mapping data between systems. Whatever choice you make you need to understand the structure of your data at each stage in the pipeline and make conscious transformation and formatting decisions. We also highlighted the benefits of using schemas and a schema registry to properly define and enforce the structure of the data.

We then examined the challenges in handling the many kinds of failures that can arise from the crash of a full worker down to errors in a single task. Although Connect is generally considered to be resilient, it cannot recover from all failures,

so if for example a worker or task goes down it is likely you would need to step in. You should understand the levers we discussed so you know when to use these in response to failures.

Finally we detailed how all the decisions taken regarding data models, error handling, runtime and connector configurations directly impact the delivery semantics that can be achieved by pipelines with Connect. For sink pipelines, dead letter queues are a powerful feature to avoid losing data, and achieving exactly-once semantics can be relatively straight forward with many downstream systems. For source pipelines achieving exactly-once is much more tricky and really depends on the connector and the external system.

CHAPTER 4

Deploying and Operating Kafka Connect Clusters

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the eighth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *KafkaConnectBook@gmail.com*.

In this chapter, we will focus on how to deploy and operate Kafka Connect clusters. Starting with deployment we will look at how to build a Connect environment, customize it using connector and worker plugins and determine the network and permissions requirements. We will also discuss how to size a cluster efficiently to handle the workload for your use cases. Then moving into operation, we will look at the most common operations that administrators perform on a running Connect cluster, such as adding and removing workers, applying upgrades, restarting failed tasks and resetting offsets. Finally we will give an overview of the Connect REST API and explain how to use each of the available endpoints to manage and monitor clusters.

After reading this chapter you will be able to deploy and maintain a production Connect cluster.

Preparing the Kafka Connect environment

In Chapter 3 we talked about the difference between standalone and distributed mode when you deploy Connect. Distributed mode comes with more operating steps but is recommended for production deployments due to the added resiliency it provides. For this chapter we will assume you are deploying Connect in distributed mode.

If you do choose to use standalone mode keep the following in mind as you read this chapter:

- In standalone mode, you only deploy a single, independent worker and cannot add additional workers later
- In standalone mode, workers store state in the file system, keep this in mind when considering sizing and capacity planning

Before deciding how to configure your Connect cluster, you must first choose the environment to deploy it in. Like Kafka, Connect is a Java based project. This means as long as you have the Connect libraries you can run it on any Java environment. Make sure you use the Java version that is recommended on the Kafka website.

Connect workers should be deployed, configured and upgraded independently of the brokers in the related Kafka cluster. Although workers and brokers require a similar environment, they have different lifecycles and need to be scaled independently. That being said, the most common place to deploy Connect is next to the Kafka cluster that it will be interacting with. Doing so provides these benefits:

- The processes for operating Kafka and Connect are similar. They both require a Java environment, they are distributed workloads that can be scaled and monitored in a similar way. This means whether you deploy them on bare metal or on a platform like Kubernetes, it's easier to use the same infrastructure for both, rather than having to adopt or create tools for multiple environments.
- Since Connect uses Kafka for storing state and coordinating between workers, if the clusters are close to each other, the latency for this kind of traffic is lower.

There are some scenarios where it is preferable to run Connect alongside the external system, rather than alongside the Kafka cluster. This depends on the external system and the connector. For example the message queuing system IBM MQ includes two connection options; bindings mode and client mode. Both are supported by the IBM MQ connectors, but for MQ installations that only allow bindings connections, both the Connect cluster and connectors must be running in the same environment as the IBM MQ system for the connection to be successful.

Let's now look at the steps you need to take to prepare your chosen environment for your Connect cluster.

Building a Connect environment

You should have already used the Connect scripts and JAR files to start a Connect cluster on your local machine. You will need all of these available in whatever environment you choose for your Connect cluster. As a reminder you should have:

- The Connect JAR files from the `libs` folder of the Kafka distribution.
- The `connect-distributed.sh` script from the `bin` folder of the Kafka distribution.
- The `kafka-run-class.sh` script from the `bin` folder of the Kafka distribution (this is invoked by the `connect-distributed.sh` script).
- A configuration file for each Connect worker using the properties format.
- The JAR files for the connector and worker plugins you will use (see the next section for more on worker plugins).

A *properties file* is a plaintext file that maps keys to values. Each line is a single mapping with the following syntax: `key=value`.

For example, the following file has the keys `group.id` and `bootstrap.servers` set to `test-group` and `localhost:9092` respectively:

```
group.id=test-group
bootstrap.servers=localhost:9092
```

Many developers choose to deploy Connect workers as containerized applications on platforms like Kubernetes. If you want to run your Connect workers as containers you can either create the container image yourself, use one from the community, or purchase some proprietary software that provides one.

Here is an example of a Dockerfile for a Connect worker using a well-known Java image as the base:

```
FROM adoptopenjdk:latest

COPY /opt/kafka/bin/ /opt/kafka/bin/
COPY /opt/kafka/libs/ /opt/kafka/libs/
COPY /opt/kafka/config/connect-distributed.properties /opt/kafka/config/
COPY /opt/kafka/config/connect-log4j.properties /opt/kafka/config/
RUN mkdir /opt/kafka/logs
COPY my-connector.jar /opt/kafka/plugins/
```

```
WORKDIR /opt/kafka

EXPOSE 8083

ENTRYPOINT ["./bin/connect-distributed.sh", "config/connect-distributed.proper-
ties"]
```

This example Dockerfile assumes that your Kafka distribution is downloaded to `/opt/kafka` and the `connect-distributed.properties` file has set the `plugin.path` to be `/opt/kafka/plugins/`.

There are plenty of open source projects that also provide images for you to use. These are often available on public container registries such as DockerHub and Quay. Two such projects are Strimzi and Debezium.

Strimzi is an open source project that provides tools for deploying Connect clusters onto Kubernetes. The Operator provides a mechanism to let you build a custom Connect Docker image that includes connector plugins you need. In Chapter 11 we explore more options for deploying Connect on Kubernetes and see Strimzi in action.

Debezium provides pre-built Docker images on DockerHub that already include all of the Debezium Connect plugins. Finally if you need a Connect container image that is fully supported you can look to products such as Red Hat AMQ Streams, Confluent Platform or IBM Event Streams.

Now let's look at how you can add plugins for your use case to your Connect installation.

Installing plugins

You must make sure that you add all plugins to your environment before starting your Connect workers. Workers load all plugins at start up and will not notice plugins that are added at runtime. There are some default plugins that are already added to the classpath of Connect. These can be found in the `libs` directory of the Kafka distribution. To add custom plugins we would recommend using the `plugin.path` worker configuration. Although you can place plugins into the `libs` directory you are likely to hit classpath errors.

The `plugin.paths` configuration expects a comma-separated list of directories. There is no limit to the number of directories you can include. The files for a plugin must include all dependencies. This means the structure of your `plugin.path` directories depends on the way your plugin is packaged. Either you will have multiple JAR files that together provide the plugin and all the dependencies, or a single "Uber" JAR file that contains the plugin and dependencies all packaged together.

When Connect loads a particular plugin, it expects that any JAR files at the top level of the `plugin.path` directory are Uber JARs, and that any plugins that require multiple JAR files are in directories. For example consider the following structure:

```
+-- custom-plugin-1.jar      ❶
+-- custom-plugin-2          ❷
|     +-- custom-plugin-2-lib1.jar
|     +-- custom-plugin-2-lib2.jar
```

❶ A single Uber JAR file containing the plugin and all its dependencies

❷ A directory containing a set of JAR files that include the JAR file for the plugin and the JAR files for all its dependencies

If you didn't have the `custom-plugin-2` directory and instead placed those files next to `custom-plugin-1.jar`, when you tried to use that plugin you would see `ClassNotFound` exceptions for any classes in the `custom-plugin-2-lib2.jar` file.

Once you have decided which plugins to add make sure you can see them being loaded correctly in the startup logs for Connect:

```
INFO Added plugin 'org.apache.kafka.connect.converters.ByteArrayConverter'
INFO Added plugin 'org.apache.kafka.connect.file.FileStreamSourceConnector'
INFO Added plugin 'org.apache.kafka.connect.transforms.TimestampRouter'
```

Once you've decided where to deploy Connect and added your plugins, you also need to consider the networking and permission requirements.

Networking and permissions

To deploy Connect into a production environment you will need to make sure the workers are granted the connect permissions to receive and send requests. This should include restricting which endpoints are accessible from which components. This is important to reduce the opportunity for bad actors to disrupt your pipeline. There are four kinds of network traffic you should consider that are shown in Figure 4-1.

These are:

1. Connections in-between Connect workers
2. Connections between Connect workers and Kafka
3. Connections between Connect workers and external systems
4. Connections from REST clients to the administrative endpoint on workers

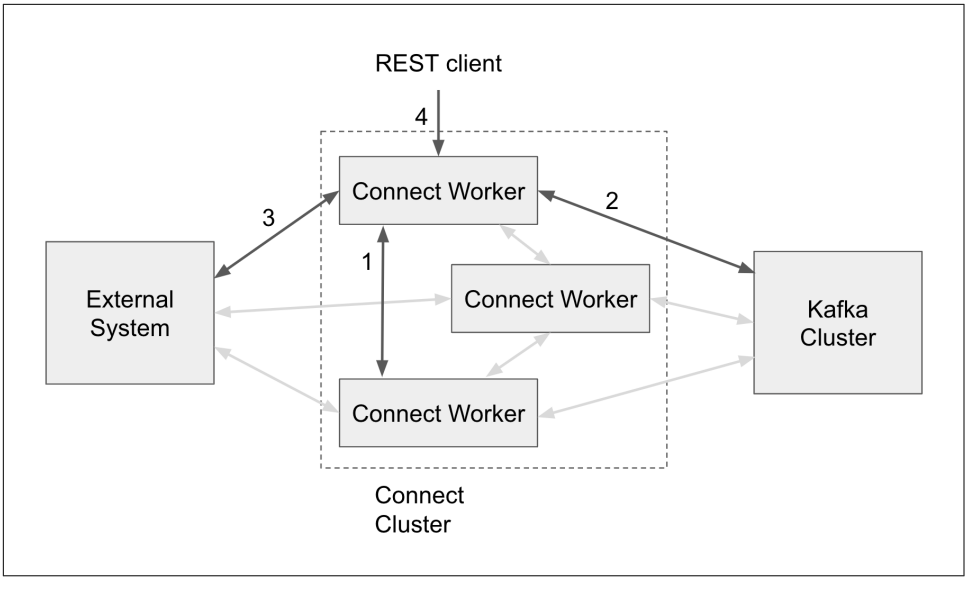


Figure 4-1. Network connections in a Connect pipeline

Each connection requires different permission and network considerations. We will go through each in turn.

Let's first look at connections between workers (connection 1 in Figure 4-1). Connect workers communicate directly with each other if a task needs to be reconfigured. This could occur because a connector has been reconfigured manually using the Connect REST API or because a connector has automatically detected a reconfiguration is required. As an operator of Connect you must make sure that the Connect workers can talk to each other via the configured REST API port.

The second connection type is between the Connect workers and Kafka (connection 2 in Figure 4-1). In a production system it is likely that the Kafka cluster that Connect uses will require clients to authenticate and will restrict which operations a specific user can perform. If your Kafka cluster is secured in this way you must make sure the Connect user is granted all the permissions it needs. The operations that Connect needs to perform fall into two categories, operations required by a Connect worker and operations for a specific connector.

To act as part of the Connect cluster successfully, Connect workers must have access to the correct topics and consumer groups so they can coordinate. Specifically they must have:

- Read permission for the group listed as the `group.id` in the worker configuration
- Create and write permissions for the configuration topic

- Create and write permissions for the offset topic
- Create and write permissions for the status topic

You can pre-create the configuration, offset and status topics and if you do that Connect will only need write permissions, not create.

Depending on which connectors you want to run you will also need:

- Write permissions for any topics used by source connectors
- Read permissions for any topics used by sink connectors
- Read permissions for the group used by the sink connectors (named `connect-<connector name>`)
- Write permissions for the external systems used by sink connectors
- Read permissions for the external systems used by source connectors

In addition, if you aren't pre-creating the topics for source connectors you will also need permissions to create those topics.

The third connection type in Figure 4-1 is between connectors and the external system. Exactly how you configure this depends on the connector, but most connectors provide configuration settings for this. Both for connections to the external system and connections to Kafka consider using configuration providers that can help pull credentials from a variety of places and make it safer to configure these credentials.

Finally, you will need to configure your system so you can perform administrative tasks such as starting and stopping connectors and tasks (connection 4 in Figure 4-1). This is normally done through the REST API so consider whether this endpoint can be exposed to all users, or if it also needs to be secured in some way.

In Chapter 9 we go into more details on securing Connect, including providing user credentials to workers and connectors and how to secure the REST API endpoint.

Worker Plugins

Connectors, converters, transformations and predicates are all commonly referred to as *Connector Plugins* because they control the behavior of connectors. Connect also supports a few other plugin types that administrators can use to customize the Connect runtime. These other plugins are called *Worker Plugins*. There are two types of worker plugins:

- Configuration providers
- REST extensions

Let's look at each of the plugins and what they do.

Configuration Providers

Configuration providers enable Connect to dynamically retrieve configurations at runtime instead of hard coding values. They are classes that implement the `ConfigProvider` interface from the Connect API. You can specify them when configuring the Connect runtime and for individual connectors. The configuration provider mechanism was introduced in Kafka 2.0.0 via KIP-297.

Configuration providers are useful when handling sensitive values, like passwords or security keys, as they let you put a placeholder value in the configuration instead of the actual value. When the Connect runtime starts, or when a connector is started, it will automatically replace the placeholders with the values generated by the configured configuration providers. This is safer because when a user accessed the connector configuration via the REST API, the password is not exposed and instead it shows the placeholder. Otherwise, by default, anybody with access to the REST API can see the full configuration of all connectors. Another benefit of configuration providers is that you can have one configuration template that can be applied to multiple environments. Then at runtime each environment will use its own specific values.

In order to be used, configuration providers have to be enabled in the runtime configuration and present in either the classpath or in `plugin.path`.

Kafka comes with two built-in configuration providers:

- `FileConfigProvider`: This is for retrieving configuration values from a properties file on the worker file system.
- `DirectoryConfigProvider`: This is for retrieving configuration values from a directory containing files. For example, this works well with Kubernetes secrets where a single file cannot be provided. This provider was added via KIP-632 in Kafka 2.7.0.

Let's see an example of how to use a configuration provider in practice. We want to run the MirrorMaker source connector and configure it to connect to the source Kafka cluster over TLS. To do so we need to specify values for a few SSL settings including passwords for our truststore. Let's look at how we can use `FileConfigProvider` to avoid putting the password in cleartext in our connector creation request.

First the Connect runtime needs to have `FileConfigProvider` enabled. You do this by adding the following to `connect-distributed.properties`:

```
config.providers=file
config.providers.file.class=org.apache.kafka.common.config.provider.FileConfig-
Provider
```

The first line gives nicknames to the providers we are going to use. Here it says we're defining a single configuration provider that will be called `file`. On the second line, we specify which specific implementation we want to use for the `file` nickname.

Then the runtime needs to have a properties file on its file system that contains the password we want to use. For example, we can create a file, `/tmp/ssl.properties`, that contains the following:

```
password=secret-password
```

Once the runtime is running with the above configurations, we can create a connector that uses the configuration provider to retrieve the password value. In the connector JSON configuration instead of using:

```
"ssl.truststore.password": "secret-password"
```

We can have:

```
"ssl.truststore.password": "${file:/tmp/ssl.properties:password}"
```

In this example, `file` is the nickname of the configuration provider set in the runtime, `/tmp/ssl.properties` is the file on the runtime that contains the mappings and `password` is the key we want to get the value from.

While it appears we just moved the password from one place to another, a file on the worker's file system can easily be secured. There are also other configuration providers that support supplying the credentials from for example an identity and secret management system, such as Vault.

Another category of worker plugins is REST extensions.

REST Extensions

REST extensions allow you to customize the Connect REST API. For example, you can use them to add user authentication, fine grained authorizations or validation logic for connector changes via the REST API. A REST extension is a class that implements the `ConnectRestExtension` interface from the Connect API. Like configuration providers, in order to use REST extensions, they have to be enabled in the runtime configuration via `rest.extension.classes` as a comma separated list, and present in either the classpath or in `plugin.path`.

Using REST extensions you can register JAX-RS plugins and make use of any functionality they provide. JAX-RS is a Jakarta EE API for defining web services.

By default, Kafka Connect includes a single built-in REST extension, `BasicAuthSecurityRestExtension` that lets you secure the REST API with HTTP Basic Authentication.

Let's look at an example using `BasicAuthSecurityRestExtension` to enforce authentication to access the REST API. First you need to enable the plugin in the Connect runtime configuration by adding the following to `connect-distributed.properties`:

```
rest.extension.classes=org.apache.kafka.connect.rest.basic.auth.extension.BasicAuthSecurityRestExtension
```

Then we need a Java Authentication and Authorization Service (JAAS) file that is used to specify the mechanism to verify credentials. A JAAS file contains a `LoginModule` which is a generic way to provide authentication in the Java Virtual Machine (JVM). In production environments, a `LoginModule` may interact with an external authentication system such as Kerberos or LDAP. For this scenario, we can use `PropertyFileLoginModule` which is an example built-in `LoginModule` that stores credentials in a properties file.

We need to create a new file, `connect-jaas.conf`, with the following content:

```
KafkaConnect {
  org.apache.kafka.connect.rest.basic.auth.extension.PropertyFileLoginModule
required file="/tmp/credentials.properties";
};
```

We also need the file, `/tmp/credentials.properties`, that stores the credentials for accessing the REST API with the following content:

```
mickael=p4ss
kate=w0rd
```

The syntax of this file is `<username>=<password>`. The example above creates two credentials, one called `mickael` with `p4ss` as the password and one called `kate` with `w0rd` as the password.

Before starting Connect, we need to tell the JVM to load our JAAS file. We can do that by setting `KAFKA_OPTS` in the environment where you will run Connect:

```
$ export KAFKA_OPTS="-Djava.security.auth.login.config=tmp/connect-jaas.conf"
```

`KAFKA_OPTS` is a specific environment variable that is automatically picked up by all Kafka tools. We can now start Connect. If we try to access the REST API without credentials, the request is rejected:

```
$ curl http://localhost:8083/
User cannot access the resource.
```

To access it, we need to provide one of the credentials we created. Since we are using the Basic authentication REST extension, we pass the credentials by setting the `Authorization` header in the request. Its value must be set to `Basic <credentials>` where `<credentials>` is the base64 encoding of `<username>:<password>`. To generate this value we can run:

```
$ echo -n mickael:p4ss | base64
bWlja2FlbDpwNHNz
```

Then when we provide the authorization header we can successfully access the REST API:

```
$ curl -H "Authorization: Basic bWlja2FlbDpwNHNz" http://localhost:8083/
{"version":"3.1.0","commit":"37edeed0777bacb3","kafka_cluster_id":"xbuOyGSLT-
XyVsME9qLDgZg"}
```

Note that this example is not secured because it is using HTTP, so the credentials are sent over the network in cleartext. When using authentication schemes that rely on sending secrets, like Basic Authentication, the REST API should be accessed via HTTPS.

REST extensions, including `BasicAuthSecurityRestExtension`, were added via KIP-285 in Kafka 2.0 in July 2018.

Sizing and Planning capacity

In order to effectively provision and allocate resources for setting up a Connect cluster, it's important to keep in mind how Connect spreads the workload across workers and tasks.

Tasks are the units that perform the work for connectors. Each task is basically composed of a Kafka client, producer for source tasks, consumer for sink tasks, combined with a client for an external system. The Connect runtime dynamically spreads tasks onto the available workers. So to make sure your cluster can support your workload, you need to make sure the workers have enough capacity to run these tasks. A Connect cluster can be scaled in two dimensions. It can be *scaled up* if administrators increase the capacity of the workers. It can also be *scaled out* if administrators increase the number of workers.

In this section we will look at the resources that Connect uses, how to decide the number of tasks and workers you need and how to combine these to inform sizing and planning decisions. We will also discuss running a single Connect cluster versus multiple clusters.

Understanding Connect resources utilization

Kafka Connect does not require a minimum hardware configuration for workers and it will start and run on a wide variety of hardware and virtual environments. In practice however, and especially for production environments, it's useful to understand how Connect uses hardware resources to maximize its capacity and reliability while keeping cost at a minimum.

In distributed mode, Connect does not store any state locally on workers. Instead all data is stored in its three internal topics in Kafka. This means in order to deploy Connect you don't need a lot of storage, just enough to store its logs and debugging artifacts like verbose garbage collection (GC) logs and thread/heap dumps.

For the other main resources, CPU, memory and network, as is the case most of the time with performance and sizing, the optimal configuration depends on your exact use cases. Let's look at the main factors that influence resource usage and performance:

- **Number of connectors and tasks**: Each connector and task uses some resources, so the more you start, the more resources Connect will use.
- **Types of connectors:** Some connectors are more resource intensive than others. It depends mostly on how they interact with the external system.
- **Enabled converters and transformations:** Converters have to constantly serialize and deserialize data as it flows through Connect. Simple converters like `ByteArrayConverter` don't use a lot of resources as they effectively just forward raw bytes. On the other hand, converters that construct complex types, such as `AvroConverter`, use more CPU and memory. Also each transformation you add to your pipelines uses some resources.
- **Compression:** Compressing messages uses CPU but allows you to reduce the network usage. Compression can significantly improve throughput but note that downstream systems will have to decompress data, and also use CPU, in order to use the data.
- **Client configuration:** Many client settings can also impact performance. For example, if you have clients (for both Kafka and the external systems) that use TLS to encrypt data, you will be using extra CPU and memory.

In many scenarios, there can also be an important difference in the resource usage when starting a connector compared to when it's running steadily. At startup, a connector can discover a huge backlog of records to flow through Connect and will start processing them as fast as possible. For example when you are sinking a topic that contains hundreds of GB of data, this can lead to the connector having a much higher throughput than the regular incoming throughput on the topic. Only

once the initial backlog is processed, will Connect's throughput match the incoming throughput of the source.

Finally in terms of capacity planning, you also need to consider the extra load your Connect clusters will add to both your Kafka cluster and external systems. This is especially important when sinking data out of Kafka. Kafka scales horizontally very well and for example if you have a topic with 50 partitions, you can have up to 50 Connect sink tasks exporting data from it. This means all 50 tasks will create a client and connect to the external system. For some systems, this can very quickly lead to a lot of contention in case all tasks end up accessing the same entities in that system, such as a table in a database.

How many workers and tasks?

In order to start sizing a use case, you need to estimate how much network throughput, in bytes per second, is required. For sink connectors this is easier because the required throughput is the combined incoming byte rate of the input topics. For source connectors it really depends on the external system. Most systems expose metrics describing the rate of change on their resources, otherwise you can use a test system to get a rough idea. If you do this, make sure you are using a system that is as close as possible to what you intend to deploy in production.

Once you have estimated network throughput for all the use cases you want to run on Connect, we can start thinking about the number of workers to have in the cluster. Regardless of capacity, in order to be resilient to worker failures and permit basic operations, you should always have at least two workers. Then you need to compare your network throughput estimate with the worker capacity.

For example, if you estimate your use cases require 300MB/s and each worker has 1Gb/s network cards, hence 128MB/s, you need a minimum of three workers. In practice, you should provision enough capacity to sustain this workload with a worker down or failing. This means you should either provision four workers or scale up and use workers with more network capacity. For example if you upgraded to 2Gb/s (256MB/s) network cards, then three workers would be enough.

It's common for use cases to evolve and new use cases to appear so you need to regularly review the capacity and number of workers in your clusters. You need to monitor their resource usage via metrics. This enables you to notice when a worker approaches its capacity and either increase its resources or add new workers to share the load. Connect also emits a lot of metrics that enable you to gauge how each connector and task performs. We cover all these metrics in detail in Chapter 10.

With the worker count and specification sorted, you need to decide how many tasks each connector will run.

Tasks enable splitting the workload into smaller chunks. With many connectors, it's possible to use a single task and have it handle the whole workload. But a single task can only run on a single worker. By increasing the number of tasks, we make the workload more granular so it can be spread efficiently across more workers. In addition, with multiple tasks, the failure of a single task limits the scope of the failure to only the resources that task is handling. However more tasks also means more overhead as each task has its own clients, each with their own connections and associated memory usage.

You only control the maximum number of tasks, via `tasks.max`, and connectors are free to start less based on their implementation. In practice the maximum number of tasks you can run on a worker depends on the connector, as different connectors have different requirements.

First, for each connector you are running, you can work out the highest value you can set `tasks.max` to for that connector. For sink connectors, this corresponds to the number of partitions from the input topics, for example when sinking data from 3 topics of 20 partitions each, a connector will only be able to create up to 60 tasks. For source connectors, it depends how the connector and external system work so you should check their documentation.

The actual value you set `tasks.max` will be between 1 and the highest value possible for that connector. You should choose this by considering all the connectors you want to run and making sure that they each get enough resources. A method to estimate a good value for `tasks.max` is to start with all connectors using a small value and monitor closely the CPU, memory and network usage of the worker. If all these resources have spare capacity, you can bump `tasks.max` on one or more connectors and repeat the process.

Single Cluster vs Separate Clusters

When scaling to handle larger or additional workloads, the first option is to add workers to an existing Connect cluster. This is a good starting point and for many types of use cases, centralizing all data pipelines onto a single Kafka Connect cluster is the easiest and best solution. Also if the workload is fairly small, a single cluster with a few workers will easily handle it.

However there are cases when you should consider splitting your workloads across multiple Connect clusters. Let's look at a few of the factors that you need to take into account when making this decision.

Maintainability

It seems pretty obvious that only having a single cluster to maintain is easier. Any processes or tools you use will have to connect and operate on all your clusters. This

includes performing upgrades, applying security patches, maintaining configuration, scaling but also includes all the monitoring via logs and metrics that enable SREs to operate the clusters.

Simplifying and reducing the number of maintenance operations also lowers the cost of ownership. Less hours are spent keeping clusters running and can be directed at improving or building new cases.

Isolation

By having workloads on different clusters, they are better isolated because they can't directly interact with each other. This was very important when Connect only had the `eager` rebalance protocol which triggers a rebalance of all connectors and tasks every time a connector is reconfigured. In clusters using `eager`, a single misbehaving connector can pretty much stop all workloads constantly. Now with `sessioned`, or even `compatible`, reconfigurations only affect the connector being reconfigured, so workloads are a bit more isolated by default.

If you are running a set of connectors belonging to different teams, it may be preferable to run them on different clusters. Connect does not have a mechanism for allocating resources to a connector so all connectors effectively contend to use them. In most cases, the operating system gives a fair share to each task (each task is executed in a thread) but depending on the configuration and the connectors used, that may not always be true.

The blast radius in a system indicates the number of affected workloads when a failure happens in that system. If you have a single Connect cluster and it goes offline, all your workloads are affected. By having workloads spread across several environments we can reduce the blast radius in case of failures.

Security

Another use case for having separate clusters is if a workload needs a specific security configuration because, for example, it's handling regulated data like medical or financial records. Such use cases usually require you to secure the REST API, access to the workers and generally introduce some sort of auditing capabilities. If another team using this same cluster does not need these security measures, it may become an annoyance for them to follow these rules.

Use case optimization

Finally some use cases benefit from a very specific configuration. This could be the case for performance, regulations or just functionality reasons. By default, with `connector.client.config.override.policy` set to `all`, the runtime enables connectors to provide specific configurations to optimize their behavior so it's possible to run

connectors with different settings each. But this often makes tasks from different connectors behave very differently. While Connect automatically spreads tasks onto workers, it assumes all tasks are similar. If this is not the case, it can lead to very uneven loads on the workers. So for connectors requiring bespoke configurations, it may be preferable to run them in a purpose built cluster.

Now we have discussed the options for deploying a Connect cluster, let's look at what you need to do once it is running.

Operating Connect clusters

To run a Connect cluster in production you need to consider not only how to deploy it, but also how to handle its ongoing operation. Although Connect will move tasks between workers it does not include any logic to automatically scale workers or restart failed connectors. Instead you will need to create these processes for these operational requirements as part of your system.

Since Connect is a very popular technology there are plenty of tools available so you needn't design your system from scratch. Rather than covering specific tools here, we will step you through the different administrative steps you need to know. That way you can understand what steps you need to automate, or what actions a particular tool is taking in your cluster to achieve a specific goal.

To automate these processes yourself you can use the Connect REST API or a dedicated Command Line Interface (CLI) tool like kcctl. Alternatively, if you are deploying on Kubernetes you can make use of an operator to do the hard work for you. We will discuss Kubernetes operators for Connect more in Chapter 11.

Let's look at the following actions that you might need to perform:

- Adding workers
- Removing workers
- Upgrading and applying maintenance to workers
- Restarting failed tasks and connectors
- Resetting offsets of connectors

Adding Workers

Whether you use multiple clusters or a single one, one of the first tasks administrators perform is to start multiple workers and have them work together. You may want multiple workers to get better resiliency, or to increase the capacity of the cluster.

For a worker to join a Connect cluster, it needs to be able to connect to

- The same Kafka cluster as the other workers
- All of the other workers on the port they expose via `admin.listeners` (or if unset via `listeners`).
- All the external systems you want to connect to

Once you have a suitable environment, the runtime on this new worker needs to have some specific settings that match the other existing workers' runtimes:

- `group.id`
- `config.storage.topic`
- `offset.storage.topic`
- `status.storage.topic`
- `bootstrap.servers` and any associated SSL and SASL connection settings

You should also ensure that any settings that affect the behavior of the runtime, such as timeouts and security, are configured the same way on all workers. Otherwise this could cause the whole cluster to behave erratically or even completely fail.

Once you have built a matching configuration, you can simply start the Connect runtime on the new worker:

```
$ ./bin/connect-distributed.sh ./config/connect-distributed.properties
```

If you have more tasks than running workers, within a few minutes some tasks should be assigned to this worker. Also once it is started, ensure your monitoring is picking it up and you are getting both metrics and logs.

Removing Workers

In case the workload decreases, or you move some of it to a separate cluster, you can also remove workers from a cluster. The process is extremely simple as you just need to shut down the Connect runtime on that worker. Connect will automatically rebalance the tasks this worker was running onto other workers.

To stop the Connect process cleanly, you can send it a `SIGQUIT` signal. For example, once on the worker, first find the Connect process:

```
$ ps -ef | grep ConnectDistributed
7152 ttys005  0:54.26 ... org.apache.kafka.connect.cli.ConnectDistributed ./con-
fig/connect-distributed.properties
```

Then shut it down by sending `SIGQUIT` to the process ID, in this example:

```
$ kill -3 7152
```

Connect waits up to `scheduled.rebalance.max.delay.ms` milliseconds, by default 5 minutes, before rebalancing the tasks this worker was running. Once all tasks have been reassigned to the remaining brokers, this operation is complete.

Upgrading and Applying Maintenance to Workers

Once a cluster is running, the proper way for administrators to apply maintenance onto workers is via rolling restarts. A rolling restart means restarting a single worker at a time and each time waiting for it to return before moving onto the next one. This allows you to limit the disruption to a minimum. At the end of a rolling restart, all workers have been restarted once.

Administrators need to follow this process in order to upgrade the Connect runtime, plugin JAR files and any environment dependencies such as the JVM, or the operating system. When upgrading the runtime you should check the notable changes list from the upgrade section on the Kafka website. Kafka tries to maintain compatibility between releases and this section will mention any changes that could affect your workload as well as new features. For each connector, you should check its documentation to verify whether new versions are compatible.

Before getting started you need to identify the steps to perform while Connect will be stopped in order to minimize the downtime. You also need to estimate how long these steps will take you. If it will take longer than `scheduled.rebalance.max.delay.ms`, then Connect will reassign all tasks currently on this worker onto other workers. Connect will automatically handle it so this is not problematic but it can significantly lengthen the overall duration of the operation as you will need to wait for all tasks to be reassigned before moving onto the next worker. If you need just a bit longer than the default `scheduled.rebalance.max.delay.ms` value, it may be worth slightly increasing this setting to avoid the extra rebalance.

First ensure your Connect cluster is healthy and all tasks are running as expected. In order to do so you can check the tasks metrics as described in Chapter 10 or check the connector status via the REST API. Find all running connectors:

```
$ curl -s http://localhost:8083/connectors | jq
[
  "file-sink"
]
```

Then for each instance, check the connector and all tasks are in the `RUNNING` state:

```
$ curl -s http://localhost:8083/connectors/file-sink/status | jq
{
  "name": "file-sink",
  "connector": {
    "state": "RUNNING",
    "worker_id": "192.168.1.12:8083"
  },
```

```
  "tasks": [
    {
    "id": 0,
    "state": "RUNNING",
    "worker_id": "192.168.1.12:8083"
    }
  ],
  "type": "sink"
}
```

If everything is running, you can shut down the Connect process by sending it a `SIGQUIT` signal like for removing a worker. Then you can perform the maintenance tasks you want. Once done, you need to restart the Connect runtime:

```
$ ./bin/connect-distributed.sh ./config/connect-distributed.properties
```

Before moving onto the next worker, ensure everything is running again.

Restarting failed tasks and connectors

Like in every system, sometimes things can go wrong in Connect. If a task or a connector hits an unrecoverable error, it will be stopped and put in the `FAILED` state. In that case, the administrator is responsible for investigating the failure and restarting any `FAILED` tasks and connectors.

Connect emits metrics to track the state of all components so you should rely on them to identify if any tasks or connectors have failed (We cover this in Chapter 10.). If this happens, you can use the REST API to get detailed status for the relevant connector or tasks:

```
$ curl http://localhost:8083/connectors/file-source/status
{
  "name": "file-source",
  "connector": {
    "state": "RUNNING",
    "worker_id": "192.168.1.12:8083"
  },
  "tasks": [
    {
    "id": 0,
    "state": "FAILED",
    "worker_id": "192.168.1.12:8083",
    "trace": "org.apache.kafka.connect.errors.ConnectException:
java.nio.file.AccessDeniedException: /tmp/source\n\tat
    ...
    java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExe-
cutor.java:635)\n\tat java.base/java.lang.Thread.run(Thread.java:833)\nCaused
by: java.nio.file.AccessDeniedException: /tmp/source\n\tat
    ...
    org.apache.kafka.connect.file.FileStreamSourceTask.poll(FileStreamSource-
Task.java:91)\n\t... 9 more\n"
```

```
    }
  ],
  "type": "source"
}
```

In this example, the task with `id=0` has failed and needs to be restarted. The `trace` field indicates the reason for the failure. In this case, the Connect runtime does not have permissions to access the file (`AccessDeniedException`). In most cases, you first need to address the root cause of the failure, otherwise if you simply restart the task it will immediately hit the same issue and return to the `FAILED` state.

Once we've fixed the permissions issue on our file, you can use the REST API to restart it. Since Kafka 3.0.0, you can directly restart all failed components (connector and tasks) of a connector using the `POST /connectors/{name}/restart` endpoint:

```
$ curl -X POST "http://localhost:8083/connectors/file-source/restart?include-
Tasks=true&onlyFailed=true"
```

This endpoint accepts two parameters:

- `includeTasks`: This defaults to `false`. When enabled, both the connector and its tasks are restarted.
- `onlyFailed`: This also defaults to `false`. When enabled, only components that are in the `FAILED` state are restarted.

Once restarted, you need to check again the detailed status of the connector to ensure everything actually restarted correctly.

Before Kafka 3.0.0, administrators had to restart connectors via `POST /connectors/{name}/restart` (without any parameters), and every failed task one by one using the `POST /connectors/{name}/tasks/{taskId}/restart` endpoint.

Resetting offsets of Connectors

Connectors use the concept of offsets to identify their position in the data they are flowing through Connect. For sink connectors, the offsets are literally the offsets in the partitions it is sinking and these are stored in a regular consumer group. For source connectors, the offsets are arbitrary mappings of keys and values generated by the connector and these are stored in the topic specified via `offset.storage.topic`.

When a connector is deleted, the offsets it stored are not deleted. On one hand this is useful if you want to restart the connector later and have it carry on where it was, but it can also be a drawback if you want to restart the connector from scratch or if you don't plan to use this connector anymore. In that case you need to manually

delete the offsets. Before deleting any offsets, ensure the matching connector has been deleted using the REST API.

Sink Connector Offsets

Sink connector offsets can be deleted like any other committed offsets in Kafka. For example you can use the `kafka-consumer-groups.sh` tool. The group name is the connector name prefixed with `connect-`.

For example, let's say we have a sink connector called file-sink and we want to restart it from scratch. The consumer group for this connector is called `connect-file-sink`.

First ensure the connector is deleted by running:

```
$ curl -X DELETE http://localhost:8083/connectors/file-sink
```

Then check the offsets committed for that group by running the following command and ensuring it shows the group has no active members; otherwise you will not be able to delete the offsets:

```
$ ./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --describe \
  --group connect-file-sink

Consumer group 'connect-file-sink' has no active members.

GROUP                TOPIC                PARTITION     CURRENT-OFFSET     LOG-
END-OFFSET     LAG     CONSUMER-ID     HOST          CLIENT-ID
connect-file-sink     topic-to-export      0             210
210               0       -               -             -
```

You can now delete the offsets using:

```
$ ./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --group connect-file-sink \
  --delete \
  --execute
Deletion of requested consumer groups ('connect-file-sink') was successful.
```

Note that we can also adjust the offset by using `kafka-consumer-groups.sh` to commit a new value. For example, to reset the offset to 100:

```
$ ./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --group connect-file-sink \
  --reset-offsets \
  --to-offset 100 \
  --topic topic-to-export
  --execute

GROUP                TOPIC                PARTITION      NEW-OFFSET
connect-file-sink      topic-to-export      0              100
```

Finally we can restart our connector:

```
$ curl -X PUT -H "Content-Type: application/json" http://localhost:8083/connec-
tors/file-sink/config --data "@file-sink.json"
```

Source Connector Offsets

Deleting source connector offsets is a bit more difficult as we can't use Kafka consumer groups tools and instead need to directly interact with the offsets topic from the Connect runtime.

In this example, let's say we have a source connector running called `file-source`. First ensure the connector is deleted by running:

```
$ curl -X DELETE http://localhost:8083/connectors/file-source
```

We then need to find which key Connect used for the offsets of our connector. To do so we can use the `kafka-console-consumer.sh` tool:

```
$ ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic connect-offsets \
  --from-beginning \
  --property print.key=true | grep file-source
["file-source",{"filename":"/tmp/source"}]    {"position":24}
["file-source",{"filename":"/tmp/source"}]    {"position":124}
```

Here `connect-offsets` is the name of the offset topic specified in the Connect configuration using `offset.storage.topic`.

In addition to the connection settings, we used a few flags to display all the information we needed:

- `--property print.key=true`: This prints the key of each record. The key contains the connector name and allows us to filter the specific connector we are interested in with `grep`.
- `--from-beginning`: Since our connector is stopped, it's not emitting new offsets so we need to consume existing offset records.

Looking at the output, we are only interested in the last entry. It contains:

- `["file-source",{"filename":"/tmp/source"}]`: This is the record key. Take note of its value as we will need it to delete the offsets.
- `{"position":124}`: This is the last offset our connector saved.

The offsets topic is a compacted topic, so in order to clear our offsets we need to send a tombstone record with the same key. To do so, since Kafka 3.2.0, we can use the `kafka-console-producer.sh` tool:

```
$ ./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic connect-offsets \
  --property parse.key=true \
```

```
    --property null.marker=NULL \
    --property key.separator=#
>["file-source",{"filename":"/tmp/source"}]#NULL
```

Again we use a few flags to get the desired behavior:

- `--property parse.key=true`: This allows us to include a key in the message we produce.
- `--property null.marker=NULL`: This defines the marker that will represent a `null` value in our message.
- `--property key.separator=#`: This defines how the message key and value are separated.

The message we entered, `["file-source",{"filename":"/tmp/source"}]#NULL`, is a concatenation of the key we retrieved via the `kafka-console-consumer.sh` tool, the key separator and the marker for `null`.

Note that we can also adjust the offset by producing a record with `{"position":<OFFSET>}` as the value where `<OFFSET>` is the desired new offset. For example, to reset the offset to 100:

```
>["file-source",{"filename":"/tmp/source"}]#{"position":100}
```

Finally we can restart our connector:

```
$ curl -X PUT -H "Content-Type: application/json" http://localhost:8083/connec-
tors/file-source/config --data "@file-source.json"
```

Before Kafka 3.2.0 (KIP-810), `kafka-console-producer.sh` was not able to produce tombstones, so you either needed to write a small program or use a third party tool like `kcat` to reset offsets for source connectors.

Administering Connect using the REST API

If you are administering a Connect pipeline it's likely that you will want to use the REST API at some point. The REST API is primarily designed for managing the connectors and tasks that are running on the Connect cluster. Although there are a few endpoints that provide more general purposes.

By default the REST API is available on the port 8083 and is not secured. In Chapter 9 we explain how to change the port and further configure the endpoint, for example to require authentication. The REST API expects all request bodies to use the content type `application/json` and will send all responses using that content type as well.

One thing to note about Connect is that there isn't necessarily a clear separation between the owner of the Connect cluster and the owners of the individual connectors. This is clear when you look at the different endpoints that are offered. The REST API can be used by the administrator of the pipeline to control logging levels or see an overview of the status, or by individual data engineers that are responsible for just a single connector. This makes it even more important to secure your REST endpoint properly to restrict who can access it.

The Kafka documentation provides a reference guide that covers every endpoint. Here we will discuss some operational scenarios that you will likely encounter while using Connect to put each endpoint into context.

We will discuss:

- Creating and deleting a connector
- Connector configuration
- Controlling the lifecycle of a connector
- Debugging issues

Let's first look at the API endpoints that you are most likely to need when first getting started with Connect by examining how to create and delete a connector.

Creating and deleting a connector

Before starting any connector, you should first check the version of your Connect cluster. This will determine the features and options you will be able to use and access. You can retrieve the version using the `GET /` endpoint:

```
$ curl http://localhost:8083/
{
  "version": "3.1.0",
  "commit": "37edeed0777bacb3",
  "kafka_cluster_id": "PSCn87RpRoqhfjAs9KYtuw"
}
```

The next step is to check what connector plugins are available in the Connect cluster. For that you use the `GET /connector-plugins` endpoint:

```
$ curl http://localhost:8083/connector-plugins
{
    "class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type":"sink",
    "version":"3.1.0"
  },
  {
```

```
    "class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
    "type":"source",
    "version":"3.1.0"
  },
  ...
}
```

By default, this will only list the source and sink connector plugins that are installed in the Connect cluster. To discover other kinds of plugins such as transformations, converters and predicates that are installed you can set the `connectorsOnly` query parameter to `false`:

```
$ curl "http://localhost:8083/connector-plugins?connectorsOnly=false"
```

The `connectorsOnly` flag was added to Kafka in version 3.2 as part of KIP-769.

Now let's look at the endpoints for creating a connector. After Connect has started up you can see that no connectors are running yet using the `GET /connectors` endpoint:

```
$ curl http://localhost:8083/connectors
[]
```

To create your connector you have two options: use a `POST` request to this `/connectors` endpoint, or use a `PUT` request using the connector name in the path. The body of the request is slightly different for each endpoint but both have the same effect. The two code snippets below both result in a new FileStreamSink connector that reads records from the `topic-to-export` topic and writes them to a file called `/tmp/sink.out`.

Example `PUT` request:

```
$ curl -X PUT -H "Content-Type: application/json" http://localhost:8083/connectors/file-sink/config --data "@sink-config.json"
{
  "name":"file-sink",
  "config":{
    "connector.class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max":"1",
    "topics":"topic-to-export",
    "file":"/tmp/sink.out",
    "value.converter":"org.apache.kafka.connect.storage.StringConverter",
    "name":"file-sink"
  },
  "tasks":[],"type":"sink"
}
```

Where `sink-config.json` contains:

```
{
  "connector.class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
  "tasks.max": "1",
  "topics": "topic-to-export",
  "file": "/tmp/sink.out",
  "value.converter": "org.apache.kafka.connect.storage.StringConverter"
}
```

Example `POST` request:

```
$ curl -X POST -H "Content-Type: application/json" http://localhost:8083/connec-
tors --data "@sink-config.json"
```

Where `sink-config.json` contains:

```
{
  "name": "file-sink",
  "config": {
    "connector.class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": "1",
    "topics": "topic-to-export",
    "file": "/tmp/sink.out",
    "value.converter": "org.apache.kafka.connect.storage.StringConverter"
  }
}
```

As you can see an advantage of using the `PUT` request is the configuration file is simpler as the config section isn't nested. You can also reuse the `PUT` request to update an existing connector.

Whichever option you choose, once you've created the connector you should now see the connector running:

```
$ curl http://localhost:8083/connectors
["file-sink"]
```

The `GET /connectors` endpoint lists only the names of the connectors in the cluster. To see more information about the connectors and their current status you can use two query parameters. Calling `GET /connectors?expand=info` will list things like the configuration and any tasks, while `GET /connectors?expand=status` shows the status of the connector and related tasks.

You can also use these query parameters together:

```
$ curl "http://localhost:8083/connectors?expand=status&expand=info"
{
  "file-sink": {
    "status": {
      "name": "file-sink",
      "connector": {
        "state": "RUNNING",
        "worker_id": "192.168.1.110:8083"
      },
```

```
      "tasks": [
        {
          "id": 0,
          "state": "RUNNING",
          "worker_id": "192.168.1.110:8083"
        }
      ],
      "type": "sink"
    },
    "info": {
      "name": "file-sink",
      "config": {
        "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnec-
tor",
        "file": "/tmp/sink.out",
        "tasks.max": "1",
        "topics": "topic-to-export",
        "name": "file-sink",
        "value.converter": "org.apache.kafka.connect.storage.StringConverter"
      },
      "tasks": [
        {
          "connector": "file-sink",
          "task": 0
        }
      ],
      "type": "sink"
    }
  }
}
```

If you want to find out the status of a specific connector, rather than all connectors at once, there are a few different endpoints available for you to use. These are:

- `GET /connectors/{connector}/status`: to see the status for a specific connector
- `GET /connectors/{connector}/tasks`: to see tasks for a specific connector
- `GET /connectors/{connector}/tasks/{taskId}/status`: to see status of task for a specific connector

You can see these in action for the connector we just created:

```
$ curl http://localhost:8083/connectors/file-sink/status
{
  "name": "file-sink",
  "connector": {
    "state": "RUNNING",
    "worker_id": "192.168.1.110:8083"
  },
  "tasks": [
    {
      "id": 0,
```

```
      "state": "RUNNING",
      "worker_id": "192.168.1.110:8083"
    }
  ],
  "type": "sink"
}

$ curl http://localhost:8083/connectors/file-sink/tasks
[
  {
    "id": {
      "connector": "file-sink",
      "task": 0
    },
    "config": {
      "file": "/tmp/sink.out",
      "task.class": "org.apache.kafka.connect.file.FileStreamSinkTask",
      "topics": "topic-to-export"
    }
  }
]

$ curl http://localhost:8083/connectors/file-sink/tasks/0/status
{
  "id": 0,
  "state": "RUNNING",
  "worker_id": "192.168.1.110:8083"
}
```

As an administrator of a Connect pipeline it can be useful to understand which topics a particular connector has interacted with. For sink connectors the topic is listed in the configuration, but you can use wildcards such as `sink-*` which makes it harder to get an exact list. For source connectors this problem is even greater since it varies from connector to connector. Some source connectors expect you to specify a topic, others will choose one based on the name of resources in the external system.

To help with this, Connect provides a mechanism to allow you to discover every topic that a connector has interacted with. This endpoint is `GET /connectors/{connector}/topics`. For example, for the `file-sink` connector we just created, you can see the single topic it has interacted with:

```
$ curl http://localhost:8083/connectors/file-sink/topics
{
  "file-sink": {
    "topics": [
      "topic-to-export"
    ]
  }
}
```

You can reset this list so that Connect will "forget" previous interactions using the `PUT /connectors/{connector}/topics/reset` endpoint:

```
$ curl -X PUT http://localhost:8083/connectors/file-sink/topics/reset
```

If you check the topic list again you can see that it's now empty:

```
$ curl http://localhost:8083/connectors/file-sink/topics
{
  "file-sink": {
    "topics": []
  }
}
```

This topic tracking feature was added in Kafka 2.5.0 as part of KIP-558.

The final action you are likely to need in a simple deployment is to delete a connector. You can do that using the following, where `file-sink` is the name of the connector:

```
$ curl -X DELETE http://localhost:8083/connectors/file-sink
```

When a connector is deleted all tasks underneath it are also removed. However, the offsets for the connector aren't reset. That means if a new connector is created with the same name, it will start trying to read using those offsets. To protect against this, it is best practice to reset offsets for the connector once it is deleted. This is especially important for source connectors, since Kafka will never clean these up. Sink offsets will get reset eventually by Kafka, since they are normal consumer offsets, but it is best not to rely on this behavior. You can do this using the steps we described earlier in this chapter.

Now we've looked at creating and deleting connectors, let's review the other endpoints that are available, starting with connector and task configuration.

Connector and task configuration

The REST API provides options to help you list and validate configuration settings for a specific plugin, as well as viewing and updating the configuration of a running connector.

For example, you can get a list of the configuration settings for `FileStreamSinkConnector` like this:

```
$ curl http://localhost:8083/connector-plugins/org.apache.kafka.con-
nect.file.FileStreamSinkConnector/config
[
  {
    "name": "file", ❶
    "type": "STRING", ❷
    "required": false, ❸
    "default_value": null, ❹
    "importance": "HIGH", ❺
    "documentation": "Destination filename. If not specified, the standard out-
put will be used", ❻
    "group": null, ❼
    "width": "NONE", ❽
    "display_name": "file", ❾
    "dependents": [], ❿
    "order": -1 ⓫
  }
]
```

❶ `name`: Name of this configuration setting.

❷ `type`: Expected type of the configuration value, one of `BOOLEAN`, `STRING`, `INT`, `SHORT`, `LONG`, `DOUBLE`, `LIST`, `CLASS`, `PASSWORD`

❸ `required`: Whether this configuration value is required.

❹ `default_value`: Default value, if `required` is `true`, the default will be `null`

❺ `importance`: The importance level of the configuration setting. One of `HIGH`, `MEDIUM`, `LOW`

❻ `documentation`: Information about the configuration setting.

❼ `group`: Which `group` this configuration setting belongs to. Plugins may introduce their own groups for their own settings.

❽ `width`: The `width` of the configuration setting. One of `NONE`, `SHORT`, `MEDIUM`, `LONG`

❾ `display_name`: Display name for the configuration setting, this may match the name.

❿ `dependents`: List of other configuration settings that depend on this setting.

⓫ `order`: Integer order number of the configuration value. `-1` if not set.

As you can see there are a lot of fields for a specific setting. The values for these fields are set by the plugin author so not all plugins have all fields set. They can be very useful if set, but don't rely on them being always present.

The ability to list configuration options for connectors was introduced in Kafka 3.2 via KIP-769 in April 2022. Prior to this release the only way to list them was through a validation request.

Once you have decided what configuration to use, you can validate it using a `PUT` request to `/connector-plugins/{connector plugin}/config/validate`. This endpoint will validate the connector specific configuration, and any generic configuration like `name` or `tasks.max` that is required for all connectors. Let's see an example with `FileStreamSinkConnector` if we miss the connector name from the configuration:

```
$ curl -X PUT -H "Content-Type: appli-
cation/json" http://localhost:8083/connector-plugins/org.apache.kafka.con-
nect.file.FileStreamSinkConnector/config/validate -d '{"connector.class":
"org.apache.kafka.connect.file.FileStreamSinkConnector", "tasks.max": "1", "top-
ics": "sink-topic"}'
{
  "name": "org.apache.kafka.connect.file.FileStreamSinkConnector", ❶
  "error_count": 1, ❷
  "groups": [  ❸
    "Common",
    "Transforms",
    "Predicates",
    "Error Handling"
  ],
  "configs": [
    {
      "definition": {  ❹
        "name": "name",
        "type": "STRING",
        "required": true,
        "default_value": null,
        "importance": "HIGH",
        "documentation": "Globally unique name to use...",
        "group": "Common",
        "width": "MEDIUM",
        "display_name": "Connector name",
        "dependents": [],
        "order": 1
      },
      "value": {
        "name": "name", ❺
        "value": null,  ❻
```

```
        "recommended_values": [],  ❼
        "errors": [  ❽
          "Missing required configuration \"name\" which has no default value."
        ],
        "visible": true  ❾
      }
    },
 ...
  ]
}
```

❶ `name`: Name of the class that provides the plugin.

❷ `error_count`: Count of how many errors were found while validating the provided configuration.

❸ `groups`: Groups that are present in the configuration settings returned.

❹ `definition`: Definition of this configuration setting. Matches the output from the `/connector-plugins/{connector plugin}/config` endpoint.

❺ `name`: Name of this configuration setting.

❻ `value`: The value provided for the configuration setting, `null` if not provided.

❼ `recommended_values`: Valid values for the configuration setting, given the other configuration values provided.

❽ `errors`: An empty array if no error, or an array of error messages for why this value was not acceptable for the configuration setting.

❾ `visible`: Whether this configuration value should be listed.

You might see examples where the `/connector-plugins` endpoint is used with a shortened class name like `FileStreamSinkConnector` or even `FileStreamSink` instead of the fully qualified class name. This is referred to as an "alias". We wouldn't recommend using aliases since it doesn't work if two plugins have the same class name with different packages. This is especially true with transformations that often use nested classes.

We have already mentioned that you can use the `/config` endpoint to create and update the configuration for a specific connector. This endpoint can also be used to check the current configuration of an existing connector:

```
$ curl http://localhost:8083/connectors/file-sink/config
{
  "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
  "file": "/tmp/sink.out",
  "tasks.max": "1",
  "topics": "topic-to-export",
  "name": "file-sink",
  "value.converter": "org.apache.kafka.connect.storage.StringConverter"
}
```

While the configuration of a connector is fully specified both when creating and updating it, the task configuration isn't. An individual connector will take the connector configuration and generate the configuration for the individual tasks. This configuration could be identical, or it could depend on other runtime information. Because of this generation process, it can be useful to check the configuration of individual connector tasks.

You can do this using the `GET /connectors/{connector}/task-config` endpoint:

```
$ curl http://localhost:8083/connectors/file-sink/tasks-config
{
  "file-sink-0": {
    "file": "/tmp/sink.out",
    "task.class": "org.apache.kafka.connect.file.FileStreamSinkTask",
    "topics": "topic-to-export"
  }
}
```

As you can see the configuration for this task is identical to the connector configuration. If you run, for example, one of the MirrorMaker2 connectors, this is not the case.

Now let's look at how you can control the lifecycle of a connector.

Controlling the lifecycle of connectors

In the section on operating Connect clusters earlier in this chapter, we covered the steps to restart failed tasks and connectors. For example you can restart all failed tasks in a specific connector:

```
$ curl -X POST http://localhost:8083/connectors/file-source/restart?include-
Tasks=true&onlyFailed=true
```

As well as restarting them, you can also pause a running connector. This is useful if you want to temporarily stop a connector from flowing data, but want to be able to resume it later from where it left off. This could be to give your external systems a break if they are getting overwhelmed or if you need to apply some maintenance to the external system, such as updating configuration, and don't want any traffic during that time.

You can get a similar result to pausing and resuming by deleting the connector and recreating it with the same name. However we wouldn't recommend this because there is nothing to stop someone else creating a new connector with that name before you can restart yours.

To pause a running connector use the `PUT /connectors/{connector}/pause` endpoint. For example, for a connector called `file-sink` you can pause by running:

```
$ curl -X PUT http://localhost:8083/connectors/file-sink/pause
```

The status endpoint will now show the connector and all tasks in a PAUSED state:

```
$ curl http://localhost:8083/connectors/file-sink/status
{
  "name": "file-sink",
  "connector": {
    "state": "PAUSED",
    "worker_id": "192.168.1.110:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "PAUSED",
      "worker_id": "192.168.1.110:8083"
    }
  ],
  "type": "sink"
}
```

To resume the connector you use a similar command, but with `resume` instead of `pause`:

```
$ curl -X PUT http://localhost:8083/connectors/file-sink/resume
```

The status of the connector and tasks will go back to the `RUNNING` state.

Debugging issues

When investigating and debugging issues with Connect, it's crucial to use logs from the runtime or connectors to peek in their inner workings. Connect exposes a couple of endpoints under `/admin` to enable administrators to view and update logger levels at runtime.

You can use `GET /admin/loggers` to see the current log levels:

```
$ curl http://localhost:8083/admin/loggers
{
  "org.apache.zookeeper": {
    "level": "ERROR"
  },
```

```
  "org.reflections": {
    "level": "ERROR"
  },
  "root": {
    "level": "INFO"
  }
}
```

Let's say we're investigating issues with one of the MirrorMaker connectors, `MirrorSourceConnector` This connector is in the `org.apache.kafka.connect.mirror` package. With the default loggers, it will match `root` and only log at the `INFO` level. To help us out, we can switch it to a more verbose level to get more details. To do so we can use the `PUT /admin/loggers/{logger}` endpoint, for example:

```
$ curl -X PUT -H "Content-Type: application/json" http://localhost:8083/admin/
loggers/org.apache.kafka.connect.mirror -d '{"level": "DEBUG"}'
[
  "org.apache.kafka.connect.mirror",
  "org.apache.kafka.connect.mirror.MirrorCheckpointConnector",
  "org.apache.kafka.connect.mirror.MirrorSourceConnector"
]
```

From the least to the most verbose, the valid log levels are `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG` and `TRACE`.

If we list loggers again, we can see the new loggers we just added:

```
$ curl http://localhost:8083/admin/loggers
{
  "org.apache.kafka.connect.mirror": {
    "level": "DEBUG"
  },
  "org.apache.kafka.connect.mirror.MirrorCheckpointConnector": {
    "level": "DEBUG"
  },
  "org.apache.kafka.connect.mirror.MirrorSourceConnector": {
    "level": "DEBUG"
  },
  "org.apache.zookeeper": {
    "level": "ERROR"
  },
  "org.reflections": {
    "level": "ERROR"
  },
  "root": {
    "level": "INFO"
  }
}
```

Once you have finished debugging any issues, make sure you change the log levels back to `INFO`. Otherwise your log will be diluted with these messages and it might be difficult to diagnose other problems in other connectors. Similarly you should avoid

changing the `root` log level if possible, and instead always configure more specific loggers. That way you can see the exact log lines that will help debug your issue.

Summary

In this chapter we introduced the core concepts administrators need to understand to run Connect clusters in production. Connect is a runtime and while it can run anywhere Java is installed, administrators have to carefully consider how they deploy and run it. Administrators are responsible for picking a deployment model and integrating Connect in their environments.

We also looked at the worker plugins administrators can use to customize their Connect runtimes. Configuration providers are easy to use and useful in most deployments. REST extensions are typically used in more advanced use cases where specific security measures are required.

Then we explained how administrators should size their Connect environments by describing how Connect uses resources such as CPU, memory and network. We saw how estimating the total throughput required helps determine the number of workers to deploy and how to decide the maximum number of tasks each connector should run.

We walked through the most common operations administrators have to perform when maintaining a Connect cluster. Administrators typically use tools or automation to perform these but it's important to understand how they work under the cover to effectively operate workers.

Finally we explored all the endpoints of the REST API by highlighting some operational scenarios. In addition to managing connectors, the REST API is a powerful tool for administrators to perform day to day operations.

CHAPTER 5

Configuring Kafka Connect

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the ninth chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *KafkaConnectBook@gmail.com*.

Throughout this book we mention configuration settings that can be used to customize Connect. All of these settings are documented on the Apache Kafka website where you can find their names, types, default values and descriptions.

In this chapter we go over all the configuration settings and provide additional context to help you identify the key settings to be aware of. You will also see how some of these configurations can be combined to achieve the behaviors required for more advanced use cases.

We first look at the runtime from its most basic configurations that are necessary to start it, to more specific settings that affect the behavior of connectors. We then look at configuring connectors, again starting from the basics and then covering more advanced concepts like client overrides and error handling. Finally in the last section, we look at all the configurations available to secure Kafka Connect clusters for production use cases.

In this chapter we refer to both "runtime configuration" and "worker configuration". Since a worker is really just a single instance of the runtime these terms are somewhat interchangeable. However we aim to use "runtime configuration" when talking generically about how configuration affects the Connect runtime and "worker configuration" when talking about the specific configuration that you use to start a worker.

Since we recommend using distributed mode for production Connect clusters, we won't specifically call out the configurations that don't apply in standalone. As a general rule anything that configures connections between workers doesn't apply for standalone mode. There is only one configuration that is unique to standalone mode: `offset.storage.file.filename`. This configuration specifies the file that the standalone runtime will use to store offsets of source tasks.

Configuring the Runtime

The first set of configurations you should know are those that are required by the runtime to start up. These are:

- `bootstrap.servers`: The address of the Kafka cluster the worker will connect to, this setting takes a list. You should always set it to either multiple brokers, or use the hostname of a redundant load balancer, so Connect is able to reach Kafka even if a broker is down.
- `group.id`: The id used by a Connect worker to join the cluster. This value must not collide with existing consumer groups in this Kafka cluster, otherwise it will lead to errors such as `InconsistentGroupProtocolException`.
- `key.converter` and `value.converter`: The default converters the runtime will provide to connectors. These configurations have no default value so you need to explicitly set them. The last converter, `header.converter`, is set to `SimpleHeaderConverter` by default. We discussed the impact of converters in more detail in Chapter 4. Whatever you set them to, make sure the runtime has both the default converters, and any converters used by specific connectors installed. Connectors cannot install converters themselves, they rely on the plugins already being present. We explain in the next section how connectors can override these values to configure their own converters.

The last set of mandatory settings are for Connect internal topics. Each Connect cluster must use different names for its internal topics. You can specify the names of the 3 topics using:

- `offset .storage.topic`

- `config.storage.topic`
- `status.storage.topic`

You can also specify the replication factor of each topic with:

- `offset.storage.replication.factor`
- `config.storage.replication.factor`
- `status.storage.replication.factor`

By default, the replication factors are all set to 3 which is a good value for ensuring durability. In development environments with a single broker, you need to change the replication factor for all three topics to either -1, to use the broker default, or explicitly to 1 if you are running a version older than Kafka 2.4.

Finally you can set the number of partitions for the offset and status topics with `offset.storage.partitions` and `status.storage.partitions`. Both of these have good default values (25 and 5 respectively) and you should only consider increasing them if you have a very large Connect cluster with dozens of workers. On the other hand, the config topic requires a single partition and this is not configurable. For all three topics, you can also specify any other topic configurations by prefixing the configuration with "`status.storage.`", "`offset.storage.`" or "`config.storage.`". For example, to set `min.insync.replicas` to 2 on the status topic, you can use `status.storage.min.insync.replicas=2`.

The next three configurations are not mandatory but we recommend always using them in all environments:

- `client.id`: This setting has no functional impact however it is useful for monitoring and debugging. You should set a unique value to each worker so you can easily match logs and metrics to each worker instance.
- `plugin.path`: You should install connector and worker plugins using this configuration instead of adding them directly to the JVM classpath or the libs directory. Using `plugin.path` provides some classpath isolation and avoids conflicts between libraries loaded by different plugins.
- `config.providers`: Configuration providers have to be explicitly enabled in order to be used for resolving configurations at runtime. This setting takes a comma separated list of aliases. Each alias needs to be associated with a class that implements the `org.apache.kafka.common.configure.provider.ConfigProvider` interface, by using the following syntax: `config.providers.<ALIAS>.class=<CLASS>`. You need to make sure these classes

are located in a path included in `plugin.path`. You see an example of this in action in Chapter 8.

Now that you know the configurations that are required for getting started, let's look at the configuration settings you will need before deploying to production.

Configurations for Production

In this section we go over all the settings you should be aware of when configuring a Connect cluster for production. Most of these settings have good default values, so you will not need to configure them all. Instead you must understand the options Connect provides and focus on the settings that matter for your specific requirements. In addition to these settings, be sure to check the last section of this chapter that details how to secure Connect clusters.

We will talk about configurations in three categories:

- Clients and connector overrides
- REST configurations
- Miscellaneous configurations

Clients and connector overrides

The runtime runs a consumer for each sink task and a producer for each source task. It also runs admin clients for source tasks that require topics to be created, and for sink tasks that use dead letter queues. You can change the configuration of these clients by including the desired client settings with the appropriate prefix, "`consumer.`" for consumers, "`producer.`" for producers and "`admin.`" for admin clients.

For example if you want producers for source tasks to compress records with GZIP you can add `producer.compression.type=gzip` to your runtime configuration.

Some connectors may want to customize their associated clients further. As a runtime administrator you have the choice of whether or not to allow connectors to do this by setting `connector.client.config.override.policy` to use a configuration override policy. A configuration override policy is a class that implements `org.apache.kafka.connect.connector.policy.ConnectorClientConfigOverridePolicy` and defines which client settings can be overridden by connectors. Connect has three built in implementations:

- `AllConnectorClientConfigOverridePolicy` (or `All`): This is the default, it allows connectors to override all client configurations.

- `NoneConnectorClientConfigOverridePolicy` (or `None`): This prevents connectors from overriding any of the client configurations.
- `PrincipalConnectorClientConfigOverridePolicy` (or `Principal`): This only allows connectors to override the `jaas.sasl.config`, `sasl.mechanism` and `security.protocol` client configurations.

Configuration override policies are pluggable and you can implement your own if you want to prevent connectors from overriding specific configurations.

Finally the Connect runtime internally uses a few Kafka clients to manage its state and interact with its internal topics. The configurations of these clients can be modified by directly including producer, consumer or admin client settings in the runtime configuration. Because of the role of these clients, the runtime prevents you from overriding some fields such as the serializers and deserializers. Outside of very fine tuning, for most production use cases, it only makes sense changing very few client configurations. In environments where Connect is deployed alongside Kafka across multiple racks or data centers, one useful client configuration to change is `client.rack`. By setting this, you allow internal consumers to fetch from a broker located in the same rack, if there is one available.

REST configurations

The next set of configurations for production are for the REST API. By default, the `listeners` configuration is set to `http://:8083`. This means the REST API will be accessible via HTTP on the default network interface (typically localhost) at port 8083. This setting accepts a list of values so that Connect can be configured to listen on multiple interfaces and ports. Connect can be configured to also accept HTTPS traffic, this is described in the last section of this chapter.

The REST API has a subset of endpoints that are all under `/admin`. These endpoints are only meant to be used by administrators, compared to the other endpoints that are typically used by both administrators and engineers building pipelines. By default, the `/admin` endpoints are exposed on the same listeners as all other endpoints. Connect allows you to separate them to dedicated listeners so you can easily provide finer grained access controls to these endpoints. This is done using the `admin.listeners` setting.

For both `listeners` and `admin.listeners`, when the hostname is omitted, like in the `listeners` default value, Connect listens on the default network interface. To listen on a specific interface you need to use its IP address, or you can use 0.0.0.0 to listen on all interfaces.

As we explain in Chapter 8, workers communicate with each other using the REST API `PUT /{connector}/tasks` endpoint. To do this, they need to advertise

their REST API hostname, port and protocol to other workers when they join the cluster. By default this advertised listener will be the first listener in the `listeners` list. However, in many network environments the interface a worker listens on is not directly accessible from the network. In this scenario the worker must advertise a listener that is different from the one they are listening on. You can customize the advertised hostname, port and protocol respectively using `rest.advertised.host.name`, `rest.advertised.port` and `rest.advertised.listener`. The `admin.listeners` configuration does not have a matching advertised configuration because it is not used for inter-worker communication.

Combining all of these REST configurations, a worker might have runtime configuration containing the following:

```
listeners=http://:8093 ❶
admin.listeners=http://:9093 ❷
rest.advertised.host.name=connect-worker-1 ❸
rest.advertised.port=18093
rest.advertised.listener=https
```

❶ For all non admin endpoints, the worker will listen on its default interface on port 8093.

❷ The `/admin` endpoints will be served on port 9093, also on the default interface.

❸ The worker will advertise `https://connect-worker-1:18093` to other workers. For this configuration to work in practice, you would need another network component listening on 18093, terminating TLS and forwarding the requests to the 8093 port. In a Kubernetes environment you typically achieve this with a sidecar container or a service mesh.

To further customize the REST API you can use `rest.extension.classes` to enable REST extensions. This setting accepts a comma separated list of fully qualified class names that implement the `org.apache.kafka.connect.rest.ConnectRestExtension` interface. These classes must be located in a path included in `plugin.path`. We discuss typical use cases for REST extensions in the Securing Connect Clusters section of this chapter.

Miscellaneous configuration

The following settings don't fall under client overrides or REST configurations, but they are still important to consider when you start running Connect in production:

- `metric.reporters`: Whenever you are running Connect, you should always set up a system to collect its metrics so that you are able to monitor it. Like all the other Kafka clients, Connect can use metric reporters to expose its metrics

to monitoring systems such as Prometheus or Graphite. By default, Connect always exposes its metrics via JMX. This setting takes a comma separated list of classes that implement the `org.apache.kafka.common.metrics.MetricsReporter` interface to let you add additional reporters.

- `scheduled.rebalance.max.delay.ms`: The maximum duration a worker can leave the cluster before a rebalance is triggered and its tasks are moved to other workers (defaults to 300000 which is 5 minutes). You should time how long you take to restart a worker in production and set this to a slightly longer duration. That way doing a rolling restart of your Connect cluster will not trigger rebalances. Note that this setting is only applied when the rebalance protocol, `connect.protocol`, is set to `compatible` or `sessioned` (which is the default from Kafka 2.3 onwards).
- `exactly.once.source.support`: Whether exactly once delivery support for source connectors is enabled or not. The default value for this configuration is `disabled`. If you have existing workers with the default configuration of `disabled` you can't switch them straight to `enabled`. First you must change each worker to have `exactly.once.source.support` set to `preparing`. Then you can go back through each worker in the cluster and update the configuration to `enabled`.

Connect exposes a lot of configurations. Fortunately for most use cases many of them can be left untouched and keep their default values. However there are cases where you might set them, so we will discuss them for completeness.

Fine Tuning Configurations

The following configurations are very situational and administrators typically only set them for very specific reasons such as performance or compatibility. If you want to use these do take a look at the documentation for the specific configuration to make sure you are using it correctly. We have grouped these into:

- Connection configurations
- Inter-worker and rebalance configurations
- Topic tracking configurations
- Metrics configurations
- Offset flush configurations

Connection configurations

There are a number of configurations to fine tune the connections between Connect and Kafka. These are the same configurations you find on the other Kafka clients such as the consumer or producer.

- `client.dns.lookup`: How Connect uses DNS records returned when looking up hostnames in `bootstrap.servers`. The default since 2.6 is `use_all_dns_ips`, so if a hostname resolves to multiple IP addresses, Connect tries using them, one at a time, until it connects to the Kafka cluster.
- `connections.max.idle.ms`: The duration Connect keeps a connection to the Kafka cluster alive if there is no traffic.
- `request.timeout.ms`: The duration Connect waits for a response from the Kafka cluster after it has sent a request. Once the limit is reached, Connect retries sending the request.
- `retry.backoff.ms`: The duration Connect waits before retrying sending a request.
- `metadata.max.age.ms`: How often Connect refreshes its metadata. For example if partitions are added to a topic, Connect detects them in the worst case `metadata.max.age.ms` later.
- `reconnect.backoff.max.ms` and `reconnect.backoff.ms`: These two settings control the duration and maximum duration Connect waits before reconnecting if the connection to the Kafka cluster has dropped.
- `socket.connection.setup.timeout.max.ms` and `socket.connection.setup.timeout.ms`: These two settings control the duration and maximum duration Connect can take setting up its connection to the Kafka cluster before it will give up.
- `receive.buffer.bytes` and `send.buffer.bytes`: The sizes of the TCP receive (`SO_RCVBUF`) and send (`SO_SNDBUF`) buffers.

Inter-worker and rebalance configurations

There are also a few settings that configure how workers cooperate to form the Connect cluster.

- `connect.protocol`: The protocol workers use to cooperate and spread the workload across them. Since Kafka 2.4, this defaults to `sessioned` which provides incremental cooperating rebalancing and inter worker security.
- `heartbeat.interval.ms`: Workers, like consumers in a group, have to heartbeat regularly to the coordinator to notify it that they are part of the Connect cluster. This controls how often each worker sends its heartbeats.

- `rebalance.timeout.ms`: The maximum duration allowed for workers to rejoin the cluster when a rebalance happens.
- `session.timeout.ms`: The maximum duration allowed between two heartbeats from a worker. If the coordinator does not receive a heartbeat within this limit, it will kick the worker out of the Connect cluster and start a rebalance.
- `task.shutdown.graceful.timeout.ms`: When a worker wants to stop, it first attempts to gracefully shutdown all tasks it is currently running. This setting controls how long tasks can take to shutdown. If this limit is reached, the worker will terminate and hence halt all tasks. You should take this setting into account when configuring `scheduled.rebalance.max.delay.ms`.

Changing the `heartbeat.interval.ms`, `rebalance.timeout.ms` and `session.timeout.ms` configurations to shorter durations reduces the time it takes to detect worker failures. However this comes at the cost of extra heartbeat traffic and potentially detecting false positives (and triggering unnecessary rebalances) if a worker is just a bit slow. On the other hand, when they are set to longer durations, it will take longer to detect failures but reduce the risk of unnecessary rebalances. The default values offer a good trade off between these two options.

The following settings control the security configuration of inter-worker communications when `connect.protocol` is set to `sessioned`:

- `inter.worker.key.generation.algorithm`: The algorithm used to generate keys.
- `inter.worker.key.size`: The size of generated keys.
- `inter.worker.key.ttl.ms`: The Time To Live (TTL) of generated keys.
- `inter.worker.signature.algorithm`: The algorithm used to sign requests.
- `inter.worker.verification.algorithms`: The list of algorithms used to verify requests.
- `worker.sync.timeout.ms`: The duration a broker will spend trying to resynchronize with other workers if it gets out of sync before leaving the cluster.
- `worker.unsync.backoff.ms`: The duration a worker will wait before attempting to rejoin the cluster when it is out of sync and has already failed rejoining the cluster within `worker.sync.timeout.ms`.

Topic tracking configurations

Since Kafka 2.5, Connect tracks the topics connectors interact with and exposes this information via the REST API. Since sink connectors can subscribe to topics via a regular expression and source connectors can use arbitrary topics this feature helps administrators get some visibility about the topics actively used by each connector. This feature is enabled by default but there's a configuration, `topic.tracking.enable`, that can be set to `false` to allow administrators to disable it.

The REST API also allows users to reset the list of active topics for a connector. This is especially useful if you reconfigure a connector to use a different set of topics. By default, resetting the list is allowed but administrators can decide to prevent this by setting `topic.tracking.allow.reset` to `false`.

Metrics configurations

Connect, like all Kafka clients, exposes some configurations to tune how metrics are recorded.

Several types of metrics, including averages, maximums, minimums, percentiles and more, are measured over multiple samples. A sample can be bound by the number of events or a duration. You can configure the number of samples used for measurements by using `metrics.num.samples` (it defaults to 2). Once this number of samples is reached, the oldest sample is dropped when a new one is created. You can also set the duration of samples by using `metrics.sample.window.ms` (it defaults to 30000).

Similar to application logs, each metric is associated with a recording level which determines how fine grained it is. The last metrics setting is `metrics.recording.level` which allows you to specify the minimum level a metric must be in order to be recorded. It can be set to `INFO` which is the default, or `DEBUG` which will cause all metrics at both `INFO` and `DEBUG` to be recorded. As of Kafka 3.2, all Connect metrics use the `INFO` level, so this setting currently does not allow you to enable recording additional metrics.

Offset flush configurations

There are two settings that allow you to tune how Connect saves offsets for source tasks. You can configure how often offsets are saved to the internal offset topic using `offset.flush.interval.ms` and you can set the timeout for that operation using `offset.flush.timeout.ms`. Since Kafka 3.0.1 you will very rarely have to tune these settings and you may only need to change `offset.flush.timeout.ms` if Connect is geographically far, and hence has a high latency, from the Kafka cluster.

With Connect versions older than 3.0.1, you may see from time to time the following messages in your logs:

```
INFO WorkerSourceTask{id=MyConnector-0} flushing
1000 outstanding messages for offset commit
(org.apache.kafka.connect.runtime.WorkerSourceTask)
ERROR WorkerSourceTask{id=MyConnector-0} Failed to
flush, timed out while waiting for producer to flush
outstanding 510 messages (org.apache.kafka.connect.run-
time.WorkerSourceTask)
```

A search on the Internet will quickly point you to the `offset.flush.interval.ms` and `offset.flush.timeout.ms` settings. Instead of bumping the values of these two configurations, it's important to understand the scenarios that lead to this error.

When a source connector starts, in many cases it will have a sizable backlog of data to flow through Connect. So the connector will provide a lot of `ConnectRecord` objects to the runtime to send to Kafka. By default the producer builds batches of 16kB (`batch.size`) and can send requests up to 1MB (`max.request.size`). But if you're only sending records to a few partitions, the internal producer will not be able to use up the 1MB request size and it will be limited by the relatively small batch size. Before Connect 3.0.1, the flush timeout counted the time the producer took to send records, so when the producer was overwhelmed, and took a few seconds to send many batches, the offset flush timeout could often expire and cause this issue.

Before Connect 3.0.1, if you see this error, instead of increasing `offset.flush.timeout.ms`, we recommend that you first check the producer average request size and if it is low (just a few batches), increase the producer `batch.size` configuration to increase the throughput.

The best solution is to upgrade to a newer Connect release.

Now that you know how to configure the runtime, let's discuss configuring connectors.

Configuring Connectors

When configuring connectors there are two different groups of configurations you must consider. First, there is a set of configurations that the runtime uses to start the connector and build the pipeline. These are configuration settings such as the connector class, converters and client overrides.

Second, each connector also has its own specific configurations that it uses to interact with the external system and emit records. For example for a file connector you

might specify a file name. There aren't any naming conventions or requirements for the connector specific configurations. This means the only way to discover and understand these configuration settings is to either read the connector documentation or use the `GET /connector-plugins/{connector plugin}/config` endpoint. We discuss some common connectors in Chapter 5, so you can also refer to that chapter if you are using one of those connectors.

In this section we will only discuss in detail the first group of configuration settings. The ones that the runtime requires for all connectors, no matter the external system they interact with. Some of these configurations apply to both source and sink connectors, whereas some are specific to one or the other.

Both source and sink connectors have the following required configurations:

- `name`: The name the runtime should use for this connector.
- `connector.class`: The class name or alias for this connector.

In addition, sink connectors require one of `topics` or `topics.regex` to be specified. We discuss what to set these to and other configurations for topics in more detail later in this chapter.

You need to use the connector `name` in many REST API requests, such as to make configuration updates or check status. For that reason, it is recommended to use a name that describes the connector and avoid using completely random names such as generated identifiers. If there is already an existing connector within your Connect cluster with the same name you have chosen, then a create request using the POST endpoint will fail. However the PUT endpoint will accept your request and update the existing connector with your configuration. Make sure the name you have chosen is unique across the Connect cluster before making the request, to avoid accidentally updating an existing connector, rather than creating a new one.

For sink connectors the name actually needs to be globally unique across not only your Connect cluster, but any Connect clusters that are connected to the same Kafka cluster. This is because the name is used to generate the consumer group id that sink connectors use when consuming. If two separate Connect clusters have connectors with the same name connecting to the same topic, the records will be split between the connectors, rather than each of them seeing every record. Currently there isn't anything in Kafka or Connect that prevents this from happening, so be aware of this if you have multiple Connect clusters deployed that are connected to the same Kafka cluster.

The `connector.class` configuration can be either a full class name or an alias. For example to configure the `FileStreamSink` connector you can provide any of `FileStreamSink`, `FileStreamSinkConnector`, `org.apache.kafka.connect.file.FileStreamSinkConnector`. We recommend always using the full class name instead of the alias as this prevents unexpected errors. If Connect has multiple connector plugins installed with the same alias you will get an error when creating the connector:

```
{
    "error_code":500,
    "message":"More than one connector matches alias FileStreamSourceConnector.
Please use full package and class name instead. Classes found: ..."
}
```

The following configurations are optional and can be supplied for both source and sink connectors:

- `tasks.max`: The maximum number of tasks for the connector to start (default is 1).
- `key.converter`: The converter class for record keys.
- `value.converter`: The converter class for record values.
- `header.converter`: The converter class for record headers.
- `transforms`: A comma separated list of aliases for the transformations that will be applied to records.
- `predicates`: A comma separated list of aliases for predicates that will be used with the transformations.
- `config.action.reload`: The action the Connect runtime should take when changes in external configuration providers results in a change to the connector configuration. The default is `restart` and the valid values are `none` and `restart`.

The `tasks.max` configuration is passed to the connector and the connector will use it to determine the actual number of tasks to run. This is an important configuration setting to get right because it impacts throughput and ordering. We discuss how to decide on the value in Chapter 8.

Each of the converter configuration settings (`key.converter`, `value.converter`, `header.converter`) is optional and they have no default value. If these aren't provided, the connector will instead use the values specified in the Connect runtime configuration. These are important settings because they define the format of the data as it goes into and out of Kafka. It is good practice to always override `key.converter` and `value.converter` at the connector level, as the correct converters to use are likely to be dependent on the specific connector that is running and the use case it satisfies.

The `transforms` and `predicates` configurations are only needed if you want to apply a transformation to your data as part of the flow. We cover how to configure these in detail in Chapter 3.

The `config.action.reload` configuration is used in conjunction with configuration provider plugins. When a configuration provider returns a configuration value to the Connect runtime, it has the option to include a "time-to-live". This indicates how long this configuration value will be valid for. When the runtime receives this information it can plan to restart the connector once the amount of time specified in the time-to-live has passed. Connect will only schedule a restart if the `config.action.reload` configuration is set to `restart`, which is the default value. It will do nothing if this is set to `none`. Connect will also only restart the connector, not the running tasks, so the tasks won't automatically pick up this new value when the time-to-live is hit. Instead, the next time the task reloads the configuration, for example after a restart or during a rebalance, it will pick up the new value. If you take a look at the Javadoc for `ConfigProvider` you will see that it also includes the `subscribe()`, `unsubscribe()` and `unsubscribeAll()` methods. However, as of Kafka 3.2, these are not called by the Connect runtime so `ConfigProvider` authors should not implement them.

Here is an example of a connector configuration with both the required and optional configuration values set:

```
{
  "name": "file-sink",
  "config": {
    "connector.class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": "1",
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "value.converter": "org.apache.kafka.connect.storage.StringConverter",
    "header.converter": "org.apache.kafka.connect.storage.SimpleHeader-
Converter",
    "transforms": "filterTombstones",
    "transforms.filterTombstones.type": "org.apache.kafka.connect.trans-
forms.Filter",
    "transforms.filterTombstones.predicate": "isTombstone",
    "predicates": "isTombstone",
    "predicates.isTombstone.type": "org.apache.kafka.connect.transforms.predi-
cates.RecordIsTombstone",
    "config.action.reload": "none"
  }
}
```

In addition to these common configurations, the source and sink connectors each have their own configuration settings for the topics they will use.

Topic Configurations

The topic configuration settings for sink and source connectors are different. We will cover each setting and discuss how topic creation works for both source and sink connectors. The topic configurations are:

- Sink connectors
  - `topics`
  - `topics.regex`
- Source connectors
  - `topic.creation.groups`

Let's start with sink connectors. Sink connectors must be configured with either a `topics` or `topics.regex` field. The `topics` setting is a comma separated list of topics in Kafka that the runtime will consume from, for example `topics=mytopic1,mytopic2`. If the broker setting `auto.create.topics.enable` is set to `true` (the default), the topics listed will be created by the broker when the connector is started. If `auto.create.topics.enable` is `false` you will need to create the topics manually.

The `topics.regex` setting is a regular expression describing the topics to consume from. For example `topics.regex=mytopic-(.*)` would match the topics `mytopic-a`, `mytopic-second` and `mytopic-1`. When the connector starts, it will subscribe to all topics that match the regex. If you do create additional topics later, the connector will detect them automatically, but only when it refreshes its metadata after the duration set by `metadata.max.age.ms`.

Source connectors need topics to produce records to, and these topics can be created in one of three ways:

- Auto-created by the broker when the first record is sent.
- Created by the Connect runtime.
- Manually created ahead of time.

The first option is the simplest because it doesn't require any additional configuration. Make sure the broker configuration `auto.create.topics.enable` is still using the default value of `true`, then topics will be created using the broker defaults.

In many production environments auto topic creation is disabled to prevent applications from inadvertently creating topics. In these environments, Connect can use its admin client to explicitly create topics. This feature was added by KIP-158 in Kafka 2.6. To use this option all Connect workers must have `topic.creation.enable` set to the default value of `true` in their runtime configuration. Then a connector

can opt-in to topic creation by configuring various settings that are prefixed with `topic.creation.`. Let's look at how this works.

You first have to specify a group of topics that will be known by an alias. There is a predefined group with the alias `default` which applies to all topics. You can define additional groups using `topic.creation.groups`. For example `topic.creation.groups=ordered,single` results in three groups; `default`, `ordered` and `single`.

You define the specific topics to include in each group using `topic.creation.<alias>.include` and `topic.creation.<alias>.exclude`, where `<alias>` is the group name. These are both comma separated lists of topic names or topic regex values. To define topic level configurations for a specific group you use the following syntax where `<config>` is any topic level configuration:

```
topic.creation.<alias>.<config>
```

When you set topic level configurations make sure you are choosing values that are supported by your Kafka brokers. For example, from Kafka 2.4 onwards admin clients can use `-1` for the replication factor and partitions to use the broker defaults, so setting the `topic.creation.<alias>.partitions` to `-1` will only work if your brokers and Connect workers are from Kafka 2.4 or newer and therefore support this feature.

So for example the following connector configuration will result in all topics having a replication factor of 3. Then the topic called `status` and any topics beginning with `single.` will have 1 partition, while the rest will have 5 partitions.

```
{
  ...
  "topic.creation.groups": "ordered",
  "topic.creation.default.replication.factor": "3",
  "topic.creation.default.partitions": "5",

  "topic.creation.ordered.include": "status,single.*",
  "topic.creation.ordered.partitions": "1",
  ...
}
```

Connect will only create topics using the admin client if the default group has both the `replication.factor` and `partitions` set. If you specify one but not the other, the connector will fail to start.

The final option is to manually create topics. For this approach you don't need to worry about giving Connect the permissions to create topics. However, be aware that if the topics aren't created before connectors are started, your Connect logs will fill up with warning messages:

```
WARN [file-source|task-0] [Producer clientId=connector-producer-file-source-0]
Error while fetching metadata with correlation id 570 : {topic-from-
source=UNKNOWN_TOPIC_OR_PARTITION} (org.apache.kafka.clients.NetworkClient:1090)
```

We have seen how you can override the topic configurations for the topics that connectors use, now let's look at how you can override the client configurations.

Client Overrides

As discussed in the Clients and connector overrides section of this Chapter, the Connect runtime can specify configurations for the clients that are used for source and sink tasks. You can also configure these settings at the connector level to give you even more control over the producer, consumer and admin clients used by tasks.

To provide connector level client overrides, first make sure the runtime configuration `connector.client.config.override.policy` allows you to set the configurations you want to change. Then add these client configurations to your connector configuration with the prefix `producer.override.`, `consumer.override.` or `admin.override.` depending on the client you want to override.

The advantage of providing overrides at the connector level is you can choose configuration settings that suit the particular connector and workload you are dealing with. As a general rule, you should set at the connector level any configuration that needs to be a specific value for a specific connector instance. Following this rule means you can be sure about the configuration values that your connector is using, and it protects you if the runtime level configuration gets updated.

If you are configuring a source connector you should specifically think about what producer client overrides you might want. Consider settings that affect how the data is batched and buffered before being sent to Kafka, for example `batch.size`, `linger.ms` and `buffer.memory`. If you know the connector will be reading large amounts of data from the external system in one go you might increase the batch size. For source connectors you can also override the `partitioner.class` configuration to influence how data is partitioned when it is sent to Kafka. Remember if you want to use a custom partitioner it has to be present in the runtime.

Configuration for a source connector could for example include:

```
{
  ...
  "producer.override.batch.size": "32768",
  "producer.override.partitioner.class": "org.apache.Kafka.clients.producer.Uni-
formStickyPartitioner",
```

```
  ...
}
```

For sink connectors, you override consumer configuration settings. Consider where you want new connectors to start processing from and make sure you set `auto.offset.reset` appropriately. If the topic your connector is reading from has records that are sent as part of transactions, you should also consider changing the `isolation.level` configuration to `read_committed` so your connector will only get records that are part of committed transactions. For example you might have sink connector configuration with:

```
{
  ...
  "consumer.override.auto.offset.reset": "latest",
  "consumer.override.isolation.level": "read_committed",
  ...
}
```

As well as being able to override any client configurations, there are also some connector specific configurations for exactly once delivery of records.

Configurations for Exactly Once

We have already discussed the options you have around delivery guarantees in Chapter 4. In that Chapter we explain the different ways that at most once, at least once and exactly once delivery can be achieved between external systems and Kafka. The only specific configurations that Connect provides for exactly once delivery are for data being processed by source connectors. That is what we discuss in this section.

Exactly once support for source connectors is a pretty recent addition to Connect. It was added in Kafka 3.3 as part of KIP-618. If you want to make use of this feature, first make sure the connector you are using supports it and every worker in the Connect cluster has the `exactly.once.source.support` configuration set to `enabled`. Then you need to specify some connector level configurations. There are four connector-level configurations you can set:

- `exactly.once.support`
- `transaction.boundary`
- `transaction.boundary.interval.ms`
- `offsets.storage.topic`

The `exactly.once.support` configuration specifies whether the connector is required to support exactly once delivery. If `exactly.once.source.support` is enabled at the worker level, the runtime will query new connectors during create and validate requests to see if they support it. The default value for `exactly.once.sup`

port is requested and means the connector isn't required to support exactly once delivery. This doesn't necessarily mean that it can't deliver records exactly once though, and if you use this default value you should carefully read the connector documentation to understand the behavior you will get. The other value for the exactly.once.support configuration is required and this means any validate and create requests will only succeed if the connector reports that it does support exactly once delivery.

Exactly once delivery support is done using transactions that span the two actions of sending the data to Kafka and committing offsets to the offsets storage topic. The transaction.boundary configuration lets you specify when these transactions should be created and committed. The allowed values are:

- poll: A new transaction is used for every batch of records that a task in the connector passes to the Connect runtime. This is the default value.
- interval: Transactions are committed after a user-defined interval. You set this interval using either the transaction.boundary.interval.ms configuration, or the worker level offset.flush.interval.ms configuration if the former isn't set.
- connector: The connector should provide the transaction boundaries. Not all connectors are able to do this. If the connector doesn't support this feature, attempts to create the connector with the configuration set to this value will fail.

Since this feature uses transactions, whatever transaction.boundary you configure you will need to make sure that Connect has the right permissions to participate in transactions. The Kafka documentation includes the exact permissions that are needed.

Finally you can use offsets.storage.topic to configure the topic the connector will use to commit offsets and keep track of what data it has processed in the external system. This topic must be in the same Kafka cluster as the topic the connector is writing to, since transactions cannot span different Kafka clusters. By default this configuration will use the global worker level offset storage topic.

Finally for connector configurations let's look at how you can influence error handling.

Configurations for Error Handling

The default configuration of Connect means that if a task encounters an error it will stop processing data and be moved into the FAILED state. This is true for both source and sink connectors and requires you to restart the failed task using the REST API. However Connect does include configurations to let you alter this behavior. First we will look at the common error configurations that can be used for both

source and sink connectors, and then discuss dead letter queue configurations for sink connectors.

The set of common error configurations are:

- `errors.retry.timeout`
- `errors.retry.delay.max.ms`
- `errors.tolerance`
- `errors.log.enable`
- `errors.log.include.messages`

The first two configurations handle the behavior of Connect when an operation fails. The configuration `errors.retry.timeout` indicates how long, in milliseconds, Connect should spend retrying a failed operation. If you don't specify this configuration it will use the default value of 0 which means no retries occur. To use infinite retries you can set this to -1. Remember, even if you enable retries, Connect will only retry certain operations that are viewed as "retriable". We cover which operations those are in Chapter 4. If you have enabled retries you can then use `errors.retry.delay.max.ms` to control the maximum time in milliseconds Connect should wait between consecutive retries. The default value for `errors.retry.delay.max.ms` is 60000 (1 minute).

The `errors.tolerance` configuration lets you configure what the connector should do if it encounters an error when processing a record. By default this is set to `none` which means the connector task will stop processing records and move into the `FAILED` state. If you instead want the connector to skip over the record that caused the error and keep processing, you can set this to `all`. Until Kafka 3.2 this setting did not apply to source tasks that failed while producing records. So connectors with older versions of Kafka would fail in this scenario even if `errors.tolerance` is set to `all`. From 3.2 onwards Connect will inform the connectors if it encounters this kind of error in a source flow.

The final two configurations handle how errors are logged. The first, `errors.log.enable`, configures whether Connect should log tolerated errors. If you haven't changed the `errors.tolerance` to `all` you can keep this as the default value of `false`, since Connect will always log errors that fail a task. If you have enabled connectors to skip records then you should change `errors.log.enable` to `true` so you can see when records are being skipped.

Finally you can set `errors.log.include.messages` to `true` (default is `false`) so that Connect logs the contents of records that resulted in a failure. For sink connectors this logs the topic, partition, offset and timestamp of the Kafka record. For source connectors it logs the key, value, key schema, value schema, headers, timestamp, Kafka topic, Kafka partition, source partition and source offset. This configuration is

useful when developing your pipeline to understand why certain records are causing failures.

In production you should be cautious about changing `errors.log.enable` to `true`. Record contents, such as the key, value and headers might contain sensitive information, which would then be leaked to the log files. Moreover if a lot of records are causing problems, or the records are very large, this could result in a lot of additional log lines, which could make it harder to spot and diagnose other problems.

In addition to the error configurations we have already mentioned, sink connectors have three configurations for dead letter queues (DLQ). We discuss why and how you might use dead letter queues in Chapter 4. The configurations for DLQs are:

- `errors.deadletterqueue.topic.name`: The name of the topic to use as the DLQ. You must set it if you want records to be sent to a DLQ.
- `errors.deadletterqueue.topic.replication.factor`: The replication factor Connect should use for the DLQ topic if it needs to be created. The default value is 3.
- `errors.deadletterqueue.context.headers.enable`: Whether Connect should add headers containing error context to the records sent to the DLQ. The default value is `false`. If you set this to `true` Connect will create error context headers prefixed with `__connect.errors.` to prevent header clashes.

By default Connect is not secured and any user can access pretty much everything. While this is practical for getting started and development environments, this is often not suitable for production environments. In these environments, with real data flowing, you typically want to tighten security and police access to all resources, and there are numerous configurations that enable you to do this.

Configuring Connect Clusters for Security

Security is often looked at via these 3 pillars:

- Confidentiality: This means controlling who has access to each piece of data.
- Integrity: This means controlling who can alter data and ensuring data stays intact and can't be corrupted.
- Availability: This means ensuring data stays available and ready to use.

Connect supports a number of security standards, providing encryption, authentication and authorization, that you can combine to obtain the required security for your

use cases. By encryption we mean encrypting the data as it is sent between Connect, the data source and sinks, and Kafka. Authentication is the process of identifying the originator of a request, while authorization describes whether a permission is associated with this identity.

In this section we look at the options that Connect provides for security between itself and Kafka. You should also be securing the connection between Connect and your source and sink systems, but the way you do that is dependent on the system and the connector. Instead we discuss options for assigning permissions to connectors, both for Kafka and the external system. We also give recommendations around vetting connectors from a security perspective. Finally we will discuss how to secure connections to the Connect REST API.

First let's look at securing the connection to Kafka.

Securing the Connection to Kafka

The first aspect to consider when securing a Connect cluster is identifying how it is going to connect to Kafka. By default, Connect interacts with Kafka via a plaintext connection, so all the data is exchanged in clear and Connect does not obtain a specific identity or any associated permissions in Kafka.

Using `security.protocol`, you can change the connection to use encryption via TLS and authentication via SASL. The valid values for `security.protocol` are:

- `PLAINTEXT`: This is the default and provides no encryption or authentication.
- `SSL`: This provides encryption and can also be used for authentication using client certificates. To use this you need to configure some of the settings listed in the TLS section below.
- `SASL_PLAINTEXT`: This provides authentication but no encryption. To use this you need to configure some of the settings listed in the SASL section below.
- `SASL_SSL`: This provides encryption and authentication. To use this you need to configure some of the settings listed in both the TLS and SASL sections below.

The `security.protocol` you choose has to match one of the listeners the Kafka cluster is configured with. If Kafka only exposes a `PLAINTEXT`listener, Connect cannot use encryption or authentication.

Now let's look at what configurations you can use for an encrypted connection.

TLS configurations

If the Kafka cluster exposes a `SASL_SSL` or `SSL` listener, you can configure how the connection is encrypted using TLS configurations.

In most cases you do not need to configure all of the settings. Actually in many instances, such as when connecting to some public Kafka-as-a-service offerings, you don't need to set any of them as the default values are sufficient.

If the Kafka cluster certificates are signed by a public Certificate Authority (CA), for example Let's encrypt, then by default the JVM will trust them. Otherwise, for example you are using your own CA, you need to load the server certificate in a truststore and provide that to Connect.

By default Connect supports two types of truststore files, `JKS` (Java KeyStore, the default) and `PEM` (Privacy-Enhanced Mail). You can configure the type you want to use with `ssl.truststore.type`. You specify where the file is in the filesystem using the `ssl.truststore.location` setting. If your JKS truststore is password protected, you can specify the password using `ssl.truststore.password`. For `PEM` certificates, you can provide them directly inline in the configuration rather than using a file by using `ssl.truststore.certificates`.

By default, Connect also validates that the server name matches the name specified in the certificate. This is done to prevent man in the middle attacks. It is recommended to leave this enabled, but if you need to disable this feature you can do it by setting `ssl.endpoint.identification.algorithm` to an empty string.

Now that we've looked at validating the identity of the Kafka brokers, the next set of TLS configurations are about providing an identity to Connect. This allows you to use Mutual TLS (mTLS) and ensure that both peers, the Kafka brokers and Connect, are who they claim to be. The identity provided can then be used by Kafka to perform authentication and enforce permissions. This is done via a keystore.

Similarly to truststores, Connect works out of the box with `JKS` (the default) or `PEM` keystores and it's configured using `ssl.keystore.type`. You specify the location of the keystore on the filesystem using the `ssl.keystore.location` setting. If your `JKS` keystore is password protected, you can specify the password using `ssl.keystore.password`. The key in a JKS keystore can also be password protected. If that is the case, you can provide its password using `ssl.key.password`. For `PEM` keystores, you can provide them directly inline in the configuration rather than using a file by using `ssl.keystore.certificate.chain`.

There are also settings to configure the TLS version and ciphers to use when establishing a connection. When running with Java 11 or above, `ssl.protocol` defaults to `TLSv1.3`, otherwise it defaults to `TLSv1.2`. Connect will try to use the chosen version when connecting to Kafka. If the Kafka cluster does not support it, Connect will try the other versions specified in `ssl.enabled.protocols`, this defaults to `TLSv1.2,TLSv1.3`. You should not use older TLS versions than those in this list as they all have security issues. Alongside the version, ciphers determine how data is

encrypted and can be configured using `ssl.cipher.suites`. By default this is set to `null` which allows all ciphers that are enabled in the JVM to be used.

Finally there are advanced TLS configurations to customize how TLS is handled by the JVM running Connect. You can change the security provider used by the JVM using `ssl.provider`. By default this setting is `null` and the JVM uses its default security provider, but you can change this setting to load your own implementation of `java.security.Provider`. A custom security provider allows you to deeply alter how the JVM handles TLS. If you only need to customize the `SSLEngine`, you can use `ssl.engine.factory.class` to provide your own `SSLEngine` factory that implemented `org.apache.kafka.common.security.auth.SslEngineFactory`. A custom `SslEngineFactory` can be used to load custom keystores and truststores, or finely configure the `SSLEngine` and `SSLContext` used to encrypt data. You can also tweak various TLS components such as the algorithm used by the Trust Manager using `ssl.trustmanager.algorithm` (the default is `PKIX`), the algorithm used by the Key Manager using `ssl.keymanager.algorithm` (the default is `SunX509`) and the pseudo random number generator (PRNG) implementation using `ssl.secure.random.implementation` (the default is `null` which uses the default PRNG implementation of the JVM).

Now we have discussed the configurations for encrypted connections we will look at the configurations for connections that require SASL authentication.

SASL configurations

Simple Authentication and Security Layer (SASL) is a framework for providing authentication. The framework does not specify how authentication is done, this part is defined by SASL mechanisms. Kafka supports five SASL mechanisms:

- `GSSAPI`: The default mechanism. This provides ticket based authentication. The main implementation of this mechanism is Kerberos.
- `PLAIN`: Simple username and password authentication.
- `SCRAM-SHA-256`: This works with a username and password but instead of exchanging credentials directly like in `PLAIN`, credentials are cryptographically hashed to 256 bits before being exchanged.
- `SCRAM-SHA-512`: The same as `SCRAM-SHA-256` but it uses a 512 bit hash instead.
- `OAUTHBEARER`: OAuth2 bearer token authentication.

You set the mechanism for Connect to use with the `sasl.mechanism` setting.

Once the mechanism is selected, SASL is configured via Java Authentication and Authorization Service (JAAS). There are two ways to provide a JAAS configuration, using the `sasl.jaas.config` configuration setting or using a JAAS file.

The recommended method is to use `sasl.jaas.config`. The value for that setting must be in the following format `loginModuleClass controlFlag (optionName=optionValue)*;`. Each mechanism has its own set of valid `optionName` values.

For example:

```
sasl.jaas.config="org.apache.kafka.common.security.plain.PlainLoginModule
required username=\"alice\" password=\"alice-secret\";"
```

You can also use a JAAS file and pass it to the JVM using the `-Djava.security.auth.login.config` system property. The file must have the following format:

```
KafkaClient {
    loginModuleClass controlFlag
    (optionName=optionValue)*;
};
```

For example:

```
KafkaClient {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="alice"
    password="alice-secret";
};
```

The format of the JAAS configuration is very finicky and can fail for many reasons including invalid spacing! Be sure to review it first when investigating authentication issues.

When using a JAAS file, all Kafka clients created in the JVM share the same credentials. So with Connect you should avoid this method. Also if you use both methods, the credentials provided via `sasl.jaas.config` take precedence.

Depending on your use case, in production environments, you may have to provide custom callback handlers to securely retrieve credentials at runtime. To do so, you can use:

- `sasl.client.callback.handler.class`: This setting takes a class that implements the `AuthenticateCallbackHandler` interface. It serves as the interface between the SASL client and the login module loaded in the JVM. When using `PLAIN` this lets you retrieve credentials from an external source.
- `sasl.login.callback.handler.class`: This setting also takes a class that implements the `AuthenticateCallbackHandler` interface. It's used by the login module loaded by the JVM to retrieve credentials from an identity provider. For example it can be used with `OAUTHBEARER` to obtain and refresh tokens.

If you need to further customize the SASL login flow you can also provide your own implementation of the `org.apache.kafka.common.security.auth.Login` interface by using `sasl.login.class`.

Let's now look at SASL configurations that apply to specific mechanisms, starting with `OAUTHBEARER`.

SASL OAUTHBEARER configurations

This mechanism was added in Kafka 2.0 via KIP-255 but it originally only provided a framework to integrate with OAuth2 providers. Later in Kafka 3.1, via KIP-768, a production ready OAuth/OpenID Connect (OIDC) implementation was added. In order to customize it, Connect exposes the following settings:

- `sasl.oauthbearer.token.endpoint.url`: The URL for the OAuth/OIDC identity provider.
- `sasl.oauthbearer.clock.skew.seconds`: The maximum time difference allowed between the OAuth/OIDC identity provider and Connect clocks. If a large difference is detected, the token is rejected and a new one will be retrieved.
- `sasl.oauthbearer.expected.audience` and `sasl.oauthbearer.expected.issuer`: These two settings are used to perform sanity checks on the token retrieved and ensure that the token `aud` (for audience) and `iss` (for issuer) fields respectively match these settings.
- `sasl.oauthbearer.scope.claim.name`: Allows you to override the name of the scope claim to allow compatibility with different providers. It defaults to `scope`.
- `sasl.oauthbearer.sub.claim.name`: Allows you to override the name of the sub claim to allow compatibility with different providers. It defaults to `sub`.

Since Kafka 2.2 and KIP-368 this mechanism supports reauthentications. This means brokers can force clients to reauthenticate periodically for added security. There are a few settings to configure this behavior:

- `sasl.login.refresh.buffer.seconds`: The minimum time allowed before authentication expires to trigger a reauthentication. It defaults to 300 seconds.
- `sasl.login.refresh.min.period.seconds`: The minimum time between two reauthentications. This defaults to 60 seconds.
- `sasl.login.refresh.window.factor`: When authenticating, the client is told the lifetime of its credentials. This controls at what ratio of that lifetime the client should reauthenticate. It defaults to 0.8, which means once 80% of the lifetime of the credentials has passed, the client will reauthenticate.

- `sasl.login.refresh.window.jitter`: This delays by a random amount of time when reauthentication happens. This ensures reauthentication requests will not all happen at the exact same time. It defaults to 0.05 which means reauthentication can be delayed by up to 5%.
- `sasl.login.connect.timeout.ms`: The maximum duration a connection to the authentication provider can take to be established before failing. By default this is not set and the request will not time out.
- `sasl.login.read.timeout.ms`: The maximum duration to wait when expecting a response from the authentication provider before failing. By default this is not set and the request will not time out.
- `sasl.login.retry.backoff.max.ms`: After a failure, this specifies the maximum duration to wait before retrying. This defaults to 10 seconds.
- `sasl.login.retry.backoff.ms`: After a failure, this specifies the minimum duration to wait. Retries are then done using an exponential backoff up to `sasl.login.retry.backoff.max.ms`. This defaults to 100 milliseconds.

SASL GSSAPI configurations

Kafka also exposes a few settings to configure GSSAPI authentication:

- `sasl.kerberos.service.name`: This must match the Kerberos name that the Kafka cluster is using.
- `sasl.kerberos.kinit.cmd`: In Kerberos, `kinit` is the command that retrieves tickets for authentication. This setting specifies the path to the command on the filesystem. This defaults to `/usr/bin/kinit`.
- `sasl.kerberos.min.time.before.relogin`: The minimum duration to wait before refreshing tickets. This defaults to 60 seconds.
- `sasl.kerberos.ticket.renew.jitter`: This delays by a random amount of time the ticket refresh. This defaults to 0.05 which means refresh can be delayed by up to 5%.
- `sasl.kerberos.ticket.renew.window.factor`: When retrieving a ticket, the client is given a refresh time. This controls the ratio of the ticket validity time after which the client should refresh its ticket. This defaults to 0.8, which means once 80% of the ticket validity time has passed, the client will refresh it.

Now that we have discussed specific configurations for securing the connection between Connect and Kafka, we will talk about the general considerations you should make when configuring permissions in Connect, Kafka and external systems.

Configuring Permissions

In the previous section we talked about how to configure Connect to be able to communicate with a secured Kafka cluster, but we didn't discuss how to decide the permissions to grant Connect. As a rule of thumb, you should only give Connect the minimum set of permissions it needs to connect both to Kafka and to the external system you are flowing data to/from.

In Chapter 8 we highlighted the full list of ACLs that Connect needs, both for the runtime and the individual connectors, to talk to Kafka. At a minimum the Connect runtime needs to be able to interact with its configuration, status and offsets topic and the consumer group that is used for worker coordination. Each connector then needs read or write permissions to the topics it is interacting with. When you are configuring the Kafka ACLs for Connect you have a choice between having a single principal or multiple principals for the runtime and connectors respectively.

The simplest configuration is to have one principal that is granted all the ACLs for the Connect runtime and any connector you are going to run. With this approach you don't need to configure any overrides for security configurations when you start connectors, because the connectors inherit their identity from the runtime. The downside of this approach is that all connectors can access any topic that the runtime has access to. Meaning that a connector could potentially read or write to a topic it shouldn't.

In reality you will likely have some topics in your Kafka cluster that contain sensitive data and need to be locked down to restrict which clients can access them. If you need to run a connector to interact with such a topic, you should configure a client override in the connector configuration, so it will use its own principal. Then you only need to grant access to that specific topic to the connector principal, not to the runtime principal that will be used by other connectors.

For connections to external systems, configure your connectors to use the authentication, authorization and encryption options they provide. Make sure you have chosen a connector that supports authentication and authorization and that you configure your external system to require authenticated connections. Security in a pipeline has to happen end to end, and like a chain, it is as strong as the weakest link.

You should carefully vet any connectors that you plan to install into the runtime. Remember that Connect does not provide a mechanism to restrict who can start which type of connector. Anyone with permissions for the REST API can start a connector using any of the installed connector plugins. Make sure you understand what permissions and actions the connectors in your runtime are capable of, and decide whether there are any security risks from these connectors.

If you are running Kafka 3.1 or older you should also remove the `connect-file-<version>.jar` file, that contains the `FileStreamSource` and `FileStreamSink` connector,

from the `libs` directory of your Connect runtime. This is because they can read and write from/to the file system and they can be used by bad actors to scrape sensitive information from Connect workers. They have been removed from the default classpath in Kafka 3.1.1 onwards.

The final part in securing Connect is to secure the REST API. The REST API exposes a lot of features so it shares the same security concerns we've described at the start of this section. If needed, data can be encrypted and access to REST endpoints can be protected via authentication and authorization.

Securing the REST API

To encrypt data sent via the REST API, you need to expose it over HTTPS. The HTTPS configuration reuses all the `ssl.*` configurations detailed earlier in this chapter. However note that in this case Connect will be the server as opposed to previously where Connect is the client connecting to Kafka. Being the server, we can also make Connect validate the identity of clients connecting and authenticate them by using Mutual TLS. To do so, you can set `ssl.client.auth` to `required` to only allow clients that have a trusted certificate. You can also set it to `requested` to optionally authenticate clients if they provide a certificate. By default this is set to `none` which disables this feature.

In order to change TLS settings you need to prefix them with `listeners.https.`. For example you might have the following runtime configuration:

```
listeners=https://:8443 ❶
listeners.https.ssl.keystore.location=/my/path/keystore.jks ❷
listeners.https.ssl.keystore.password=${file:/tmp/ssl.properties:password} ❸
ssl.client.auth=required ❹
```

❶ This exposes a HTTPS listener on the default interface on port 8443.

❷ The configuration overrides `ssl.keystore.location` to point to a specific keystore on the filesystem.

❸ Always use a configuration provider to protect sensitive settings.

❹ This enables mandatory Mutual TLS for all clients.

The REST API also supports Cross-Origin Resource Sharing (CORS). This allows requests to be made to the REST API from a different domain. This is disabled by default, but you can allow specific domains by adding them to `access.control.allow.origin` as a comma-separated list, or allow all domains by setting this configuration to `*`. The `access.control.allow.methods` setting also lets you restrict

the HTTP methods (GET, POST, PUT, etc) allowed from cross origin requests. The default value, empty string, allows the following methods: GET, POST and HEAD.

In order to protect REST API endpoints, it's sometimes necessary to have headers included in responses and many enterprise security scanners report issues in case headers like `X-XSS-Protection` or `Content-Security-Policy` are missing. In order to address that, from Kafka 2.6 (KIP-577) you can configure Connect to include HTTP response headers by using the `response.http.headers.config` setting. This takes a comma separated list of header rules where each rule has the following format: `[action] [header name]:[header value]`. The `action` field can be:

- `set`: To set, or create if it does not exist, a header with a value.
- `add`: To add a value to a header. Headers can have multiple values.
- `setDate`: To set, or create if it does not exist, a header with the number of milliseconds since the epoch.
- `addDate`: To add the number of milliseconds since the epoch to a header. Headers can have multiple values.

For example, you can set it to the following:

```
response.http.headers.config=add X-XSS-Protection: 1; mode=block, add Strict-
Transport-Security: max-age=31536000; includeSubDomains, add X-Content-Type-
Options: nosniff
```

In that case a response will contain:

```
HTTP/1.1 200 OK
Date: Wed, 18 May 2022 11:10:56 GMT
Strict-Transport-Security: max-age=31536000;includeSubDomains
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Type: application/json
```

The last approach to secure the REST API is to use REST extensions by setting `rest.extension.classes`. We cover REST extensions in Chapter 8 and explain how the built-in `BasicAuthSecurityRestExtension` can be used to authenticate requests to the API. In a similar manner, you can build REST extensions that perform authorizations and only allow some users to access specific endpoints. You can also use this plugin to perform validation on requests. For example, you can ensure a connector always overrides SASL settings and provides its own identity, instead of relying on the identity of the Connect runtime. REST extensions are a very powerful plugin that you can use to deeply customize the REST API to satisfy your requirements.

Summary

In this chapter we looked at all the configuration settings Connect exposes, and provided some background and context to help you decide when to tune them. While most settings have good default values, it's important to understand what each setting is doing, to be able to adapt Connect to your specific use case.

We first looked at the runtime configurations and explained the main settings that are required to start Connect. On top of the basic required configurations we also recommended that you always set the `client.id`, `plugin.path` and `config.providers` configurations. When setting up the runtime for your use cases you can also set some more fine grained configurations that may be useful in particular scenarios.

In the second section we walked through the configurations for connectors. Source and sink connectors share many configurations, but they work differently when it comes to topics. We recommend you make conscious decisions about how topics are created and configured. You can also make use of more specific settings like client overrides, exactly once delivery and error handling configurations.

The last section was dedicated to configurations that are used to secure Connect clusters. Kafka supports the use of encrypted connections using TLS and has multiple options for authentication, from mutual TLS to various SASL mechanisms. You also have a choice of how to split permissions, and can use a single principal or connector specific principals. Finally we finished with an overview of the protections administrators can put in place to secure the REST API. The REST API can be overlooked when it comes to security, but since this is how you start and stop connectors it is essential that it is properly secured.

About the Authors

Mickael Maison is a committer and member of the Project Management Committer (PMC) for Apache Kafka. He has been contributing to Apache Kafka and its wider ecosystem since 2015.

Mickael is a software engineer with over 10 years of software development experience. While working at IBM, he was part of the Kafka team that runs hundreds of Kafka clusters for customers. He is now working in the Kafka team at Red Hat and has accumulated a lot of expertise about Kafka Connect.

In addition, Mickael has developed and contributed to several connectors for Connect. He also has deep expertise in Connect's internals, as he has made a number of code contributions to Connect itself and regularly reviews pull requests from the community on this component.

Finally, Mickael really enjoys sharing expertise and teaching. He has been writing monthly Kafka digests since 2018 and enjoys presenting at conferences.

Kate Stanley is a software engineer, technical speaker and Java Champion. She has experience running Apache Kafka on Kubernetes, developing enterprise Kafka applications and writing connectors for Kafka Connect.

Kate currently works as a Principal Software Engineer across the Red Hat Kafka offerings. She also contributes to multiple projects in the Kafka ecosystem, including the open-source Kafka operator, Strimzi. Kate started her journey with Kafka as part of the Event Streams team at IBM in 2018, quickly becoming well known in the community.

Alongside development, Kate has a passion for presenting and sharing knowledge. She is a regular speaker at technical conferences around the world, including events such as Kafka Summit, Jfokus, Devoxx UK and JavaOne. She has authored two LinkedIn Learning courses on MicroProfile and Kafka and written an eBook on writing microservices with Java.