Chapter 7. Building Data Pipelines

When people discuss building data pipelines using Apache Kafka, they usually have one of two use-cases. The first is to build a data pipeline where Apache Kafka is one of the two end-points. For example, getting data from Kafka to S3 or getting data from MongoDB into Kafka. The second use-case involves building a pipeline between two different systems but using Kafka as an intermediary. For example, getting data from Twitter to Elastic Search by sending the data first from Twitter to Kafka and then from Kafka to Elastic Search.

When we added Kafka Connect to Apache Kafka in version 0.9 it was after we've seen Kafka used in both use-cases at LinkedIn and other large organizations. We've noticed that there were specific challenges in integrating Kafka into data pipelines that every organization had to solve, and decided to add APIs to Kafka that will solve some of those challenges, rather than force every organization to figure them out from scratch.

The main value Kafka provides to data pipelines is its ability to serve as a very large, reliable buffer between various stages in the pipeline, effectively decoupling producers and consumers of data within the pipeline. This decoupling, combined with reliability security and efficiency, makes Kafka a good fit into most data pipelines.

NOTE

Some organizations think of Kafka as an **end point** of a pipeline. They look at problems such as "How do I get data from Kafka to Elastic". This is a valid question to ask - especially if there is data you need in Elastic and it is currently in Kafka, and we will look at ways to do exactly this. But we are going to start the discussion by looking the use of Kafka within a larger context that includes at least two (and possibly many more), end points which are not Kafka itself. We encourage anyone faced with a data integration problem to consider the bigger picture and not focus only on the immediate end-points - focusing on short-term integrations is how you end up with complex and expensive to maintain data integration mess.

In this chapter, we'll discuss some of the common issues that you need to take into account when building data pipelines. Those challenges are not specific to Kafka, but rather general data integration problems, we'll show why Kafka is a good fit for data integration use-cases and how it addresses many of those challenges. We'll discuss how the Kafka Connect APIs are different from the normal Producer and Consumer clients and when each client type should be used. Then we'll jump into some details of the Kafka Connect. A full user manual of Kafka Connect is outside the scope of the chapter, but we will show examples of basic usage to get you started and give you pointers on where to learn more. Finally, we'll discuss other data integration systems and how they integrate with Kafka.

Considerations When Building Data Pipelines

While going into all details of building data pipelines is outside the scope of this book, we would like to highlight some of the most important things we take into account when designing software architectures with the intent of integrating multiple systems.

Timeliness

Some systems expect their data to arrive in large bulks once a day, others expect the data to arrive few milliseconds after it is generated. Most data pipelines fit somewhere in between these two extremes. Good data integration systems can support different timeliness requirements for different pipelines and also make the migration between different time-tables easier as business requirements can change. Kafka, being a streaming data platform with scalable and reliable storage can be used to support anything from near-real-time pipelines to hourly batches. Producers can write to Kafka as frequently and infrequently as needed and consumers can also read and deliver the latest events as they arrive or they can wake up every hour and read all the events that arrived since and are stored in Kafka.

A useful way to look at Kafka in this context is that it acts as a giant buffer that decouples the time-sensitivity requirements between producers and consumers. Producers can be real-time while consumers batch or any other combination. This also makes it trivial to apply back-pressure - Kafka itself applies back-pressure on producers (by delaying acks when needed) and consumption rate is driven entirely by the consumers.

Reliability

we want to avoid single points of failure and allow for fast and automatic recovery from all sorts of failure events. Data pipelines are often the way data arrives to business critical systems, failure for more than few seconds can be hugely disruptive - especially when the timeliness requirement is closer to the “few milliseconds” end of the spectrum. Other important consideration for reliability is delivery guarantees - some systems can afford to lose data, but most of the time there is a requirement for “at-least once delivery”, which means every event from the source system will reach its destination, but sometimes retries will cause duplicates. Often, there is even a requirement for “exactly once delivery” - every event from source system will reach the destination with no possibility for loss or duplication.

We’ve discussed Kafka’s availability and reliability guarantees in depth in Chapter 6. As we’ve discussed, Kafka can provide “at least once” on its own, and “exactly once” when combined with an external data store that has a transactional model or unique keys. Since many of the end points are data stores that provide the right semantics for “exactly once” delivery, a Kafka based pipeline can often be implemented as “exactly once”. It is worth highlighting that Kafka’s Connect APIs make it easier for connectors to build end-to-end exactly once pipeline, by providing APIs for integrating with the external systems when handling offsets. Indeed many of the available open source connectors support exactly once delivery.

High and varying throughput

The data pipelines we are building should be able to scale to very high throughputs as often required in modern data systems. Even more important, they should be able to adapt in case throughput suddenly increases.

With Kafka acting as a buffer between producers and consumers, we no longer need to couple consumer throughput to the producer throughput. We no longer need to implement complex back pressure mechanism - if producer throughput exceeds that of the consumer, data will accumulate in Kafka until the consumer can catch up. Kafka’s ability to scale by adding consumers or producers independently allows us to scale either side of the pipeline dynamically and independently to match the changing requirements.

Kafka is a high throughput distributed system - capable of processing hundreds of megabytes per second on even modest clusters - so there is no concern that our pipeline will not scale as demand grows. In addition, Kafka Connect API focus on parallelizing the work and not just scaling it out - we’ll describe below how the platform allows data sources and sinks to split the work between multiple threads of execution and use the available CPU resources even when running on a single machine.

Kafka also supports several types of compression allowing users and admins to control the use of network and storage resources as the throughput requirements increase.

Data Formats

One of the most important considerations in a data pipeline is reconciling different data formats and data types. The data types supported vary between different databases and other storage systems. You may be loading XMLs and relational data into Kafka, using Avro within Kafka and then need to convert data to JSON when writing it to Elastic Search, to Parquet when writing to HDFS and to CSV when writing to S3.

Kafka itself and the connect APIs are completely agnostic to data formats. As we've seen in previous chapters, producers and consumers can use any serializer to represent data in any format that works for you. Kafka Connect has its own in-memory objects that include data types and schemas, but as we'll soon discuss, it allows for pluggable converters to allow storing these records in any format. This means that no matter which data format you chose for Kafka, it does not restrict your choice of connectors.

Many sources and sinks have a schema, we can read the schema from the source with the data, store it, and use it to validate compatibility or even update the schema in the sink database. A classic example is a data pipeline from MySQL to Hive. If someone added a column in MySQL, a great pipeline will make sure the column gets added to Hive too as we are loading new data into it.

In addition, Sink connectors, writing data from Kafka to external systems are responsible for the format in which the data is written to the external system. Some connectors choose to make this format pluggable. For example, the HDFS connector allows a choice between Avro and Parquet formats.

It is not enough to support different types of data, a generic data integration framework should also handle differences in behavior between various sources and sinks. For example, Syslog is a source that push data while relational databases requires the framework to pull data out. HDFS is append-only and we can only write data to it, while most systems allow us to both append data and update existing records.

Transformations

Transformations are more controversial than other requirements. There are generally two schools of building data pipelines, ETL and ELT. ETL stands for Extract-Transform-Load, meaning that the data pipeline is responsible for making modifications to the data as it passes through. It has the perceived benefit of saving time and storage, since you don't need to store the data, modify it and store it again. Depending on the transformations, this benefit is sometimes real, but sometimes shifts the burden of computation and storage to the data pipeline itself which may or may not be desirable. The main drawback of this approach is that the transformations that happen to the data in the pipeline tie the hands of those who wish to process the data farther down the pipe. If the person who built the pipeline between MongoDB and MySQL decided to filter certain events or remove fields from records, all the users and applications who access the data in MySQL will only have access to partial data. If they require access to the missing fields, the pipeline needs to be rebuilt and historical data will require re-processing (assuming it is available). ELT stands for Extract-Load-Transform. The data pipeline does only minimal transformation (mostly around data type conversion), with the goal of making sure the data that arrives at the target is as similar as possible to the source data. These are also called high-fidelity pipelines or data-lake architecture. In these systems, the target system collects "raw data" and all required processing is done at the target system. The benefit here is that it provides maximum flexibility to users of the target system, since they have access to all the data. These systems also tend to be easier to troubleshoot - since all data processing is limited to one system rather than split between the pipeline and additional applications. The drawback is that the transformations take cpu and storage resources at the target system, in some cases these systems are expensive and there is strong motivation to move computation off those systems when possible.

Security

Security is always a concern. In terms of data pipelines, the main security concerns are:

- Can we make sure the data going through the pipe is encrypted? This is mainly a concern for data pipelines that cross datacenter boundaries.
- Who is allowed to make modification to the pipelines?
- If the data pipeline needs to read or write from an access controlled locations, can it authenticate properly?

Kafka allow encrypting data on the wire, as it is piped from sources to Kafka and from Kafka to sinks. It also supports authentication (via SASL) and authorization - so you can be sure that if a topic contains sensitive information, it can't be piped into less secured systems by someone unauthorized. Kafka also provides an audit log to track access - unauthorized and authorized. With some extra

coding, it is also possible to track where did the events in each topic came from and who modified them, so you can provide entire lineage for each record.

Failure Handling

Assuming that all data will be perfect all the time is dangerous. It is important to plan for failure handling in advance. Can we prevent faulty records from ever making it into the pipeline? Can we recover from records that cannot be parsed? Can bad records get fixed (perhaps by a human) and re-processed? What if the bad event looks exactly like a normal event and you only discover the problem few days later?

Because Kafka stores all events for long periods of time, it is possible to go back in time and recover from errors when needed.

Coupling and Agility

One of the most important goals of data pipelines is to decouple the data sources and data targets. There are multiple ways accidental coupling can happen:

- Ad-hoc pipelines: Some companies end up building a custom pipeline for each pair of applications they want to connect. For example, they use Logstash to dump logs to Elastic Search, Flume to dump logs to HDFS, GoldenGate to get data from Oracle to HDFS, Informatica to get data from MySQL and XMLs to Oracle, and so on. This tightly couples the data pipeline to the specific end-points and creates a mess of integration points that require significant effort to deploy, maintain and monitor. It also means that ever new system the company adopts require building additional pipelines, increasing the cost of adopting new technology and inhibiting innovation.
- Loss of Metadata: If the data pipeline doesn't preserve schema metadata and does not allow for schema evolution, you end up tightly coupling the software producing the data at the source and the software that uses it at the destination. Without schema information, both software products need to include information on how to parse the data and interpret it. If data flows from Oracle to HDFS and a DBA added a new field in Oracle, without preserving schema information and allowing schema evolution, either every app that reads data from HDFS will break or all the developers will need to upgrade their applications at the same time. Neither option is agile. With support for schema evolution in the pipeline, each team can modify their applications at their own pace without worries that things will break down the line.
- Extreme Processing: As we mentioned when discussing data transformations, some processing of the data is inherent to data pipelines. After all, we are moving data between different systems where different data formats make sense and different use-cases are supported. However, too much processing ties all the downstream systems to decisions made when building the pipelines. Decisions about which fields to preserve, how to aggregate data, etc. This often leads to constant changes to the pipeline as requirements of downstream applications change, which isn't agile, efficient or safe. The more agile way is to preserve as much of the raw data as possible and allow downstream apps to make their own decisions regarding data processing and aggregation.

When to use Kafka Connect vs. Producer and Consumer

When writing to Kafka or reading from Kafka, you have the choice between using traditional producer and consumer clients, as described in Chapters 3 and 4, or to use the Connect APIs and the connectors as we'll describe below. Before we start diving into the details of Connect, it makes sense to stop and ask yourself: "When do I use which?"

As we've seen, Kafka clients are clients to be embedded in your own application. It allows your application to write data to Kafka or to read data from Kafka. As such, you will use Kafka clients when you are a developer, you want to connect an application to Kafka and can modify the code of the application, and you want to push data into Kafka or pull data from Kafka.

You will use Connect to connect Kafka to datastores that you did not write and can't or won't modify their code. Connect will be used to pull data from the external datastore into Kafka or push data from Kafka to an external store. For data stores where a connector already exists, Connect can be used by non-developers who will only need to configure the connectors.

If you need to connect Kafka to a data store and a connector does not exist yet, you can choose between writing an app using the Kafka clients or the Connect APIs. Connect is recommended because it provides out of the box features like configuration

management, offset storage, parallelization, error handling, support for different data types and standard management REST API. Writing a small app that connects Kafka to a data store sounds simple, but there are many little details you will need to handle around data types and configuration that make the task non-trivial - Kafka connect handles most of this for you, allowing you to focus on transporting data to and from the external stores.

Kafka Connect

Kafka Connect is a part of Apache Kafka and provides a scalable and reliable way to move data between Kafka and other data stores. It provides APIs and a runtime to develop and run `connector plugins` - libraries that Kafka Connect executes and which are responsible for moving the data. Kafka Connect runs as a cluster of `worker processes`, you install the connector plugins on the workers and then use a REST API to configure and manage `connectors`, which are long running jobs of connectors plugins running with specific configuration. Connectors start additional `tasks` to move large amounts of data in parallel and use the available resources on the worker nodes more efficiently. Source connector tasks just need to read data from the source system and provide connect data objects to the worker processes. Sink connector tasks get connector data objects from the workers and are responsible for writing them to the target data system. Kafka Connect uses `convertors` to support storing those data objects in Kafka in different formats - JSON format support is part of Apache Kafka, and the Confluent Schema Registry provides Avro converters. This allows users to choose the format in which data is stored in Kafka independently of the connectors they use.

This chapter cannot possible get into all the details of Kafka Connect and its many connectors. This could fill an entire book on its own. We will, however, give an overview of Kafka Connect and how to use it and point to additional resources for reference.

Running Connect

Kafka Connect ships with Apache Kafka, so there is no need to install it separately. For production use, especially if planning to use Connect to move large amounts of data or run many connectors, it is recommended to run Connect on separate servers than Kafka brokers. In this case, install Apache Kafka on all the machines, and simply start the brokers on some servers and start connect on other servers.

Starting a Connect worker is very similar to starting a broker, you call the start script with a properties file:

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

There are few key configurations for the Connect workers:

- bootstrap.servers - list of Kafka brokers that Connect will work with. Connectors will pipe their data either to or from those brokers. You don't need to specify every broker in the cluster, but its recommended to specify at least 3.
- group.id - all workers with the same group id are part of the same Connect cluster. A connector started on the cluster will run on any worker and so will its tasks.
- key.converter and value.converter - Connect can handle multiple data formats stored in Kafka. The two configurations set the converter for the key and value part of the message that will be stored in Kafka. The default is JSON format using the JSONConverter included in Apache Kafka. These configurations can also be set to AvroConverter, which is part of the Confluent Schema Registry.

  Some converters include converter-specific configuration parameters. For example, JSON messages can include a schema or be schema-less. To support either, you can set key.converter.schema.enable=true or false, respectively. Same configuration can be done for the value converter, but setting value.converter.schema.enable. Avro messages also contain a schema, but you need to configure the location of the Schema Registry using key.converter.schema.registry.url and value.converter.schema.registry.url.
- rest.host.name and rest.port - Connectors are typically configured and monitored through the REST API of KafkaConnect. You can configure the specific port for the REST API.

  Once the workers are up and you have a cluster, make sure it is up and running by checking the REST API:

```
gwen$ curl http://localhost:8083/
{"version":"0.10.1.0-SNAPSHOT","commit":"561f45d747cd2a8c"}
```

Accessing the base REST URI should return the current version you are running. I am running a snapshot of Kafka 0.10.1.0 (pre-release). We can also check which connector plugins are available:

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

[{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"},{"class":"org.apache.kafka.connect.file.FileStrea
```

I am running plain Apache Kafka, so the only available connector plugins are the File source and File sink.

Lets see how to configure and use these example connectors, and then we'll dive into more advanced examples that require setting up external data systems to connect to.

> **NOTE**
>
> Stand-alone mode Take note that Kafka Connect also has stand-alone mode. It is similar to distributed mode, you just run `bin/connect-standalone.sh` instead of `bin/connect-distributed.sh`. You can also pass in connector configuration file on the command line instead of through the REST API. In this mode all the connectors and tasks run on the one stand-alone worker. This is usually easier to use connect in stand-alone mode for development and troubleshooting as well as cases where connectors and tasks need to run on a specific machine (Syslog connector for example listens on a port, so you need to know which machines it is running on)

Connectors Example - File source and File sink

This example will use the File connectors and JSON converter that are part of Apache Kafka. To follow along, make sure you have Zookeeper and Kafka up and running.

To start, lets run a distributed connect worker. In a real production environment you'll want at least 2 or 3 of these running, to provide high availability. In this example, I'll only start one.

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

Now its time to start a file source. As an example, I will configure it to read the Kafka configuration file - basically piping Kafka's configuration into a Kafka topic:

```
echo '{"name":"load-kafka-config", "config":{"connector.class":"FileStreamSource","file":"config/server.properties","t

{"name":"load-kafka-config","config":{"connector.class":"FileStreamSource","file":"config/server.properties","topic":"
```

To create a connector I wrote a JSON that includes a connector name, "load-kafka-config" and the connector configuration map, which includes the connector class, the file I want to load and the topic I want to load the file into.

Lets use the Kafka Console consumer to check that we have the loaded the configuration into a topic.

```
gwen$ bin/kafka-console-consumer.sh --new --bootstrap-server=localhost:9092 --topic kafka-config-topic --from-beginnin
```

If all went well, you should see something along the lines of:

```
{"schema":{"type":"string","optional":false},"payload":"# Licensed to the Apache Software Foundation (ASF) under one o

<more stuff here>

{"schema":{"type":"string","optional":false},"payload":"############################## Server Basics ##################
{"schema":{"type":"string","optional":false},"payload":""}
{"schema":{"type":"string","optional":false},"payload":"# The id of the broker. This must be set to a unique integer f
{"schema":{"type":"string","optional":false},"payload":"broker.id=0"}
{"schema":{"type":"string","optional":false},"payload":""}

<more stuff here>
```

This is literally the contents of config/server.properties file, as it was converted to JSON records line-by-line and placed in kafka-config-topic by our connector. Note that by default the JSON converter places a schema in each record. In this specific case the schema is very simple - there is only a single column, named "payload" of type "string" and it contains a single line from the file for each record.

Now lets use the file sink converter to dump the contents of that topic into a file. The resulting file should be completely identical to the original server.properties file, as the JSON converter will convert the JSON records back into simple text lines.

```
echo '{"name":"dump-kafka-config", "config":{"connector.class":"FileStreamSink","file":"copy-of-server-properties","to

{"name":"dump-kafka-config","config":{"connector.class":"FileStreamSink","file":"copy-of-server-properties","topics":"
```

Note the changes from the source configuration: The class we are using is now FileStreamSink rather than FileStreamSource, we still have a file property but now it refers to the destination file rather than the source of the records, and instead of specifying a "topic" you specify "topics". Note the plurality - you can write multiple topics into one file with the sink, while the source only allows writing into one topic.

If all went well, you should have a file named "copy-of-server-properties" which is completely identical to config/server.properties that we used to populate kafka-config-topic.

To delete a connector, you can run:

```
curl -X DELETE http://localhost:8083/connectors/dump-kafka-config
```

If you look at the connect worker log after deleting a connector, you should see all other connectors restarting their tasks. This is to rebalance the remaining tasks between the workers and ensure equivalent workloads after a connector was removed.

Connectors Example - MySQL to ElasticSearch

Now that we got a simple example working, lets do something more useful. Lets take a MySQL table, stream it to a Kafka topic and from there load it to ElasticSearch and index its contents.

I am running my tests on my macbook. To install MySQL and ElasticSearch, I simply ran:

```
brew install mysql
brew install elasticsearch
```

The next step is to make sure you have the connectors. If you are running Confluent OpenSource, you should have the connectors already installed as part of the platform. Otherwise you can just build the connectors from github:

- Go to *https://github.com/confluentinc/kafka-connect-elasticsearch*
- Clone the repository

- Run `mvn install` to build the project
- Repeat with the JDBC connector: *https://github.com/confluentinc/kafka-connect-jdbc*

Now take the jars that were created under the `target` directory where you built each connector and copy them into Kafka Connect's class path.

```
gwen$ mkdir libs
gwen$ cp ../kafka-connect-jdbc/target/kafka-connect-jdbc-3.1.0-SNAPSHOT.jar libs/
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-elasticsearch-3.2.0-SNAPSHOT-package/share/java/kafka-con
```

If the Kafka Connect workers are not already running, make sure to start them, and check that the new connector plugins are listed:

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

gwen$  curl http://localhost:8083/connector-plugins
[{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"},
{"class":"io.confluent.connect.elasticsearch.ElasticsearchSinkConnector"},
{"class":"org.apache.kafka.connect.file.FileStreamSinkConnector"},
{"class":"io.confluent.connect.jdbc.JdbcSourceConnector"}]
```

We can see that we now have additional connector plugins available in our connect cluster. The JDBC source requires a MySQL driver in order to work with MySQL. We downloaded the JDBC driver for MySQL from Oracle website, unzipped the package and copied mysql-connector-java-5.1.40-bin.jar to libs/ directory when we copied the connectors.

The next step is to create a table in MySQL that we can stream into Kafka using our JDBC connector.

```
gwen$ mysql.server restart

mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> use test;
Database changed
mysql> create table login (username varchar(30), login_time datetime);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into login values ('gwenshap', now());
Query OK, 1 row affected (0.01 sec)

mysql> insert into login values ('tpalino', now());
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

As you can see, we created a database, a table and inserted few rows as an example.

Next step is to configure our JDBC source connector. We can find out which configuration options are available by looking at the documentation, but we can also use the REST API to find out:

```
gwen$ curl -X PUT -d "{}" localhost:8083/connector-plugins/JdbcSourceConnector/config/validate --header "content-Type:

{
    "configs": [
        {
            "definition": {
                "default_value": "",
                "dependents": [],
                "display_name": "Timestamp Column Name",
                "documentation": "The name of the timestamp column to use to detect new or modified rows. This column
                "group": "Mode",
                "importance": "MEDIUM",
                "name": "timestamp.column.name",
```

```
            "order": 3,
            "required": false,
            "type": "STRING",
            "width": "MEDIUM"
        },
        <more stuff>
```

We basically asked the REST API to validate configuration for a connector and sent it an empty configuration. As a response, we got the JSON definition of all available configurations. I piped the output through python to make the JSON more readable.

With this information in mind, time to create and configure our JDBC connector:

```
echo '{"name":"mysql-login-connector", "config":{"connector.class":"JdbcSourceConnector","connection.url":"jdbc:mysql:


{"name":"mysql-login-connector","config":{"connector.class":"JdbcSourceConnector","connection.url":"jdbc:mysql://127.0
```

Lets make sure it worked by reading data from the mysql.login topic:

```
gwen$ bin/kafka-console-consumer.sh --new --bootstrap-server=localhost:9092 --topic mysql.login --from-beginning

<more stuff>

{"schema":{"type":"struct","fields":[{"type":"string","optional":true,"field":"username"},{"type":"int64","optional":t
{"schema":{"type":"struct","fields":[{"type":"string","optional":true,"field":"username"},{"type":"int64","optional":t
```

If you get errors saying the topic doesn't exist, or you see no data - check the connect worker logs for errors such as:

```
[2016-10-16 19:39:40,482] ERROR Error while starting connector mysql-login-connector (org.apache.kafka.connect.runtime
org.apache.kafka.connect.errors.ConnectException: java.sql.SQLException: Access denied for user 'root;'@'localhost' (u
        at io.confluent.connect.jdbc.JdbcSourceConnector.start(JdbcSourceConnector.java:78)
```

It took me multiple attempts to get the connection string right. Other issues can involve the existence of the driver in the classpath or permissions to read the table.

Note that while the connector is running, if you insert additional rows to the *login* table, you should immediately see them reflected in the *mysql.login* topic.

Getting MySQL data to Kafka is useful in itself, but lets make things more fun by writing the data to Elastic Search.

First, we start ElasticSearch and verify it is up by accessing its local port:

```
gwen$ elasticsearch &
gwen$ curl http://localhost:9200/
{
  "name" : "Hammerhead",
  "cluster_name" : "elasticsearch_gwen",
  "cluster_uuid" : "42D5GrxOQFebf83DYgNl-g",
  "version" : {
    "number" : "2.4.1",
    "build_hash" : "c67dc32e24162035d18d6fe1e952c4cbcbe79d16",
    "build_timestamp" : "2016-09-27T18:57:55Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.2"
  },
  "tagline" : "You Know, for Search"
}
```

Now lets start the connector:

```
echo '{"name":"elastic-login-connector", "config":{"connector.class":"ElasticsearchSinkConnector","connection.url":"ht

{"name":"elastic-login-connector","config":{"connector.class":"ElasticsearchSinkConnector","connection.url":"http://lo
```

There are few configurations we need to explain here. The `connection.url` is simply the url of the local Elasticsearch server we configured earlier. Each topic in Kafka will become, by default, a separate Elasticsearch index, with the same name as the topic. Within the topic, we need to define a type for the data we are writing. We assume all the events in a topic will be of the same type, so we just hardcode `type.name=mysql-data`. The only topic we are writing to Elasticsearch is mysql.login, and because back when we defined the table in MySQL we didn't give it a primary key, the events in Kafka have null key, and we need to tell the Elasticsearch connector to use the topic name, partition id and offset as the key for each event instead. We do that by setting `key.ignore=true`.

Lets check that the index with mysql.login data got created:

```
gwen$ curl 'localhost:9200/_cat/indices?v'
health status index       pri rep docs.count docs.deleted store.size pri.store.size
yellow open   mysql.login   5   1          3            0     10.7kb         10.7kb
```

If the index isn't there, look for errors in the connect worker log. Missing configuration or libraries are common causes for errors. If all is well, we can search the index for our records:

```
gwen$ curl -s -X "GET" "http://localhost:9200/mysql.login/_search?pretty=true"
{
  "took" : 29,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "mysql.login",
      "_type" : "mysql-data",
      "_id" : "mysql.login+0+1",
      "_score" : 1.0,
      "_source" : {
        "username" : "tpalino",
        "login_time" : 1476423981000
      }
    }, {
      "_index" : "mysql.login",
      "_type" : "mysql-data",
      "_id" : "mysql.login+0+2",
      "_score" : 1.0,
      "_source" : {
        "username" : "nnarkede",
        "login_time" : 1476672246000
      }
    }, {
      "_index" : "mysql.login",
      "_type" : "mysql-data",
      "_id" : "mysql.login+0+0",
      "_score" : 1.0,
      "_source" : {
        "username" : "gwenshap",
        "login_time" : 1476423962000
      }
    } ]
  }
}
```

If you add new records to the table in MySQL, they will automatically appear in mysql.login topic in Kafka and in the corresponding Elasticsearch index.

Now that we've seen how to build and install the JDBC Source and Elasticsearch Sink, you can build and use any pair of connectors that suites your use case. Confluent maintains a list of all connectors (*http://www.confluent.io/product/connectors/* (http://www.confluent.io/product/connectors/)) we know about - both those written and supported by companies and community connectors. You can pick any connector on the list that you wish to try out, build it from the github repository, configure - either based on the documentation or by pulling the configuration from the REST API, and run it on your connect worker cluster.

> **NOTE**
>
> Build your Own Connectors The Connector APIs are public and anyone can create a new connector. In fact, this is how most of the connectors arrived to the Connector Hub - different people built connectors and told us about them. So if the data store you wish to integrate with is not available in the hub, we encourage you to write your own. You can even contribute it to the community so others can discover and use it. It is beyond the scope of this chapter to discuss all the details involved in building a connector, but you can learn about it in the official documentation (*http://docs.confluent.io/3.0.1/connect/devguide.html* (http://docs.confluent.io/3.0.1/connect/devguide.html)). We also recommend looking at the existing connectors as starting point and perhaps jumpstart using a maven archtype (*https://github.com/jcustenborder/kafka-connect-archtype*). We always encourage you to ask for assistance or show off your latest connectors at the Apache Kafka community mailing lists (users@kafka.apache.org)

A Deeper Look at Connect

To understand how Connect works, you need to understand three basic concepts and how they interact. As we've explained earlier and demonstrated with examples - to use Connect you need to run a cluster of **workers** and start/stop **connectors**. An additional detail we did not dive into before is the handling of data by **convertors** - these are the components that converted MySQL rows to JSON records, which the connector wrote into Kafka.

Lets look a bit deeper into each system and how they interact with each other.

CONNECTORS AND TASKS

Connector plugins implement the connector API which includes two parts:

Connector

The connector is responsible for three important things:

- how many tasks will run for the connector
- how to split the data copying work between the tasks
- getting configuration for the tasks from the workers and passing it along.

For example, the JDBC source connector will connect to the database, discover the existing tables to copy and based on that decide how many tasks are needed - the lower of max.tasks configuration and the number of tables. Once it decides how many tasks will run, it will generate a configuration for each task - using both the connector configuration (connection.url for example) and a list of tables it assigns for each task to copy. The taskConfigs() method returns a list of maps - a configuration for each task we want to run. The workers are then responsible for starting the tasks and giving each its own unique configuration so they will each copy a unique subset of tables from the database. Note that when you start the connector via the REST API, it may start on any node and subsequently the tasks it starts may also execute on any node.

Tasks

Tasks are responsible for actually getting the data in and out of Kafka. All tasks are initialized by receiving a context from the worker: Source context includes an object that allows the source task to store offsets of source records (for example in the File connector the offsets are positions in the file, in JDBC source connector the offsets can be primary key IDs in a table). Context for the sink connector includes methods that allow the connector to control the records it is receiving from Kafka - this is used for things like applying back pressure, retrying and storing offsets externally for exactly-once delivery. After tasks are initialized, the are started with a Properties object that contains the configuration the Connector created for the task. Once tasks are started Source tasks poll an external system and return lists of records that the worker sends to Kafka brokers and Sink tasks receive records from Kafka through the worker and are responsible for writing the records to an external system.

WORKERS

Kafka Connect's worker processes are the "container" processes that execute the connectors and tasks. They are responsible for handling the HTTP requests that define connectors and their configuration, they are responsible for storing the connector configuration, starting the connectors and their tasks and passing the appropriate configurations along. If a worker process is stopped or crashes, other workers in a Connect cluster will recognize that (using the heartbeats in Kafka's consumer protocol) and will reassign the connectors and tasks that ran on that worker to the remaining workers. If a new worker joins a Connect cluster, other workers will notice that too and assign connectors or tasks to the new worker to make sure load is balanced among all workers fairly. Workers are also responsible for automatically committing offsets for both source and sink connectors and for handling retries when tasks throw errors.

The best way to understand workers is to realize that Connectors and Tasks are responsible for the "moving data" part of data integration, while the workers are responsible for the REST API, configuration management, reliability, high availability, scaling and load balancing.

This separation of concerns is the main benefit of using Connect APIs vs the classic consumer/producer APIs. Experienced developers know that writing code that reads data from Kafka and inserts it into a database takes maybe a day or two, but if you need to handle configuration, errors, REST APIs, monitoring, deployment, scaling up and down, handling failures… this can take few month to get right. If you implement data copying with a connector, your connector plugs into workers that handle a bunch of quite complicated operational issues that you don't need to worry about.

CONVERTERS AND CONNECT'S DATA MODEL

The last piece of the Connect APIs puzzle is the Connector data model and the Converters. Kafka's Connect APIs includes a data API - which includes both data objects and a schema that describes that data. For example, the JDBC source reads a column from a database and constructs a Connect Schema object based on the data types of the columns returned by the database. It then uses the schema to construct a Struct that contains all the fields in the database record - for each column we store the column name and the value in that column. Every source connector does something similar - read an event from the source system and generate a pair of Schema and Value (the value is the data objects themselves). Sink connectors do the opposite - get a Schema and Value pair and use the schema to parse the values and insert them into the target system.

Because source connectors just know how to generate objects based on the Data API, there is still a question of how Connect workers store these objects in Kafka. This is where the converters come in. When users configure the worker (or the connector), they choose which Converter they want to use to store data in Kafka. At the moment the available choices are Avro, JSON or String. JSON converter can be configured to either include a schema in the result record or not include one - so we can support both structured and semi-structured data. When the Connector returns a Data API record to the worker, the worker than uses the configured converter to convert the record to either Avro object, JSON object or a string - and the result is then stored into Kafka.

The opposite process happens for sink connectors - when the Connect worker reads a record from Kafka, it uses the configured Converter to convert the record from the format in Kafka (Avro, JSON or String) to the Connect Data API record and then passes these records to the sink connector that knows how to insert them into the destination system.

This allows the Connect API to support different types of data stored in Kafka, independently of the connector implementation - i.e. any connector can be used with any record type, as long as a converter is available.

OFFSET MANAGEMENT

Offset management is one of the convenient services that the workers perform for the connectors (in addition to deployment and configuration management via the REST API). The idea is that connectors need to know which data they already processed, and they can use APIs provided by Kafka to maintain those.

For source connectors, this means that the records that the connector returns to the Connect workers include a logical partition and a logical offset. Those are not Kafka partitions and Kafka offsets, but rather partitions and offsets as relevant in the source system. For example in the file source, a partition can be a file and an offset can be a line number or character number in the file. In a JDBC source a partition can be a database table and the offset can be an ID of a record in the table. One of the most important design decisions involved in writing a source connector is deciding on a good way to partition the data in the source system and decide on a good way to track offsets - this will impact the level of parallelism the connector can achieve and whether it can deliver at-least once or exactly once semantics.

When the source connector returns a list of records, which include the source partition and offset for each record, the worker sends the records to Kafka brokers. If the brokers successfully acknowledge the records, the worker then stores the offsets of the records it sent to Kafka. The storage mechanism is pluggable and is usually a Kafka topic. This allows connectors to start processing events from the most recent stored offset after a restart or a crash.

Sink connectors have an opposite, but similar workflow: They read Kafka records, which already have a topic, partition and offset identifiers. Then they call the Connector `put()` method that should store those records in the destination system and if the Connector reports success, they commit the offsets they've given to the Connector back to Kafka, using the usual Consumer commit methods.

Offset tracking provided by the framework itself should make it easier for developers to write connectors and guarantee some level of consistent behavior when using different connectors.

Alternatives to Kafka Connect

So far we've looked at Kafka's Connect APIs in great detail - we've seen how to use them and we looked at some details of how they work. While we love the convenience and reliability that the Connect APIs provide, they are far from being the only method for getting data in and out of Kafka. Lets look at other alternatives and when they are commonly used:

Ingest frameworks for other data stores

While we like to think that Kafka is the center of the universe, some people disagree. Some people build most of their data architectures around systems like Hadoop or ElasticSearch. Those systems have their own data ingestion tools - Flume for Hadoop and Logstash or Fluentd for ElasticSearch. We recommend Kafka's Connect APIs when Kafka is an integral part of the architecture and when the goal is to connect large numbers or sources and sinks. If you are actually building an Hadoop-centric or Elastic-centric system and Kafka is just one of many inputs into that system, then using Flume or Logstash makes sense.

GUI-based ETL tools

From old-school systems like Informatica, to open-source alternatives like Talend and Pentaho and even newer alternatives such as Apache NiFi and StreamSets - all those ETL solutions also support Apache Kafka as both a data source and a destination. Using these systems makes sense if you are already using them - if you already do everything using Pentaho for example, you may not be interested in adding another data integration system just for Kafka. They also make sense if you are used to GUI-based approach to building ETL pipelines. The main drawback of these systems is that they are usually built for involved workflows and will be a somewhat heavy and involved solution if all you want to do is get data in and out of Kafka. As mentioned in the introductory part of the chapter, we believe that data integration should focus on faithful transmition of messaging under all conditions, so most ETL tools include quite a bit more complexity.

Stream Processing Frameworks

Almost all stream processing frameworks include the ability to read events from Kafka and to write them to few other systems. If your destination system is supported and you are already intending to use that stream processing framework to process events from Kafka, it seems reasonable to use the same framework for data integration as well. This does often saves a step in the stream processing workflow (no need to store processed events in Kafka just to read them out and write them to another system), with the

drawback that it can be more difficult to troubleshoot things like lost and corrupted messages - there is less ability to track when did the messages get lost of corrupt.

Summary

In this chapter we discussed the use of Kafka for data integration. Starting with reasons to use Kafka in this manner, we covered general considerations for data integration solutions. We showed why we think Kafka and its Connect APIs are a good fit. We then gave several examples of how to use Kafka Connect in different scenarios, spent some time looking at the details of how Connect works and then discussed few alternatives to Kafka Connect.

Whatever data integration solution you eventually land with, the most important feature will always be its ability to deliver all messages under all failure conditions. We believe that Kafka Connect is extremely reliable - based on its integration with Kafka's tried and true reliability features - however it is important that you will test the system of your choice, just like we do. Make sure that your data integration system of choice can survive stopped processes, crashed machines, network delays and high loads without missing a message. After all - data integration systems only have one job, delivering those messages.

Of course, while reliability is usually the most important requirement when integrating data systems, it is only one requirement. When choosing a data system it is important to first review your requirements (refer to the considerations section at the beginning of this chapter for some examples) and then make sure your system of choice satisfies them. But this isn't enough - you must learn your data integration solution well enough to be certain that you are using it in a way that supports your requirements. It isn't enough that Kafka supports at least once semantics, you must be sure you aren't accidentally configuring it in a way that may end up with less than complete reliability.

PREV
6. Reliable Data Delivery

NEXT
8. Cross-Cluster Data Mirroring