Confluent Hub has Kafka connectors for the following messaging applications:

* Kafka Connect RabbitMQ
* Kafka Connect RabbitMQ Sink Connector
* Kafka Connect IBM MQ Source
* Kafka Connect IBM MQ Sink
* Kafka Connect ActiveMQ Source
* Kafka Connect ActiveMQ Sink

The Source Connector is for copying data from the MQ application into Apache Kafka and the Sink Connector is for copying data from Apache Kafka into the MQ application

**Kafka Connect RabbitMQ**: Kafka Source Connector for RabbitMQ

**Kafka Connect RabbitMQ Sink Connector**: Kafka Sink Connector for RabbitMQ

**Kafka Connect IBM MQ Source** : The IBM MQ Source Connector is used to read messages from an IBM MQ cluster and write them to a Kafka topic.

**Kafka Connect IBM MQ Sink**: The Kafka Connect IBM MQ Sink Connector integrates Kafka with IBM MQ. The connector consumes records from Kafka topic(s) and converts each record value to either a JMS TextMessage or BytesMessage before producing the JMS Message to IBM MQ.

Prerequisites include:

Kafka Broker: Confluent Platform 3.3.0 or above, or Kafka 0.11.0 or above

Kafka Connect: Confluent Platform 4.1.0 or above, or Kafka 1.1.0 or above (requires header support in Connect)

Java 1.8

**Kafka Connect ActiveMQ Source**: The ActiveMQ Source Connector is used to read messages from an ActiveMQ cluster and write them to a Kafka topic.

**The Kafka Connect ActiveMQ Sink**: The Kafka Connect ActiveMQ Sink Connector integrates Kafka with Apache ActiveMQ. The connector consumes records from Kafka topic(s) and converts each record value to either a JMS TextMessage or BytesMessage

before producing the JMS Message to ActiveMQ.

Prerequisites include:

1. Kafka Broker: Confluent Platform 3.3.0 or above, or Kafka 0.11.0 or above
2. Kafka Connect: Confluent Platform 4.1.0 or above, or Kafka 1.1.0 or above (requires header support in Connect)
3. Java 1.8

We are going to focus more on Kafka Connect IBM MQ Source and Kafka Connect IBM MQ Sink.

# ***Kafka Connect source connector for IBM MQ***

kafka-connect-mq-source is a [Kafka Connect](http://kafka.apache.org/documentation.html#connect) source connector for copying data from IBM MQ into Apache Kafka.

The connector is supplied as source code which we can easily build into a JAR file.

## Building the connector

To build the connector, we must have the following installed:

* [git](https://git-scm.com/)
* [Maven 3.0 or later](https://maven.apache.org)
* Java 8 or later

Clone the repository with the following command:

git clone https://github.com/ibm-messaging/kafka-connect-mq-source.git

Change directory into the kafka-connect-mq-source directory:

cd kafka-connect-mq-source

Build the connector using Maven:

mvn clean package

Once built, the output is a single JAR called target/kafka-connect-mq-source-<version>-jar-with-dependencies.jar which contains all of the required dependencies.

## Running the connector

For step-by-step instructions, see the following guides for running the connector:

* connecting to Apache Kafka [running locally](https://github.com/ibm-messaging/kafka-connect-mq-source/blob/master/UsingMQwithKafkaConnect.md)
* connecting to an installation of [IBM Event Streams](https://ibm.github.io/event-streams/connecting/mq/source)

To run the connector, we must have:

* The JAR from building the connector
* A properties file containing the configuration for the connector
* Apache Kafka 2.0.0 or later, either standalone or included as part of an offering such as IBM Event Streams
* IBM MQ v8 or later, or the IBM MQ on Cloud service

The connector can be run in a Kafka Connect worker in either standalone (single process) or distributed mode. It's a good idea to start in standalone mode.

### Running in standalone mode

We need two configuration files, one for the configuration that applies to all of the connectors such as the Kafka bootstrap servers, and another for the configuration specific to the MQ source connector such as the connection information for our queue manager. For the former, the Kafka distribution includes a file called connect-standalone.properties that we can use as a starting point. For the latter, we can use config/mq-source.properties in this repository.

The connector connects to MQ using either a client or a bindings connection. For a client connection, we must provide the name of the queue manager, the connection name (one or more host/port pairs) and the channel name. In addition, we can provide a user name and password if the queue manager is configured to require them for client connections. If we look at the supplied config/mq-source.properties, we'll see how to specify the configuration required. For a bindings connection, we must provide provide the name of the queue manager and also run the Kafka Connect worker on the same system as the queue manager.

To run the connector in standalone mode from the directory into which we installed Apache Kafka, we use a command like this:

bin/connect-standalone.sh connect-standalone.properties mq-source.properties

### Running in distributed mode

We need an instance of Kafka Connect running in distributed mode. The Kafka distribution includes a file called connect-distributed.properties that we can use as a starting point, or follow [Running with Docker](https://github.com/ibm-messaging/kafka-connect-mq-source#running-with-docker) or [Deploying to Kubernetes](https://github.com/ibm-messaging/kafka-connect-mq-source#deploying-to-kubernetes)

To start the MQ connector, we can use config/mq-source.json in this repository after replacing all placeholders and use a command like this:

curl -X POST -H "Content-Type: application/json" http://localhost:8083/connectors \

--data "@./config/mq-source.json"

## Running with Docker

This repository includes an example Dockerfile to run Kafka Connect in distributed mode. It also adds in the MQ source connector as an available connector plugin. It uses the default connect-distributed.properties and connect-log4j.properties files.

1. mvn clean package
2. docker build -t kafkaconnect-with-mq-source:1.3.0 .
3. docker run -p 8083:8083 kafkaconnect-with-mq-source:1.3.0

**NOTE:** To provide custom properties files create a folder called config containing the connect-distributed.properties and connect-log4j.properties files and use a Docker volume to make them available when running the container like this:

docker run -v $(pwd)/config:/opt/kafka/config -p 8083:8083 kafkaconnect-with-mq-source:1.3.0

To start the MQ connector, we can use config/mq-source.json in this repository after replacing all placeholders and use a command like this:

curl -X POST -H "Content-Type: application/json" http://localhost:8083/connectors \

--data "@./config/mq-source.json"

## Deploying to Kubernetes

This repository includes a Kubernetes yaml file called kafka-connect.yaml. This will create a deployment to run Kafka Connect in distributed mode and a service to access the deployment.

The deployment assumes the existence of a Secret called connect-distributed-config and a ConfigMap called connect-log4j-config. These can be created using the default files in our Kafka install, however it is easier to edit them later if comments and whitespaces are trimmed before creation.

### Creating Kafka Connect configuration Secret and ConfigMap

Create Secret for Kafka Connect configuration:

1. cp kafka/config/connect-distributed.properties connect-distributed.properties.orig
2. sed '/^#/d;/^[[:space:]]\*$/d' < connect-distributed.properties.orig > connect-distributed.properties
3. kubectl -n <namespace> create secret generic connect-distributed-config --from-file=connect-distributed.properties

Create ConfigMap for Kafka Connect Log4j configuration:

1. cp kafka/config/connect-log4j.properties connect-log4j.properties.orig
2. sed '/^#/d;/^[[:space:]]\*$/d' < connect-log4j.properties.orig > connect-log4j.properties
3. kubectl -n <namespace> create configmap connect-log4j-config --from-file=connect-log4j.properties

### Creating Kafka Connect deployment and service in Kubernetes

**NOTE:** We will need to [build the Docker image](https://github.com/ibm-messaging/kafka-connect-mq-source#running-with-docker) and push it to our Kubernetes image repository. Remember that the supplied Dockerfile is just an example and we will have to modify it for our needs. We might need to update the image name in the kafka-connect.yaml file.

1. Update the namespace in kafka-connect.yaml
2. kubectl -n <namespace> apply -f kafka-connect.yaml
3. curl <serviceIP>:<servicePort>/connector-plugins to see whether the MQ source connector is available to use

### Deploying to OpenShift using Strimzi

This repository includes a Kubernetes yaml file called strimzi.kafkaconnector.yaml for use with the [Strimzi](https://strimzi.io) operator. Strimzi provides a simplified way of running the Kafka Connect distributed worker, by defining either a KafkaConnect resource or a KafkaConnectS2I resource.

The KafkaConnectS2I resource provides a nice way to have OpenShift do all the work of building the Docker images for you. This works particularly nicely combined with the KafkaConnector resource that represents an individual connector.

The following instructions assume we are running on OpenShift and have Strimzi 0.16 or later installed.

#### Start a Kafka Connect cluster using KafkaConnectS2I

1. Create a file called kafka-connect-s2i.yaml containing the definition of a KafkaConnectS2I resource. We can use the examples in the Strimzi project to get started.
2. Configure it with the information it needs to connect to our Kafka cluster. We must include the annotation strimzi.io/use-connector-resources: "true" to configure it to use KafkaConnector resources so we can avoid needing to call the Kafka Connect REST API directly.
3. oc apply -f kafka-connect-s2i.yaml to create the cluster, which usually takes several minutes.

#### Add the MQ source connector to the cluster

1. mvn clean package to build the connector JAR.
2. mkdir my-plugins
3. cp target/kafka-connect-mq-source-\*-jar-with-dependencies.jar my-plugins
4. oc start-build <kafkaconnectClusterName>-connect --from-dir ./my-plugins to add the MQ source connector to the Kafka Connect distributed worker cluster. Wait for the build to complete, which usually takes a few minutes.
5. oc describe kafkaconnects2i <kafkaConnectClusterName> to check that the MQ source connector is in the list of available connector plugins.

#### Start an instance of the MQ source connector using KafkaConnector

1. cp deploy/strimzi.kafkaconnector.yaml kafkaconnector.yaml
2. Update the kafkaconnector.yaml file to replace all of the values in <>, adding any additional configuration properties.
3. oc apply -f kafkaconnector.yaml to start the connector.
4. oc get kafkaconnector to list the connectors. We can use oc describe to get more details on the connector, such as its status.

## Data formats

Kafka Connect is very flexible but it's important to understand the way that it processes messages to end up with a reliable system. When the connector encounters a message that it cannot process, it stops rather than throwing the message away. Therefore, we need to make sure that the configuration we use can handle the messages the connector will process.

This is rather complicated and it's likely that a future update of the connector will simplify matters.

Each message in Kafka Connect is associated with a representation of the message format known as a *schema*. Each Kafka message actually has two parts, key and value, and each part has its own schema. The MQ source connector does not currently make much use of message keys, but some of the configuration options use the word *Value* because they refer to the Kafka message value.

When the MQ source connector reads a message from MQ, it chooses a schema to represent the message format and creates an internal object called a *record* containing the message value. This conversion is performed using a *record builder*. Each record is then processed using a *converter* which creates the message that's published on a Kafka topic.

There are two record builders supplied with the connector, although we can write our own. The basic rule is that if we just want the message to be passed along to Kafka unchanged, the default record builder is probably the best choice. If the incoming data is in JSON format and we want to use a schema based on its structure, use the JSON record builder.

There are three converters built into Apache Kafka. We need to make sure that the incoming message format, the setting of the mq.message.body.jms configuration, the record builder and converter are all compatible. By default, everything is just treated as bytes but if we want the connector to understand the message format and apply more sophisticated processing such as single-message transforms, you'll need a more complex configuration. The following table shows the basic options that work.

| Record builder class | Incoming MQ message | mq.message.body.jms | Converter class | Outgoing Kafka message |
| --- | --- | --- | --- | --- |
| com.ibm.eventstreams.connect.mqsource.builders.DefaultRecordBuilder | Any | false (default) | org.apache.kafka.connect.converters.ByteArrayConverter | **Binary data** |
| com.ibm.eventstreams.connect.mqsource.builders.DefaultRecordBuilder | JMS BytesMessage | true | org.apache.kafka.connect.converters.ByteArrayConverter | **Binary data** |
| com.ibm.eventstreams.connect.mqsource.builders.DefaultRecordBuilder | JMS TextMessage | true | org.apache.kafka.connect.storage.StringConverter | **String data** |
| com.ibm.eventstreams.connect.mqsource.builders.JsonRecordBuilder | JSON, may have schema | Not used | org.apache.kafka.connect.json.JsonConverter | **JSON, no schema** |

There's no single configuration that will always be right, but here are some high-level suggestions.

* Pass unchanged binary (or string) data as the Kafka message value

value.converter=org.apache.kafka.connect.converters.ByteArrayConverter

* Message format is MQSTR, pass string data as the Kafka message value

mq.message.body.jms=true

value.converter=org.apache.kafka.connect.converters.StringConverter

* Messages are JMS BytesMessage, pass byte array as the Kafka message value

mq.message.body.jms=true

value.converter=org.apache.kafka.connect.converters.ByteArrayConverter

* Messages are JMS TextMessage, pass string data as the Kafka message value

mq.message.body.jms=true

value.converter=org.apache.kafka.connect.storage.StringConverter

# ***Kafka Connect sink connector for IBM MQ***

kafka-connect-mq-sink is a [Kafka Connect](http://kafka.apache.org/documentation.html#connect) sink connector for copying data from Apache Kafka into IBM MQ.

The connector is supplied as source code which we can easily build into a JAR file.

## Building the connector

To build the connector, we must have the following installed:

* [git](https://git-scm.com/)
* [Maven 3.0 or later](https://maven.apache.org)
* Java 8 or later

Clone the repository with the following command:

git clone https://github.com/ibm-messaging/kafka-connect-mq-sink.git

Change directory into the kafka-connect-mq-sink directory:

cd kafka-connect-mq-sink

Build the connector using Maven:

mvn clean package

Once built, the output is a single JAR target/kafka-connect-mq-sink-<version>-jar-with-dependencies.jar which contains all of the required dependencies.

## Running the connector

For step-by-step instructions, see the following guides for running the connector:

* connecting to Apache Kafka [running locally](https://github.com/ibm-messaging/kafka-connect-mq-sink/blob/master/UsingMQwithKafkaConnect.md)
* connecting to an installation of [IBM Event Streams](https://ibm.github.io/event-streams/connecting/mq/sink)

To run the connector, we must have:

* The JAR from building the connector
* A properties file containing the configuration for the connector
* Apache Kafka 2.0.0 or later, either standalone or included as part of an offering such as IBM Event Streams
* IBM MQ v8 or later, or the IBM MQ on Cloud service

The connector can be run in a Kafka Connect worker in either standalone (single process) or distributed mode. It's a good idea to start in standalone mode.

### Running in standalone mode

We need two configuration files, one for the configuration that applies to all of the connectors such as the Kafka bootstrap servers, and another for the configuration specific to the MQ sink connector such as the connection information for our queue manager. For the former, the Kafka distribution includes a file called connect-standalone.properties that we can use as a starting point. For the latter, we can use config/mq-sink.properties in this repository.

The connector connects to MQ using either a client or a bindings connection. For a client connection, we must provide the name of the queue manager, the connection name (one or more host/port pairs) and the channel name. In addition, we can provide a user name and password if the queue manager is configured to require them for client connections. If we look at the supplied config/mq-sink.properties, you'll see how to specify the configuration required. For a bindings connection, we must provide provide the name of the queue manager and also run the Kafka Connect worker on the same system as the queue manager.

To run the connector in standalone mode from the directory into which we installed Apache Kafka, we use a command like this:

bin/connect-standalone.sh connect-standalone.properties mq-sink.properties

### Running in distributed mode

We need an instance of Kafka Connect running in distributed mode. The Kafka distribution includes a file called connect-distributed.properties that we can use as a starting point, or follow [Running with Docker](https://github.com/ibm-messaging/kafka-connect-mq-sink#running-with-docker) or [Deploying to Kubernetes](https://github.com/ibm-messaging/kafka-connect-mq-sink#deploying-to-kubernetes).

To start the MQ connector, we can use config/mq-sink.json in this repository after replacing all placeholders and use a command like this:

curl -X POST -H "Content-Type: application/json" http://localhost:8083/connectors \

--data "@./config/mq-sink.json"

## Running with Docker

This repository includes an example Dockerfile to run Kafka Connect in distributed mode. It also adds in the MQ sink connector as an available connector plugin. It uses the default connect-distributed.properties and connect-log4j.properties files.

1. mvn clean package
2. docker build -t kafkaconnect-with-mq-sink:1.3.0 .
3. docker run -p 8083:8083 kafkaconnect-with-mq-sink:1.3.0

**NOTE:** To provide custom properties files create a folder called config containing the connect-distributed.properties and connect-log4j.properties files and use a Docker volume to make them available when running the container like this:

docker run -v $(pwd)/config:/opt/kafka/config -p 8083:8083 kafkaconnect-with-mq-sink:1.3.0

To start the MQ connector, we can use config/mq-sink.json in this repository after replacing all placeholders and use a command like this:

curl -X POST -H "Content-Type: application/json" http://localhost:8083/connectors \

--data "@./config/mq-sink.json"

## Deploying to Kubernetes

This repository includes a Kubernetes yaml file called kafka-connect.yaml. This will create a deployment to run Kafka Connect in distributed mode and a service to access the deployment.

The deployment assumes the existence of a Secret called connect-distributed-config and a ConfigMap called connect-log4j-config. These can be created using the default files in our Kafka install, however it is easier to edit them later if comments and whitespaces are trimmed before creation.

### Creating Kafka Connect configuration Secret and ConfigMap

Create Secret for Kafka Connect configuration:

1. cp kafka/config/connect-distributed.properties connect-distributed.properties.orig
2. sed '/^#/d;/^[[:space:]]\*$/d' < connect-distributed.properties.orig > connect-distributed.properties
3. kubectl -n <namespace> create secret generic connect-distributed-config --from-file=connect-distributed.properties

Create ConfigMap for Kafka Connect Log4j configuration:

1. cp kafka/config/connect-log4j.properties connect-log4j.properties.orig
2. sed '/^#/d;/^[[:space:]]\*$/d' < connect-log4j.properties.orig > connect-log4j.properties
3. kubectl -n <namespace> create configmap connect-log4j-config --from-file=connect-log4j.properties

### Creating Kafka Connect deployment and service in Kubernetes

**NOTE:** We will need to [build the Docker image](https://github.com/ibm-messaging/kafka-connect-mq-sink#running-with-docker) and push it to our Kubernetes image repository. Remember that the supplied Dockerfile is just an example and we will have to modify it for our needs. We might need to update the image name in the kafka-connect.yaml file.

1. Update the namespace in kafka-connect.yaml
2. kubectl -n <namespace> apply -f kafka-connect.yaml
3. curl <serviceIP>:<servicePort>/connector-plugins to see whether the MQ sink connector is available to use

### Deploying to OpenShift using Strimzi

This repository includes a Kubernetes yaml file called strimzi.kafkaconnector.yaml for use with the [Strimzi](https://strimzi.io) operator. Strimzi provides a simplified way of running the Kafka Connect distributed worker, by defining either a KafkaConnect resource or a KafkaConnectS2I resource.

The KafkaConnectS2I resource provides a nice way to have OpenShift do all the work of building the Docker images for you. This works particularly nicely combined with the KafkaConnector resource that represents an individual connector.

The following instructions assume we are running on OpenShift and have Strimzi 0.16 or later installed.

#### Start a Kafka Connect cluster using KafkaConnectS2I

1. Create a file called kafka-connect-s2i.yaml containing the definition of a KafkaConnectS2I resource. We can use the examples in the Strimzi project to get started.
2. Configure it with the information it needs to connect to our Kafka cluster. We must include the annotation strimzi.io/use-connector-resources: "true" to configure it to use KafkaConnector resources so we can avoid needing to call the Kafka Connect REST API directly.
3. oc apply -f kafka-connect-s2i.yaml to create the cluster, which usually takes several minutes.

#### Add the MQ sink connector to the cluster

1. mvn clean package to build the connector JAR.
2. mkdir my-plugins
3. cp target/kafka-connect-mq-sink-\*-jar-with-dependencies.jar my-plugins
4. oc start-build <kafkaconnectClusterName>-connect --from-dir ./my-plugins to add the MQ sink connector to the Kafka Connect distributed worker cluster. Wait for the build to complete, which usually takes a few minutes.
5. oc describe kafkaconnects2i <kafkaConnectClusterName> to check that the MQ sink connector is in the list of available connector plugins.

#### Start an instance of the MQ sink connector using KafkaConnector

1. cp deploy/strimzi.kafkaconnector.yaml kafkaconnector.yaml
2. Update the kafkaconnector.yaml file to replace all of the values in <>, adding any additional configuration properties.
3. oc apply -f kafkaconnector.yaml to start the connector.
4. oc get kafkaconnector to list the connectors. We can use oc describe to get more details on the connector, such as its status.

## Data formats

Kafka Connect is very flexible but it's important to understand the way that it processes messages to end up with a reliable system. When the connector encounters a message that it cannot process, it stops rather than throwing the message away. Therefore, we need to make sure that the configuration we use can handle the messages the connector will process.

Each message in Kafka Connect is associated with a representation of the message format known as a *schema*. Each Kafka message actually has two parts, key and value, and each part has its own schema. The MQ sink connector does not currently use message keys, but some of the configuration options use the word *Value* because they refer to the Kafka message value.

When the MQ sink connector reads a message from Kafka, it is processed using a *converter* which chooses a schema to represent the message format and creates a Java object containing the message value. The MQ sink connector then converts this internal format into the message it sends to MQ using a *message builder*.

There are three converters built into Apache Kafka. The following table shows which converters to use based on the incoming message encoding.

| Incoming Kafka message | Converter class |
| --- | --- |
| Any | org.apache.kafka.connect.converters.ByteArrayConverter |
| String | org.apache.kafka.connect.storage.StringConverter |
| JSON, may have schema | org.apache.kafka.connect.json.JsonConverter |

There are three message builders supplied with the connector, although we can write our own. The basic rule is that if you're using a converter that uses a very simple schema, the default message builder is probably the best choice. If you're using a converter that uses richer schemas to represent complex messages, the JSON message builder is good for generating a JSON representation of the complex data. The following table shows some likely combinations.

| Converter class | Message builder class | Outgoing MQ message |
| --- | --- | --- |
| org.apache.kafka.connect.converters.ByteArrayConverter | com.ibm.eventstreams.connect.mqsink.builders.DefaultMessageBuilder | **Binary data** |
| org.apache.kafka.connect.storage.StringConverter | com.ibm.eventstreams.connect.mqsink.builders.DefaultMessageBuilder | **String data** |
| org.apache.kafka.connect.json.JsonConverter | com.ibm.eventstreams.connect.mqsink.builders.JsonMessageBuilder | **JSON, no schema** |

When we set *mq.message.body.jms=true*, the MQ messages are generated as JMS messages. This is appropriate if the applications receiving the messages are themselves using JMS.

There's no single configuration that will always be right, but here are some high-level suggestions.

* Message values are treated as byte arrays, pass byte array into MQ message

value.converter=org.apache.kafka.connect.converters.ByteArrayConverter

* Message values are treated as strings, pass string into MQ message

value.converter=org.apache.kafka.connect.storage.StringConverter