



Confluent Developer Training for Apache Kafka Exercise Manual

B7/801/A

Table of Contents

- Introduction 1
- Hands-On Exercise: Using Kafka’s Command-Line Tools 5
- Hands-On Exercise: Consuming from Multiple Partitions 10
- Hands-On Exercise: Writing a Basic Producer 12
- Hands-On Exercise: Writing a Basic Consumer 18
- Hands-On Exercise: Accessing Previous Data 20
- Hands-On Exercise: Using Kafka with Avro 22
- Hands-On Exercise: Running Kafka Connect 27
- Hands-On Exercise: Writing a Kafka Streams Application 34
- Hands-On Exercise: Data Transformations with KSQL 35
- Appendix: Adding Connectors: Alternate Method 40

Introduction

This document provides Hands-On Exercises for the course *Confluent Developer Training for Apache Kafka*. You will use a setup that includes one virtual machine (VM) running ZooKeeper, one Broker, Schema Registry, REST proxy, and Connect.

You will use Confluent Control Center to configure the Kafka connectors. To achieve this, the VM can also run the Control Center service which is backed by the same Kafka cluster.

This VM is pre-configured with Confluent Platform version 4.0.0 which includes Kafka 1.0.0.

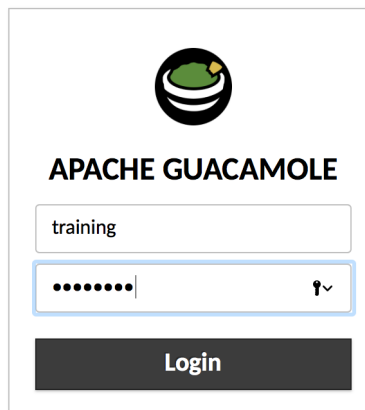
Login: Local

If the VM is installed on your local machine (e.g. with VirtualBox), then when you start the VM you are logged into the VM automatically as user `training`, and it has passwordless `sudo` enabled.

Login: Remote

If you are accessing the VM remotely, then the VM is running in the cloud and you will need a web browser on your local machine to access the them through a remote desktop utility called Guacamole. See the Copy and Paste section below for more details on browser compatibility with Guacamole.

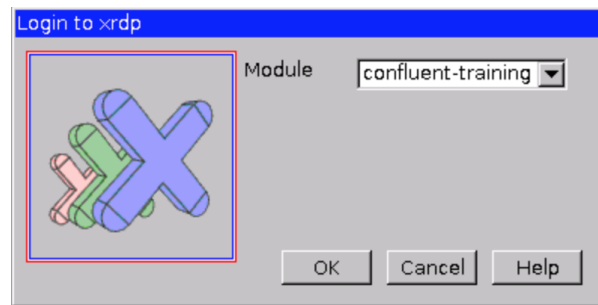
1. Through a web browser on your local machine, use the IP address provided by your instructor to navigate to the VM on its public network: `http://[IP address]/training`. This will take you to the “Apache Guacamole” login screen.

The image shows the Apache Guacamole login interface. At the top is the Guacamole logo, a stylized mole head. Below it, the text "APACHE GUACAMOLE" is displayed. There are two input fields: the first contains the username "training", and the second contains a masked password "....." with a toggle icon to the right. Below the password field is a dark grey "Login" button.

2. Use the following credentials to log in

Username: **training**
Password: **training**

3. In the “Login to xrdp” screen, select the default Module “confluent-training”, and click OK



4. You will now be automatically logged into the VM as user `training`. This user `training` has passwordless `sudo` enabled.



If you encounter an error and cannot log in, it is probably transient so please try again to log in.

Copy and Paste

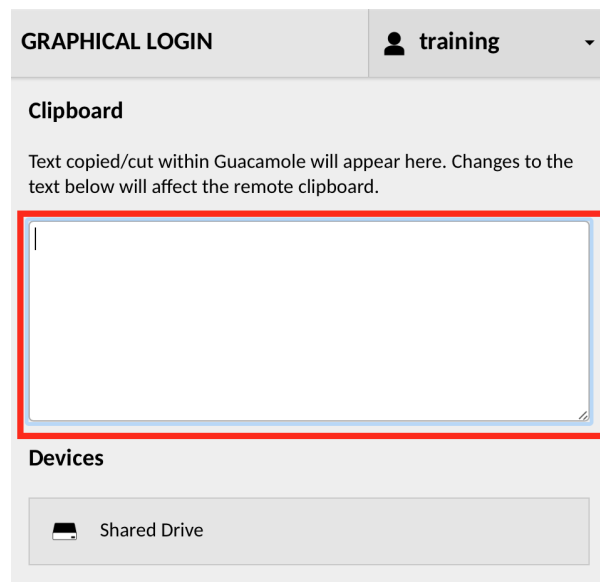
We encourage you to type out the commands in this exercise manual to become more familiar with them. However, you may copy and paste them as needed. If you are using this exercise manual from within the VM, copy and paste will be easier. Click the circular Firefox icon at the top of the VM screen next to the System menu, navigate to <http://confluentuniversity.com>, sign in, and download this exercise manual.

If you are using this exercise manual from your local machine, you can copy and paste commands between the local machine and the VM. In this case, review the following to simplify copying and pasting:

1. If you are viewing this PDF using Apple's Preview application on a Mac, you may find that copying and pasting commands into the VM does not work properly with the backslash `\`. This is a known issue with Preview; either copy each line separately, or use an alternative viewer such as Adobe Acrobat Reader.

Copy and Paste: Remote

2. Since the VM is accessed via Guacamole, Guacamole has a special clipboard area to copy and paste between your local machine and the VM. From within the browser click `Ctrl-Alt-Shift` at the same time. This brings up the Guacamole clipboard on the left side of the screen. Paste text into this clipboard and then click `Ctrl-Alt-Shift` to close the clipboard. Then you will be able to paste it on the remote VM.

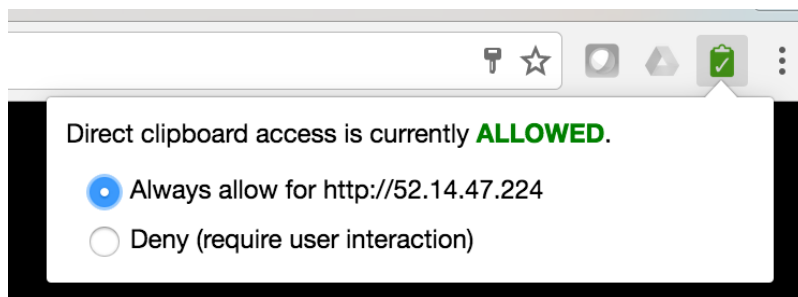


3. You should be using Google Chrome as your web browser. Consider adding a third-party extension called Clipboard Permission Manager that will enable copy and paste to work without the Guacamole clipboard:

- a. Install the Clipboard Permission Manager from <https://chrome.google.com/webstore/detail/clipboard-permission-manager/ipbhneeanpgkaleihlknhjiaamobkceh>
- b. Navigate to the webpage with the IP address provided by your instructor `http://[IP address]/training`
- c. In the top right corner, the Clipboard Permission Manager icon may show red



- d. Click on the Clipboard Permission Manager icon and select "Always allow"



- e. The Clipboard Permission Manager icon should now show green

Command Line Examples

Most Exercises contain commands that must be run from the command line. These commands will look like this:

```
$ pwd  
/home/training  
  
$ cat /etc/hosts  
127.0.0.1 localhost confluent-training-vm broker101 zookeeper1 kafkarest1 schemaregistry1  
connect1 controlcenter
```

Commands you should type are shown in bold; non-bold text is an example of the output produced as a result of the command.

Machine Aliases

The VM is a single node. To make it easier to understand what you are connecting to when writing code, we have created various aliases in the `/etc/hosts` file. The hostnames `broker101`, `zookeeper1`, etc. simply resolve back to `127.0.0.1`.

Project Directories

For the Hands-On Exercises that involve code, you will find information at the beginning of the exercise telling you where the project directory is. The location is relative to `/home/training/developer/exercise_code`.

Programming Exercises

For the programming exercises, we have provided three directories:

1. `solution` contains a working sample solution to the problem
2. `partial` contains a partial solution, with some lines left for you to complete – marked with `TODO` comments
3. `stubs` are essentially empty

If you are feeling confident about the task, start from the `stubs` files; for more help, use `partial`; and feel free to look in `solution` for hints as you go.

Hands-On Exercise: Using Kafka's Command-Line Tools

In this Hands-On Exercise you will start to become familiar with Kafka's command-line tools. You will:

- Verify Kafka is running
- Use a console program to produce a message
- Use a console program to consume a message

Verifying that Kafka is Running Correctly

There are various ways to verify that all of the Kafka system's daemons are running. It is important to verify that Kafka is functioning correctly before you start to use it. You may need to restart a daemon process during the class, if it has terminated for some reason.

1. Open the Terminal Emulator on the desktop.
2. Get the listing of Java processes:

```
$ sudo jps
2183 Jps
1752 SupportedKafka
1832 SchemaRegistryMain
1775 KafkaRestMain
1487 QuorumPeerMain
```

This will display a list of all Java processes running on the VM. (If the VM is running in the cloud, a process called `Bootstrap` used for Guacamole will be listed as well). The four Kafka-related processes you should see, and their Linux service names, are:

Java Process Name	Service Name
QuorumPeerMain	zookeeper
SupportedKafka	kafka-server
KafkaRestMain	kafka-rest
SchemaRegistryMain	schema-registry

3. If any of the four Java processes are not present, start the relevant service(s) by typing:

```
$ sudo service servicename start
```



Order matters when starting services. If multiple services are not running, they should be started in the order listed in the table above.

Console Producing and Consuming

Kafka has built-in command line utilities to produce messages to a Topic, and read messages from a Topic. These are extremely useful to verify that Kafka is working correctly, and for testing and debugging.

4. Run the command:

```
$ kafka-console-producer
```

This will bring up a list of parameters that the `kafka-console-producer` program can receive. Take a moment to look through the options. We will discuss many of their meanings later in the course.

5. Run `kafka-console-producer` again with the required arguments:

```
$ kafka-console-producer --broker-list broker101:9092 --topic testing
```

6. Type:

```
> some data
```

And hit 'Enter'. This will produce a message with the value `some data` to the `testing` Topic. You will see the following output the first time you enter a line of data:

```
WARN Error while fetching metadata with correlation id 0 : {testing=LEADER_NOT_AVAILABLE}
(org.apache.kafka.clients.NetworkClient)
```

This is because the Topic `testing` does not initially exist. When the first line of text is produced to it from `kafka-console-producer`, it will automatically be created.

7. Type:

```
> more data
```

And hit 'Enter'.

8. Type:

```
> final data
```

And hit 'Enter'.

9. Press `Ctrl-d` to exit the `kafka-console-producer` program.

10. Now we will use a Consumer to retrieve the data that was produced. Run the command:

```
$ kafka-console-consumer
```

This will bring up a list of parameters that the `kafka-console-consumer` can receive. Take a moment to look through the options.

11. Run `kafka-console-consumer` again with the following arguments:

```
$ kafka-console-consumer \
  --bootstrap-server broker101:9092 \
  --from-beginning \
  --topic testing
```

You should see all the messages that you produced using `kafka-console-producer` earlier.

12. Press `Ctrl-c` to exit `kafka-console-consumer`.

If You Have More Time

13. The `kafka-console-producer` and `kafka-console-consumer` programs can be run at the same time. Run `kafka-console-producer` and `kafka-console-consumer` in separate terminal windows at the same time to see how `kafka-console-consumer` receives the events.

14. By default, `kafka-console-producer` and `kafka-console-consumer` assume null keys. They can also be run with appropriate arguments to write and read keys as well as values. Re-run the Producer with additional arguments to write (key,value) pairs to the Topic:

```
$ kafka-console-producer \
  --broker-list broker101:9092 \
  --topic testing \
  --property parse.key=true \
  --property key.separator=,
```

When you enter data, separate the key and the value with a comma. For example:

```
> some key 1, some value 1
> some key 2, some value 2
> some key 1, some value 3
```

Then re-run the Consumer with additional arguments to print the key as well as the value:

```
$ kafka-console-consumer \
--bootstrap-server broker101:9092 \
--from-beginning \
--topic testing \
--property print.key=true
```

Press `Ctrl-c` to exit `kafka-console-consumer`.

15. Kafka's data in ZooKeeper can be accessed using the `zookeeper-shell` command:

```
$ zookeeper-shell zookeeper1
Connecting to zookeeper1
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
```

a. From within the `zookeeper-shell` application, type `ls /` to view the directory structure in ZooKeeper. Note the `/` is required.

```
ls /
[schema_registry, cluster, controller_epoch, controller, brokers, zookeeper, admin,
isr_change_notification, consumers, config]
ls /brokers
[ids, topics, seqid]
ls /brokers/ids
[101]
```

b. Press `Ctrl-c` to exit the ZooKeeper shell.



STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Consuming from Multiple Partitions

In this Hands-On Exercise, you will create a Topic with multiple Partitions, Produce data to those Partitions, and then read it back to observe issues with ordering.

1. Create a Topic manually with Kafka's command-line tool, specifying that it should have two Partitions:

```
$ kafka-topics \  
--zookeeper zookeeper1:2181 \  
--create \  
--topic two-p-topic \  
--partitions 2 \  
--replication-factor 1  
Created topic "two-p-topic"
```

2. Use the command-line Producer to write several lines of data to the Topic.

```
$ seq 1 100 > numlist  
$ kafka-console-producer \  
--broker-list broker101:9092 \  
--topic two-p-topic < numlist
```

3. Use the command-line Consumer to read the Topic

```
$ kafka-console-consumer \  
--bootstrap-server broker101:9092 \  
--from-beginning \  
--topic two-p-topic
```

4. Note the order of the numbers. Rerun the Producer command in step 2, then rerun the Consumer command in step 3 and see if you observe any difference in the output.
5. What do you think is happening as the data is being written?
6. Try creating a new Topic with three Partitions and running the same steps again.



STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Writing a Basic Producer

Project directory: `helloworld` or `helloworld_python`

In this Hands-On Exercise you will write a Producer in either Java or Python.

Choosing a Language

In this Exercise, you will write programs using the Kafka API. The class has full solutions for Java, and for Python using the REST interface. You will need to choose which language to write your code in.

Once you have chosen your language, please follow the instructions for that language in the Exercise Manual. A heading with (Java) or (Python) means that section is for a specific language. A heading that doesn't contain (Java) or (Python) means that the section is for both languages.



If you choose Python, you can use a text editor such as `vi` or use a Python editor named Geany that is included in the VM. You can use Geany to make it easier to write Python programs. Launch it by going to the Applications menu at the top left of the screen, and choose the Programming sub-menu; or type `geany` at the bash prompt.

Using Eclipse (Java)

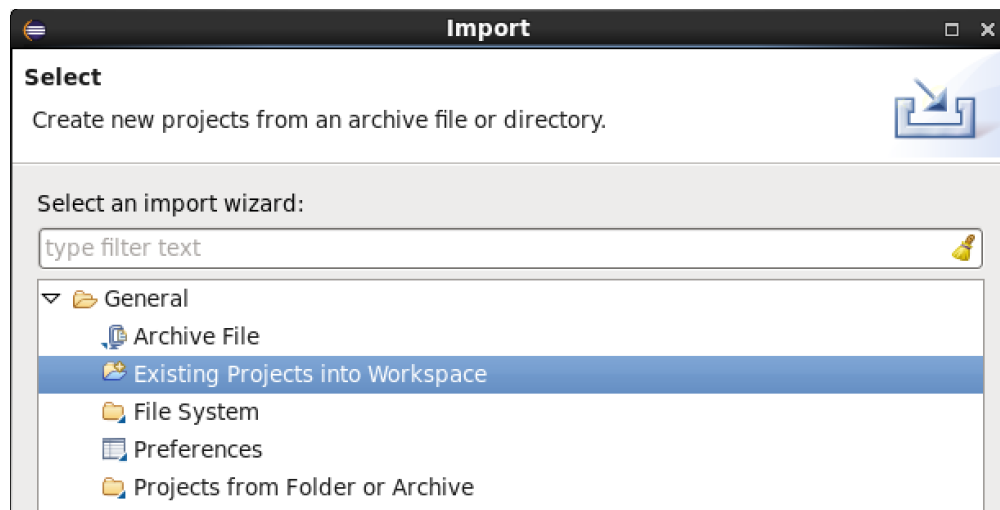
1. Go to the project directory specified at the beginning of the Exercise description, e.g. `helloworld`.

```
$ cd /home/training/developer/exercise_code/helloworld
```

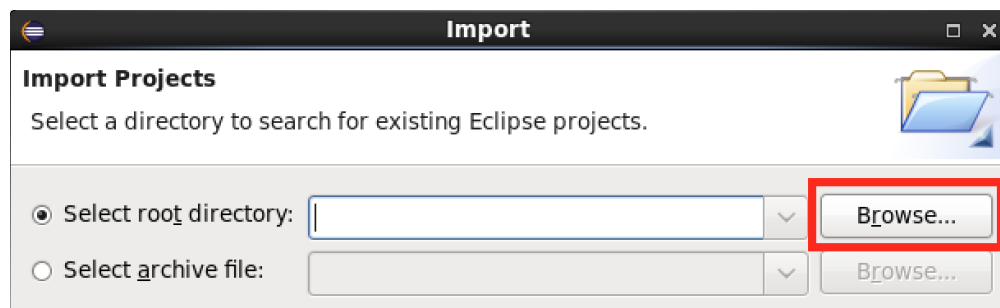
2. Create the Eclipse project with Maven. In our training class, all the project dependencies have already been downloaded, and Maven is run in offline mode.

```
$ mvn eclipse:eclipse
...
[INFO] BUILD SUCCESS
...
```

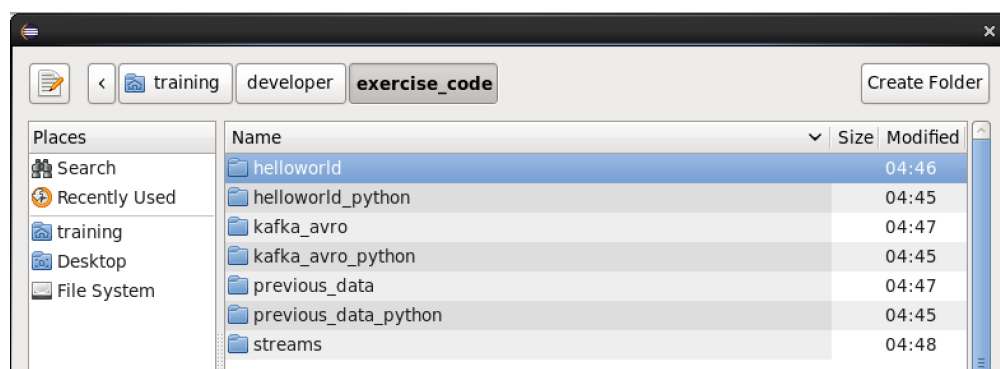
3. Start Eclipse by double-clicking the Eclipse icon on the desktop.
4. Import the `helloworld` project into Eclipse.
 - a. Go to **File→Import**.
 - b. Choose **General→Existing Projects into Workspace**.



- c. Click the **Next** button.
- d. To select the root directory for the project, click on Browse.



- e. Navigate to `/home/training/developer/exercise_code/helloworld`

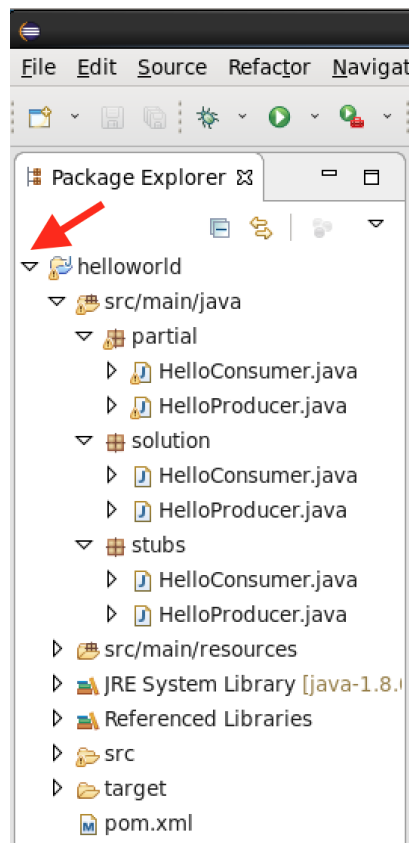


- f. Click **OK**. The Projects section of the import screen will update and show the `helloworld` project.



g. Click **Finish**.

5. Click the arrow on the left side of the `helloworld` project to view its contents.



You will need to follow a similar procedure for each of the subsequent programming Hands-On Exercises in order to create the project and import it into Eclipse.

Creating a Producer (Java)

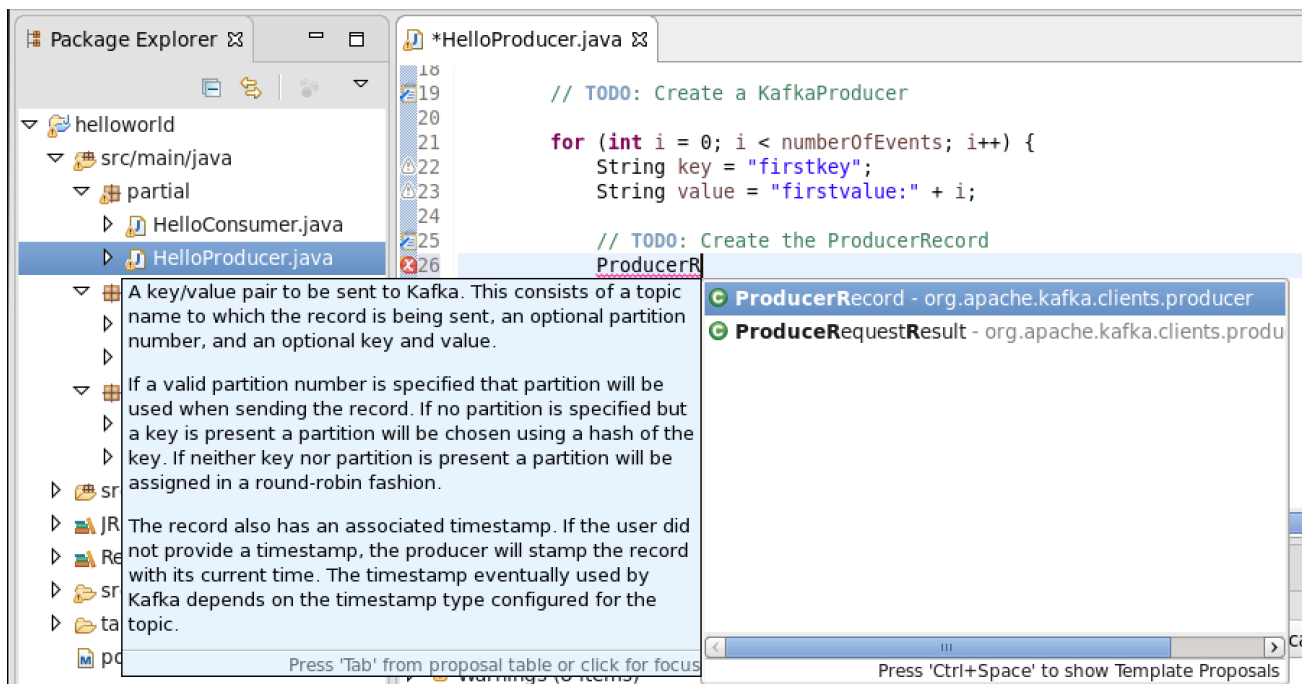
6. Double-click `HelloProducer.java` in either the `solution`, `partial`, or `stubs` folder depending on the degree of difficulty you would like. The source code will appear in the center of the screen.

7. Create a `KafkaProducer` with the following characteristics:

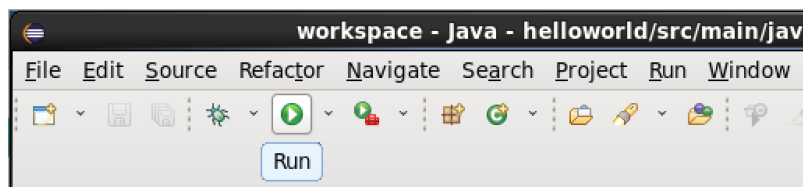
- Connects to `broker101:9092`
- Sends five messages to a Topic named `hello_world_topic`
- Sends all messages as type `String`
- Prints each key and value



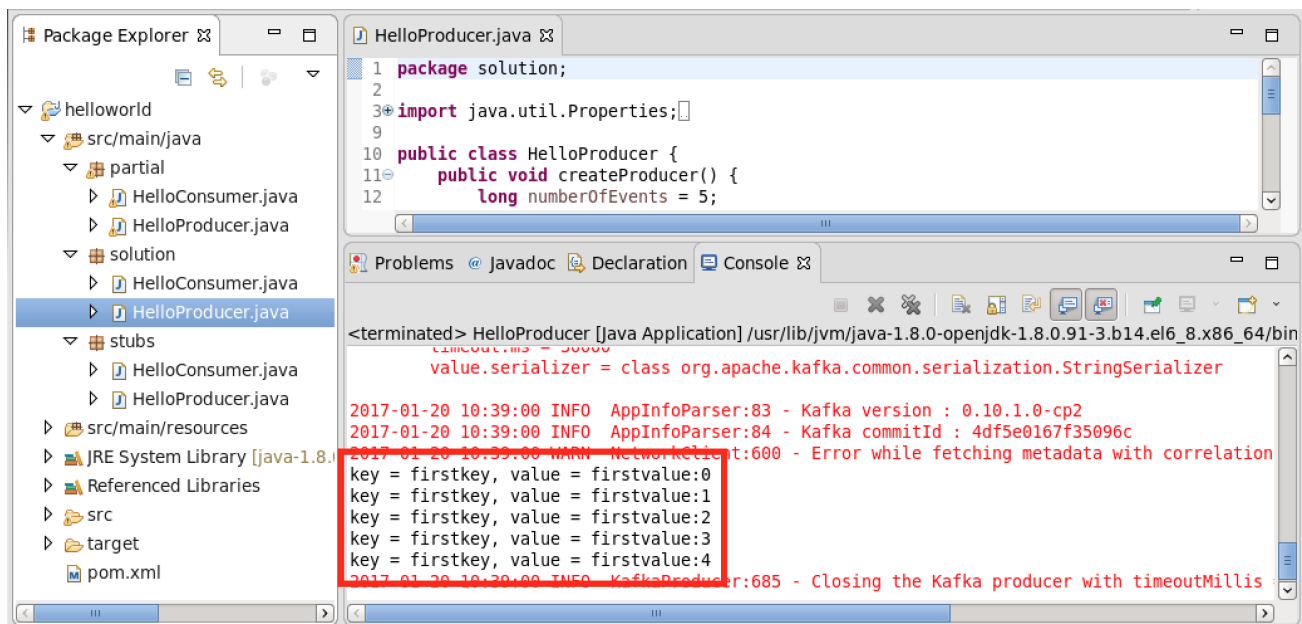
Eclipse tip: as you start to type a Java class name, click `ctrl-space` to see the possible autocomplete options and its documentation.



8. Run your `HelloProducer` code. Right-click on your java file and choose `Run As → Java Application`; or click the `Run` icon at the top of the screen to run the last executed Java program.



9. View the output of the run in the `Console` window at the bottom of the screen.



Creating a Producer (Python)

10. Go to the python project directory specified at the beginning of the Exercise description, e.g. `helloworld_python`.

```
$ cd /home/training/developer/exercise_code/helloworld_python
```

11. Create a Producer with the following characteristics:

- Sends five messages
- Connects to `kafkarest1:8082`
- Sends five messages to a Topic named `hello_world_topic`
- Sends all messages as strings
- Checks for HTTP error codes and outputs the error message if one was returned

Testing Your Code

12. To test your code, run the command-line Consumer and have it consume from `hello_world_topic`.

```
$ kafka-console-consumer \
--bootstrap-server broker101:9092 \
--from-beginning \
--topic hello_world_topic
```



Do not attempt to write the Consumer code yet. We will do that in the next Hands-On Exercise.

13. Append `--property print.key=true` to the command-line Consumer to print the key as well as the value.

```
$ kafka-console-consumer \  
--bootstrap-server broker101:9092 \  
--from-beginning \  
--topic hello_world_topic \  
--property print.key=true
```



STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Writing a Basic Consumer

Project directory: `helloworld` or `helloworld_python`

In this Hands-On Exercise, you will write a basic Consumer in either Java or Python. It will consume from the same Topic `hello_world_topic` that the Producer in the previous exercise wrote to.



By default, the Consumer will start reading from the latest offset saved for its Consumer Group, so you may need to run the Producer in the previous exercise again to write new messages to the Topic.

Creating a Consumer (Java)

1. Double-click `HelloConsumer.java` in either the `solution`, `partial`, or `stubs` folder depending on the degree of difficulty you would like. The source code will appear in the center of the screen.
2. Write a Java Consumer to read the data you wrote with the Producer you created in the previous Exercise. If you did not have time to finish that Exercise, run the sample solution provided in the previous exercise to write some messages to the Topic `hello_world_topic` first.
3. After running the Java Consumer in Eclipse, stop the process by pressing the red square in the Eclipse console window.

```

HelloConsumer [Java Application] /usr/lib/jvm/java-1
2017-12-12 11:46:32 INFO AppInfoParser:110 - CommitId : aad876836117a04f
2017-12-12 11:46:32 INFO AbstractCoordinator:341 - [Consumer clientId=consumer-1, groupId=1
2017-12-12 11:46:32 INFO ConsumerCoordinator:341 - [Consumer clientId=consumer-1, groupId=1
2017-12-12 11:46:32 INFO AbstractCoordinator:336 - [Consumer clientId=consumer-1, groupId=1
2017-12-12 11:46:32 INFO AbstractCoordinator:341 - [Consumer clientId=consumer-1, groupId=1
2017-12-12 11:46:32 INFO ConsumerCoordinator:341 - [Consumer clientId=consumer-1, groupId=1
offset = 0, key = firstkey, value = firstvalue:0
offset = 1, key = firstkey, value = firstvalue:1
offset = 2, key = firstkey, value = firstvalue:2
offset = 3, key = firstkey, value = firstvalue:3
offset = 4, key = firstkey, value = firstvalue:4
  
```

Creating a Consumer (Python)

4. Write a Consumer to read the data you wrote with the Producer you created in the previous Exercise. If you did not have time to finish that Exercise, run the sample solution provided in the previous exercise to write some messages to the Topic `hello_world_topic` first.
5. While the Python Consumer is running in one terminal window, in another terminal window rerun the Producer in the previous exercise again to write new messages to the Topic. This is because, by default, the Python Consumer using the REST proxy will start at the end of the log.
6. In the terminal window where you started the Consumer, hit `Ctrl-C` to terminate the process.



STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Accessing Previous Data

Project directory: `previous_data` or `previous_data_python`

In this Hands-On Exercise, you will create a Consumer which accesses data starting from the beginning of the Topic each time it launches.

Previous Data Consumer (Java)

1. Write a Consumer which reads all data from a Topic each time it starts. Test it using the command-line Producer, or by reading from one of the Topics you previously created.
2. After running the Java Consumer in Eclipse, stop the process by pressing the red square in the Eclipse console window.

Previous Data Consumer (Python)

3. Write a Consumer which reads all data from a Topic each time it starts. Test it using the command-line Producer, or by reading from one of the Topics you previously created.

If You Have More Time 1 (Java)

4. Write a Producer (or modify an already existing one) to write a large number of messages to a Topic. Make the messages sequential numbers (1, 2, 3...).
5. Write a Consumer which processes records and manually manages offsets by writing them to files on disk; it should write the offset after each message. (For simplicity, use a separate file for each Partition of the Topic.) To 'process' the message, just display it to the console. When the Consumer starts, it should seek to the correct offset. Modify your Consumer so that it halts randomly during processing so you can confirm that it performs correctly when restarted.
6. After running the Java Consumer in Eclipse, stop the process by pressing the red square in the Eclipse console window.

If You Have More Time 1 (Python)

7. Write a Consumer which asks you before committing its offsets. If you say no, then the next time the Consumer runs it should re-read the previous messages.

If You Have More Time 2 (Java)

8. Create a Topic with two Partitions. Write a Producer which uses a custom Partitioner; it should write random numbers between 1 and 20 to the Topic. Send numbers between 1 and 10 to one Partition, and numbers between 11 and 20 to the other.

If You Have More Time 2 (Python)

9. Write a Producer (or modify an already existing one) to write a large number of messages to a Topic. Make the messages sequential numbers (1, 2, 3...).
10. Write a Consumer which processes records from a Topic with a single partition 0 and manually manages offsets by writing them to files on disk; it should write the offset after each message. To 'process' the message, just display it to the console. When the Consumer starts, it should seek to the correct offset. Modify your Consumer so that it halts randomly during processing so you can confirm that it performs correctly when restarted.



STOP HERE. THIS IS THE END OF THE EXERCISE.

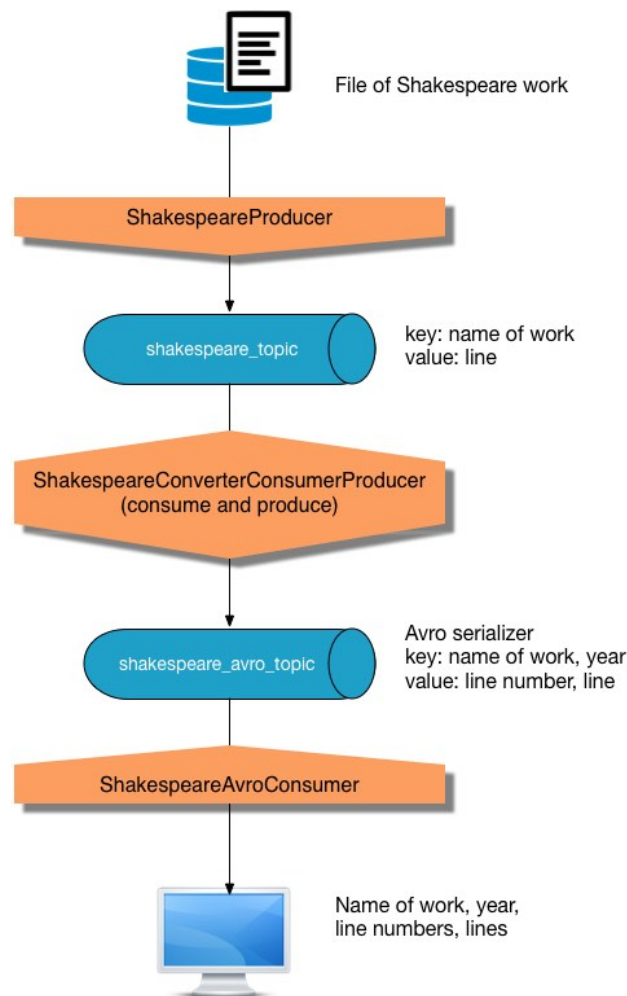
Hands-On Exercise: Using Kafka with Avro

Project directory: `kafka_avro` or `kafka_avro_python`

In this Hands-On Exercise, you will write and read Avro data to and from a Kafka cluster, and build a processing pipeline to turn text data from one Topic into Avro in another. To do this, you will:

- Write a Producer
- Write a program with a Consumer that reads data, converts it to Avro, then uses a Producer to write that Avro data to a new Topic
- Write a Consumer that reads Avro data from a Topic and displays it to the screen

The workflow is shown below:



The Dataset

This exercise will be using a subset of Shakespeare's plays. The files are located in `/home/training/developer/datasets/shakespeare`. Here is an example line from one of the files:

```
2360 Et tu, Brute?-- Then fall, Caesar!
```

Notice that the line of the play is preceded by the line number.

When Was That Play Written?

In the code you write, you will need to add the year that the play was written to the data. Here are the relevant years:

Hamlet: 1600
 Julius Caesar: 1599
 Macbeth: 1605
 Merchant of Venice: 1596
 Othello: 1604
 Romeo and Juliet: 1594

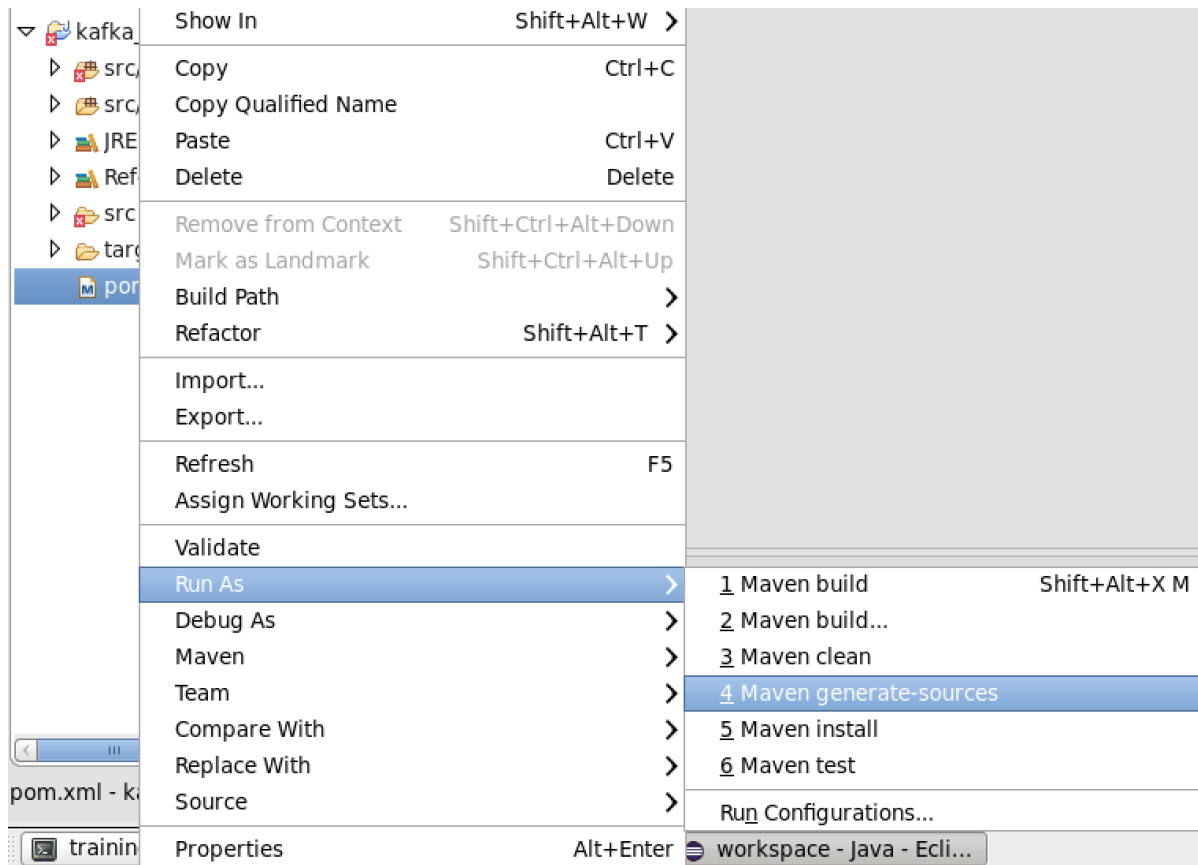
Creating the Avro Schemas

You need to create an Avro schema for the key and value of the message.

1. Make sure you are in the right directory. The `*.avsc` schema files should be in the relative path `kafka_avro/src/main/avro`
2. Create a schema for the key with following characteristics:
 - The name should be `ShakespeareKey`
 - A unique namespace, e.g. `partial.model`
 - The schema should have a field for the name of the work, e.g., `Julius Caesar`
 - The schema should have a field for the year the work was published, e.g., `1599`
3. Create a schema for the value with following characteristics:
 - The name should be `ShakespeareValue`
 - A unique namespace, e.g. `partial.model`
 - The schema should have a field for the line number
 - The schema should have a field for the line itself

Using the Schema files in Eclipse (Java only)

- Once you have created the schema files, generate the corresponding Java classes. From within Eclipse, right-click on the `pom.xml` and choose `Run As → Maven generate-sources`



- Verify that the corresponding java files were created for the `ShakespeareValue` and `ShakespeareKey` Avro schemas. They are located in the relative path `kafka_avro/src/main/java/`. From there the subdirectory corresponds to the namespace you provided in the schema file.
- If you have already imported the `kafka_avro` project into Eclipse, refresh it by right-clicking on the project in the left-hand pane and selecting 'Refresh'.

Kafka Consumers and Producers (Java)

In this Exercise, you are creating a processing pipeline. A Producer will read in every line of Shakespeare as a `String` and write it to a Topic. A Consumer will read the data from that first Topic, parse it, create Avro objects, and a Producer will write those Avro objects to a new Topic. The final piece of the pipeline is a Consumer that reads the Avro objects from the Topic and writes them to the screen with `System.out.println`.

- Create a Producer with the following characteristics:
 - Connects to `broker101:9092`
 - Reads in the files from the `shakespeare` directory line by line

- `/home/training/developer/datasets/shakespeare/`

- Sends each line as a separate message
 - Sends messages to a Topic named `shakespeare_topic`
 - Sends all messages as type `String`
 - The key should be the name of the play (which is contained in the filename)
 - The value should be the line from the play

8. Create a program containing a Consumer and Producer with the following characteristics:

- Consumes messages from `shakespeare_topic`
 - Always starts from the beginning of the Topic
- Reads all keys and values as type `String`
- Converts the key and value to Avro objects
- Writes messages to a Topic named `shakespeare_avro_topic`
 - The key should be a `ShakespeareKey` Avro object
 - The value should be a `ShakespeareValue` Avro object

9. Create a Consumer with the following characteristics:

- Consumes messages from `shakespeare_avro_topic`
 - Always starts from the beginning of the Topic
- Consumes all data as Avro objects
- Outputs the key and value of each message to the screen

Kafka Consumers and Producers (Python)

In this Exercise, you are creating a processing pipeline. A Producer will read in every line of Shakespeare as a `String` and write it to a Topic. A Consumer will read the data from that first Topic, parse it, create Avro objects, and a Producer will write those Avro objects to a new Topic. The final piece of the pipeline is a Consumer that reads the Avro objects from the Topic and writes them to the screen with `print`.

10. Create a Producer with the following characteristics:

- Connects to `kafkarest1:8082`
- Reads in the files from the `shakespeare` directory line by line
- Sends each line as a separate message
 - Sends messages to a Topic named `shakespeare_topic`

- Sends all messages as strings
- The key should be the name of the play (which is contained in the filename)
- The value should be the line from the play

11. Create a program containing a Consumer and Producer with the following characteristics:

- Consumes messages from `shakespeare_topic`
 - Always starts from the beginning of the topic
- Reads all keys and values as strings
- Converts the key and value to Avro objects
- Writes messages to a topic named `shakespeare_avro_topic`
 - The key should be a `ShakespeareKey` Avro object
 - The value should be a `ShakespeareValue` Avro object

12. Create a Consumer with the following characteristics:

- Consumes messages from `shakespeare_avro_topic`
 - Always starts from the beginning of the topic
- Consumes all data as Avro objects
- Outputs the key and value of each message to the screen



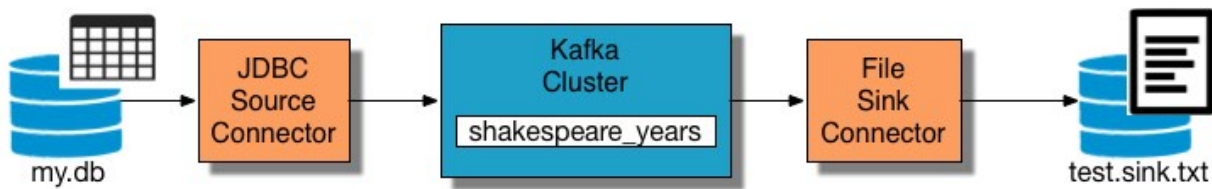
STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Running Kafka Connect

In this exercise, you will run Connect in distributed mode, and use the JDBC source Connector and File sink Connector.

Connect Pipeline

In this section, you will run Connect in distributed mode with two Connectors: a JDBC source Connector and a file sink Connector. The JDBC source Connector writes the contents of a database table to a Kafka Topic. The file sink Connector reads data from the same Kafka Topic and writes those messages to a file. It will update when new rows are added to the database.



1. Run the distributed Connect worker

```
$ sudo connect-distributed -daemon /etc/kafka/connect-distributed.properties
```

2. Create a new Topic called `shakespeare_years` with one Partition and one replica. Expect a warning about using periods and underscores.

```
$ kafka-topics \
--zookeeper zookeeper1:2181 \
--create \
--topic shakespeare_years \
--partitions 1 \
--replication-factor 1
Created topic "shakespeare_years".
```

3. Inspect the table `years` in the sqlite3 database `my.db`.

```
$ echo "SELECT * FROM years;" | sqlite3 /usr/local/lib/my.db
1|Hamlet|1600
2|Julius Caesar|1599
3|Macbeth|1605
4|Merchant of Venice|1596
5|Othello|1604
6|Romeo and Juliet|1594
7|Antony and Cleopatra|1606
```



The next few steps use the Confluent Control Center to configure the source and sink connectors. If you prefer to use `curl` commands to communicate with the Kafka Connect worker's REST endpoint instead, you may view the Appendix section called `Adding Connectors: Alternate Method`.

4. Start Control Center.

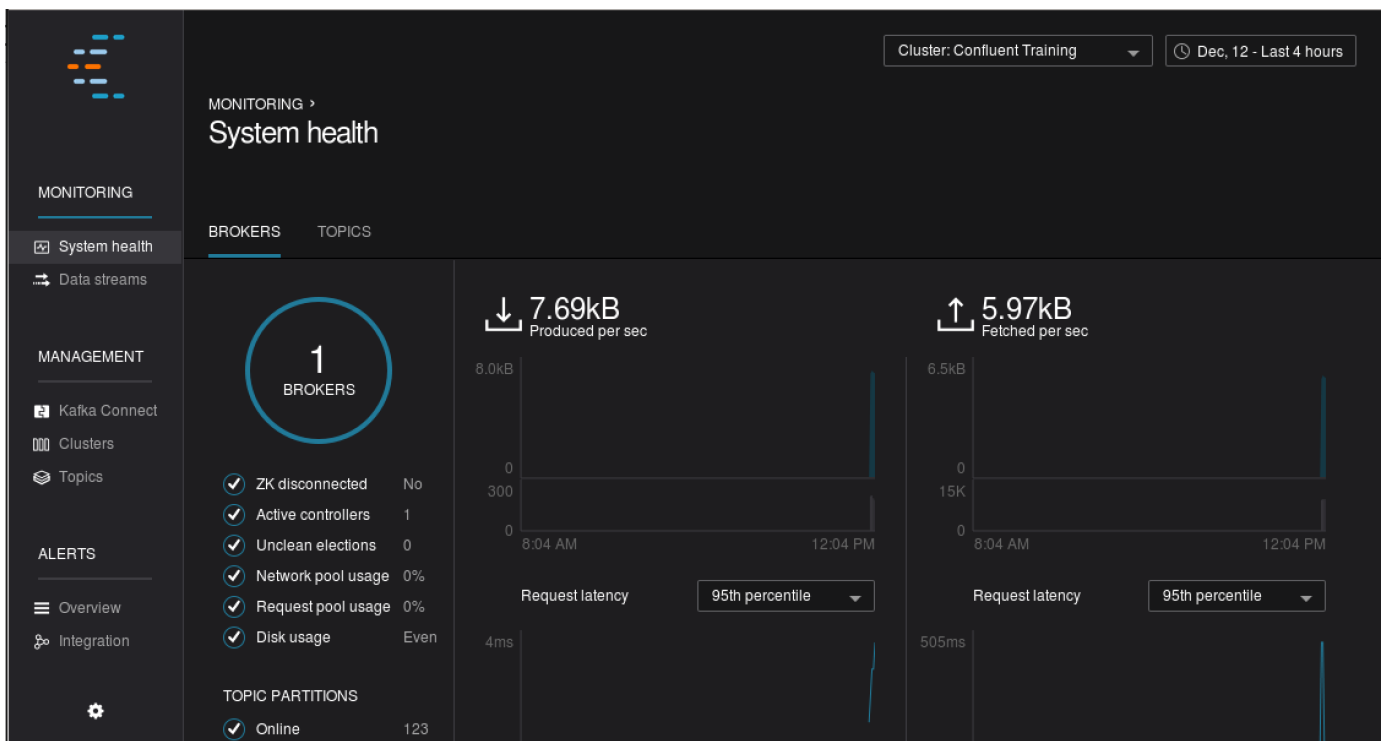
- a. Start the Control Center application and wait for a few minutes. Verify that it is running. Look for `ControlCenter`.

```
$ confluent_control_center_hard_reset.sh
$ sudo jps | grep -i ControlCenter
5210 ControlCenter
```

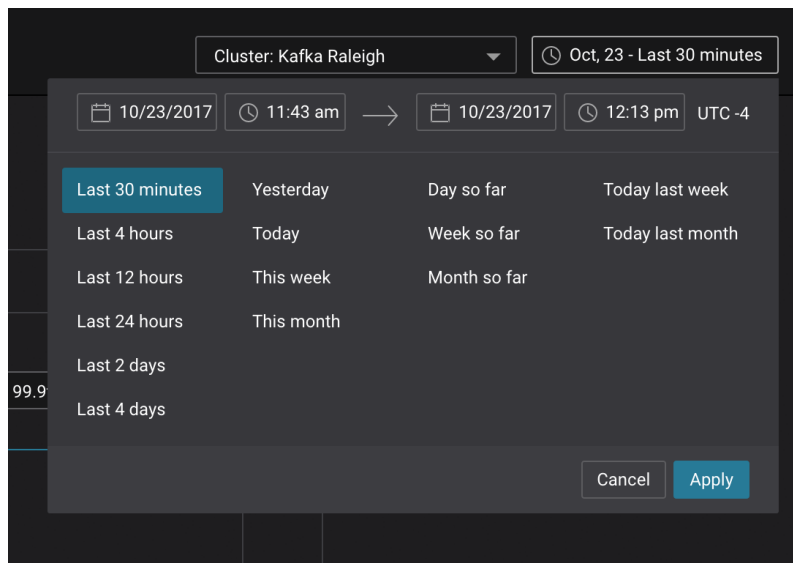
- b. On your host machine (not the VM), open a new browser in Google Chrome.
- c. Navigate to the Control Center GUI using the IP address provided by your instructor, adding port 9021 to the address: `http://[IP address]:9021`. If the VM is installed on your local machine (e.g. with VirtualBox), navigate to `http://localhost:9021`.



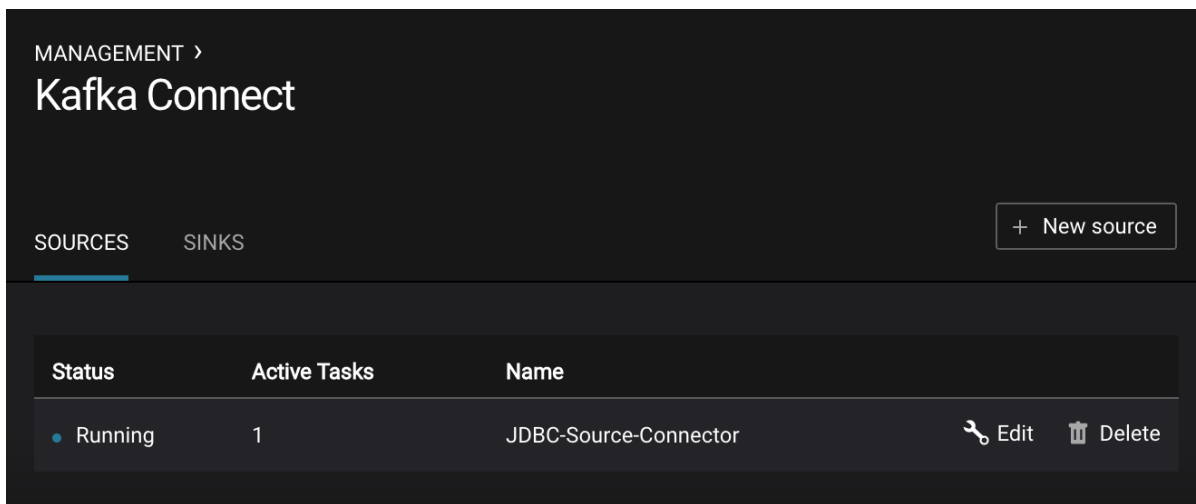
It may take a few minutes until the GUI is ready.



- d. In the Control Center GUI, click on the top right button that shows the current date, and change `Last 4 hours` to `Last 30 minutes`.



- e. In the Control Center GUI, in the `System health` landing page, observe the brokers in your cluster. Scroll down to see the table with one Broker: 101.
5. Add a new source connector to read data from the database `my.db` and write to the Kafka topic `shakespeare_years`. You can also run the command line tool for this connector, but in this class we will do it through Control Center.
 - a. In the Control Center GUI, click on `Kafka Connect`.
 - b. In the `SOURCES` tab, click `+ New Source`.
 - c. Configure the JDBC Source Connector
 - i. In the `Connector Class` dropdown, choose `JdbcSourceConnector`.
 - ii. In the `Name` text box, type `JDBC-Source-Connector`.
 - iii. Under `Database`, in the `JDBC URL` text box, type `jdbc:sqlite:/usr/local/lib/my.db`.
 - iv. Under `Mode`, in the `Table Loading Mode` text box, type `incrementing`.
 - v. Under `Mode`, in the `Incrementing Column Name` text box, type `id`.
 - vi. Under `Connector`, in the `Topic Prefix` text box, type `shakespeare_`.
 - d. Click `Continue`.
 - e. Click `Save & Finish`.
 - f. Verify you see the new connector `JDBC-Source-Connector` running.



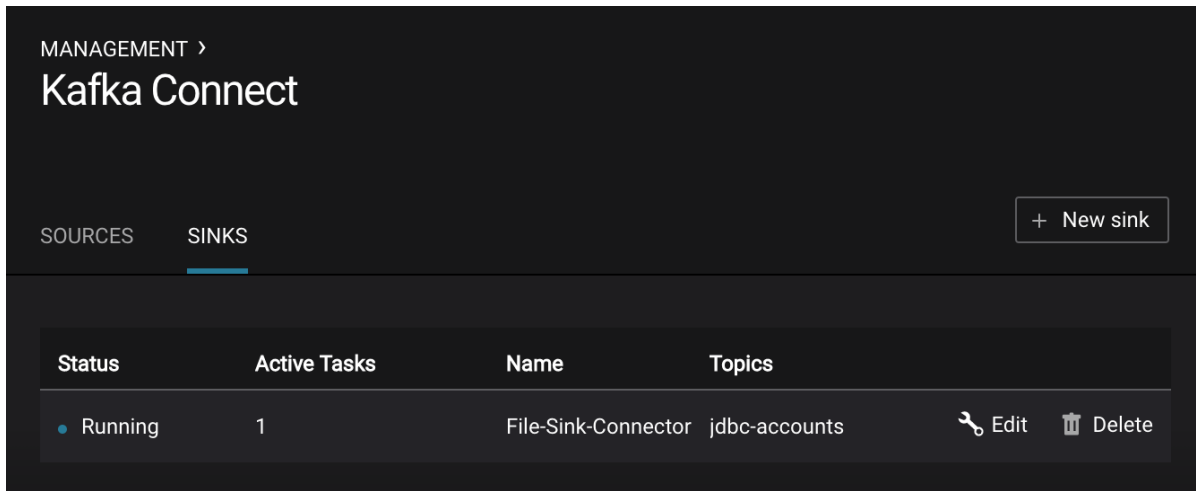
6. Launch another terminal and start the console Consumer for Topic `shakespeare_years`. Leave it running until instructed to terminate it. See what messages have been produced:

```
$ kafka-console-consumer \
--bootstrap-server broker101:9092 \
--from-beginning \
--topic shakespeare_years
{"schema":{"type":"struct","fields":[{"type":"int64","optional":false,"field":"id"}, {"type":"string","optional":true,"field":"name"}, {"type":"int64","optional":true,"field":"year"}], "optional":false, "name":"years"}, "payload":{"id":1, "name":"Hamlet", "year":1600}}
{"schema":{"type":"struct","fields":[{"type":"int64","optional":false,"field":"id"}, {"type":"string","optional":true,"field":"name"}, {"type":"int64","optional":true,"field":"year"}], "optional":false, "name":"years"}, "payload":{"id":2, "name":"Julius Caesar", "year":1599}}
...
```

Each message has a lot of JSON metadata. This is because the default key and value converters configured in the JDBC connector are set to use `JsonConverter`. In production, `AvroConverter` paired with the Schema Registry is highly recommended to best support upstream database schema changes.

7. Add a new sink connector to read data from the Kafka topic `shakespeare_years` and write to the file `/tmp/test.sink.txt`. You can also run the command line tool for this connector, but in this class we will do it through Control Center.
 - a. In the Control Center GUI, click on `Kafka Connect`.
 - b. In the `SINKS` tab, click `+ New Sink`.
 - c. In the `Topics` dropdown, choose `shakespeare_years`.
 - d. Click `Continue`.
 - e. Configure the `File Sink Connector`
 - i. In the `Connector Class` dropdown, choose `FileStreamSinkConnector`.
 - ii. In the `Name` text box, type `File-Sink-Connector`.
 - iii. Under `General`, in the `file` text box, type `/tmp/test.sink.txt`.

- f. Click Continue.
- g. Click Save & Finish.
- h. Verify you see the new connector `File-Sink-Connector` running.



8. Notice that a new file has been created in the current working directory, called `/tmp/test.sink.txt`. View this file to see that the sink connector did what it was expected to do:

```
$ cat /tmp/test.sink.txt
Struct{id=1,name=Hamlet,year=1600}
Struct{id=2,name=Julius Caesar,year=1599}
Struct{id=3,name=Macbeth,year=1605}
Struct{id=4,name=Merchant of Venice,year=1596}
Struct{id=5,name=Othello,year=1604}
Struct{id=6,name=Romeo and Juliet,year=1594}
Struct{id=7,name=Antony and Cleopatra,year=1606}
```

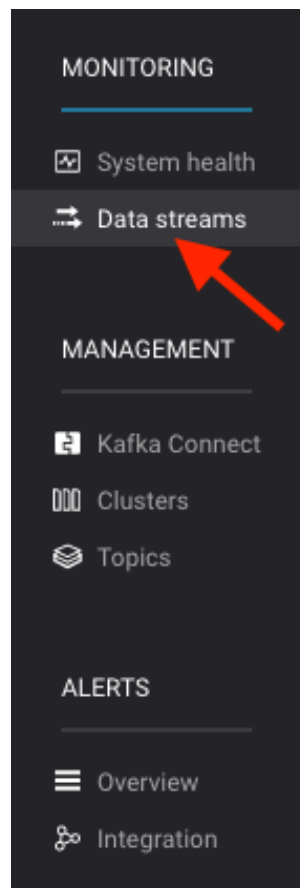
9. Insert a few new rows into the table `years`:

```
$ echo "INSERT INTO years(name,year) VALUES('Tempest',1611);" | \
sudo sqlite3 /usr/local/lib/my.db
$ echo "INSERT INTO years(name,year) VALUES('King Lear',1605);" | \
sudo sqlite3 /usr/local/lib/my.db
```

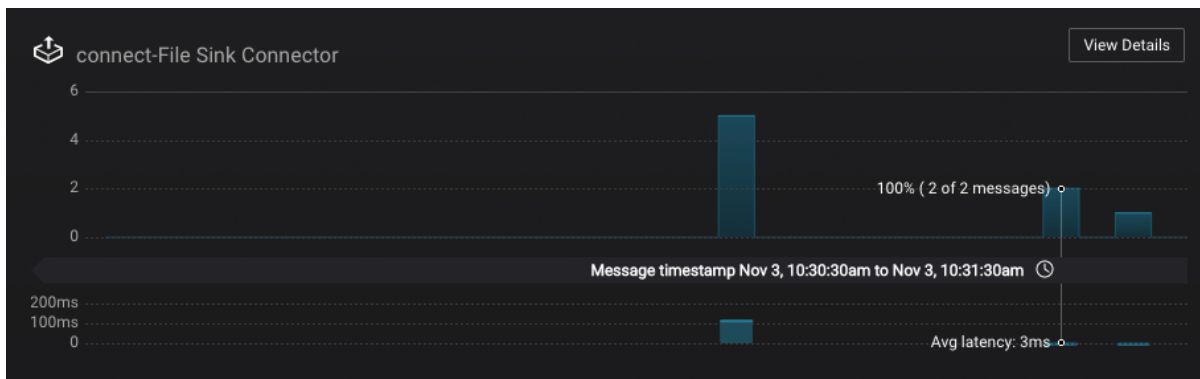
10. View the sink file, `/tmp/test.sink.txt` again. Notice that the new log lines are added to the sink file.

```
$ cat /tmp/test.sink.txt
Struct{id=1,name=Hamlet,year=1600}
Struct{id=2,name=Julius Caesar,year=1599}
Struct{id=3,name=Macbeth,year=1605}
Struct{id=4,name=Merchant of Venice,year=1596}
Struct{id=5,name=Othello,year=1604}
Struct{id=6,name=Romeo and Juliet,year=1594}
Struct{id=7,name=Antony and Cleopatra,year=1606}
Struct{id=8,name=Tempest,year=1611}
Struct{id=9,name=King Lear,year=1605}
```

11. In the Control Center GUI, observe the consumer group performance for Kafka Connect. You could use this to ensure that Kafka Connect is performing well in your cluster, just like other production traffic.
 - a. In the Control Center GUI, click on Data streams. If at first the graphs don't appear, wait a minute and refresh your browser.



- b. Scroll down to view the consumer group connect-File-Sink-Connector.



12. In each terminal window where the consumer is running, press `Ctrl-c` to terminate the process.



STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Writing a Kafka Streams Application

Project directory: `streams`

In this Hands-On Exercise, you will write a Kafka Streams application to process data from the Topic `shakespeare_topic`. This exercise builds on another Exercise:

- You must have completed the "Using Kafka with Avro" Exercise which ran the `ShakespeareProducer` application. This created the Topic `shakespeare_topic`, where the key of each message is the name of the play and the value is a line from the play.



The Kafka Streams API is currently only available as a Java API. If you do not know Java, please spend the Exercise time investigating the sample solution.

Creating a Streams Application

1. Create a new Topic based on `shakespeare_topic` where each line of the play (i.e., the message's value) is converted to uppercase.
2. Create a new Topic based on `shakespeare_topic` which only contains messages from the play "Macbeth".

If You Have More Time

3. Investigate the code in `shakespeare_example`, which converts `shakespeare_topic` to a new Avro Topic. Write a Kafka Streams application to display the contents of that Topic. (Hint: use the `print()` method.)



STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Data Transformations with KSQL

In this Hands-On Exercise, you will use KSQL to identify patterns in the data and transform it. You will do something similar to the "Using Kafka with Avro" Exercise, where you added the year that the play was written to the data, but now using KSQL instead of the Kafka Producer and Consumer.



This exercise builds on other Exercises:

- You must have completed the "Using Kafka with Avro" Exercise which ran the `ShakespeareProducer` application. This created the Topic `shakespeare_topic`, where the key of each message is the name of the play and the value is a line from the play. If you did not complete that exercise, a catchup script is available:

```
$ confluent_shakespeare_producer.sh
```

- You must have completed the "Connect Pipeline" Exercise which read data from a `sqlite3` database into Kafka. This created the Topic `shakespeare_years`, where the key of each message is `null` and the value is schema'd Json with the name of the play and the year it was written. If you did not complete that exercise, a catchup script is available:

```
$ confluent_connect.sh
```

Explore Data

1. Start KSQL.

```
$ /home/training/ksql/bin/ksql-cli local \
--properties-file /home/training/ksql/ksql.properties
....
ksql>
```



If you run into any issues running KSQL, check for errors in `/tmp/ksql-logs/`

2. List all the Topics in the Kafka cluster. Among many Topics, confirm there are at least the following two Topics: `shakespeare_years` and `shakespeare_topic`.

```
ksql> list topics;
Kafka Topic                                | Registered | Partitions | Partition Replicas
-----|-----|-----|-----
shakespeare_years                         | false      | 1          | 1
shakespeare_topic                         | false      | 1          | 1
```

3. Create a KSQL STREAM called `shakespeare` from the Kafka Topic `shakespeare_topic`, with one column `line`. The `value_format` of the Topic is `DELIMITED`.

```
ksql> CREATE STREAM shakespeare (line STRING) WITH (kafka_topic='shakespeare_topic', \
value_format='DELIMITED');
```

Message

Stream created

4. Show all the STREAMS in KSQL.

```
ksql> show streams;
```

Stream Name	Kafka Topic	Format
SHAKESPEARE	shakespeare_topic	DELIMITED

5. Describe the new STREAM.

```
ksql> DESCRIBE shakespeare;
```

Field	Type
ROWTIME	BIGINT
ROWKEY	VARCHAR(STRING)
LINE	VARCHAR(STRING)

6. Get the first five rows in the STREAM.

```
ksql> SELECT * FROM shakespeare LIMIT 5;
1512499612276 | Julius Caesar |      1
1512499612282 | Julius Caesar |      2  Project Gutenberg Etext of Julius Caesar by
Shakespeare
1512499612282 | Julius Caesar |      3  PG has multiple editions of William Shakespeare's
Complete Works
1512499612282 | Julius Caesar |      4
1512499612282 | Julius Caesar |      5
LIMIT reached for the partition.
Query terminated
```

Enrich Data

7. Create a new KSQL STREAM called `years1` from the Kafka Topic `shakespeare_years`, with two columns `schema` and `payload`. The `value_format` of the Topic is `JSON`.

```
ksql> CREATE STREAM years1 (schema STRING, payload STRING) \
WITH (kafka_topic='shakespeare_years', value_format='JSON');
```

8. Extract the Json fields from the above data into a STREAM called `years2`. The original Topic `shakespeare_years` had no key because of the way the data was imported using the Jdbc source connector, so simultaneously set the key value to be the name value, using `PARTITION BY`.

```
ksql> CREATE STREAM years2 WITH (PARTITIONS=1) AS SELECT \
EXTRACTJSONFIELD(payload, '$.id') AS id, \
EXTRACTJSONFIELD(payload, '$.name') AS name, \
EXTRACTJSONFIELD(payload, '$.year') AS year \
FROM years1 WHERE payload <> 'null' \
PARTITION BY name;
```

9. Convert the STREAM to a TABLE. We want a TABLE because this is just a changelog stream such that each Shakespeare work will have only corresponding row for the year it was written.

```
ksql> CREATE TABLE years_created (id STRING, name STRING, year STRING) \
WITH (kafka_topic='YEARS2', value_format='JSON');
```

10. Get the all the rows from the TABLE `years_created`.

```
ksql> SELECT * FROM years_created;
1512655452014 | Hamlet | 1 | Hamlet | 1600
1512655452019 | Julius Caesar | 2 | Julius Caesar | 1599
1512655452019 | Macbeth | 3 | Macbeth | 1605
1512655452019 | Merchant of Venice | 4 | Merchant of Venice | 1596
1512655452020 | Othello | 5 | Othello | 1604
1512655452020 | Romeo and Juliet | 6 | Romeo and Juliet | 1594
1512655452020 | Antony and Cleopatra | 7 | Antony and Cleopatra | 1606
```

Question:

Did you get the `ksql>` prompt back? Why not?

11. Press `Ctrl-c` to terminate the query.



Due to a known issue, it may take up to three minutes for the query to terminate.

12. Transform the data to get a new Topic called `shakespeare_output` that joins the shakespeare lines and year and matching only the lines that contain the word `strength`.

```
ksql> CREATE STREAM shakespeare_output WITH (PARTITIONS=1) AS \
SELECT line, years_created.year \
FROM shakespeare LEFT JOIN years_created \
ON shakespeare.ROWKEY = years_created.name \
WHERE shakespeare.line LIKE '%strength%';
```

13. Get the first five results of the QUERY shakespeare_output.

```
ksql> SELECT * FROM shakespeare_output LIMIT 5;
1512655256973 | Julius Caesar | 1132 | Can be retentive to the strength of spirit; | 1599
1512655257022 | Julius Caesar | 1674 | Fearing to strengthen that impatience | 1599
1512655257438 | Romeo and Juliet | 813 | Than your consent gives strength to make it fly. | 1594
1512655257489 | Romeo and Juliet | 1678 | Women may fall when there's no strength in men. | 1594
1512655257522 | Romeo and Juliet | 3458 | Jul. Love give me strength! and strength shall help afford. | 1594
LIMIT reached for the partition.
Query terminated
```

14. List all the persistent KSQL queries. You should only see two queries, YEARS2 and SHAKESPEARE_OUTPUT, because they were created with CREATE STREAM AS SELECT statements.

```
ksql> show queries;
```

15. List all the Topics that were created in the Kafka cluster as a result of the above transformations.

```
ksql> list topics;
```

Kafka Topic	Registered	Partitions	Partition Replicas
SHAKESPEARE_OUTPUT	true	1	1
shakespeare_topic	true	1	1
shakespeare_years	true	1	1
YEARS2	true	1	1

16. Exit KSQL.

```
ksql> exit
Exiting KSQL.
```




STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix: Adding Connectors: Alternate Method

In the `Running Kafka Connect` exercise, one of the sections uses the Confluent Control Center to add a source connector and sink connector. If you prefer to use `curl` commands to communicate with the Kafka Connect worker's REST endpoint instead, you may follow the instructions below.

1. Verify that you have already started Kafka Connect in distributed mode.

```
$ sudo jps | grep ConnectDistributed
30163 ConnectDistributed
```

2. View the properties file that we provided for the JDBC source connector.

```
$ less /usr/local/etc/connector-source-jdbc-properties.json
```

3. Add the JDBC source connector.

```
$ curl -X POST -H "Content-Type: application/json" \
--data @/usr/local/etc/connector-source-jdbc-properties.json http://connect1:8083/connectors
```

4. View the properties file that we provided for the File sink connector.

```
$ less /usr/local/etc/connector-sink-file-properties.json
```

5. Add the File Stream sink connector.

```
$ curl -X POST -H "Content-Type: application/json" \
--data @/usr/local/etc/connector-sink-file-properties.json http://connect1:8083/connectors
```

6. Check that both connectors are running.

```
$ curl connect1:8083/connectors
["JDBC-Source-Connector", "File-Sink-Connector"]
```