

# Level Up - Hello (Kafka) World

## Overview

In this "Level Up" session, we're going to get familiar with Apache Kafka by getting our hands on it in a local environment where you have full control to do whatever you want 🐱

Have fun and be curious 💡 Ask questions as you go.

## ✅ Goals

- ☐ Run a Dockerized Kafka Environment
- ☐ Produce Records (CLI & Java)
- ☐ Consume Records (CLI & Java)
- ☐ Explore Control Center

## 🗣️ attn. Windows Users

A few tips for Windows users.

- The terminal you use matters - use PowerShell, git bash, or the terminal in IntelliJ.
- There are Gradle commands used in these sessions that accept a parameter (`-P`) and you will need to put a space after the `P`.
  - For example:
    - Mac users can run `./gradlew -Pstream=...`
    - Windows users must run `./gradlew -P stream=...`
- The `kafka-console-consumer` commands in this session include `--tty` to ensure the consumers terminate appropriately. Windows users should remove the `--tty` (not 100% sure why, but previous groups have found `--tty` to cause issues)

## 1) Start Kafka (Confluent Platform)

Before continuing, you must download [stream-processing-workshop](#)

The `stream-processing-workshop` that you just downloaded has a streamlined Confluent environment that we're going to start up now via Docker Compose.

This environment will start up -

- **Apache Kafka Broker (x1)** - the heart of your cluster, accepting and distributing events.
  - "Real" deployments will have 3+ Brokers typically
- **Schema Registry** - manages & validates event schemas.
- **Control Center** - web-based tool for monitoring and managing Kafka.
- **ZooKeeper** - primarily used for leader elections (removed in recent release)

In your terminal, navigate into the `stream-processing-workshop/local` directory. You should see a `docker-compose.yaml` file.

From the directory containing the `docker-compose.yaml` file, run `docker compose up -d`

Depending on your version of Docker Compose, this command may be `docker-compose up -d` (with a dash between docker-compose instead of a space)

You should see some logs similar to below. The first time that you start the cluster, Docker is going to have to download some (beefy) Confluent Docker images. This may take a few minutes.

```
> cd local
> docker compose up -d

[+] Running 5/5
 :: Container zookeeper      Healthy
 :: Container broker         Healthy
 :: Container schema-registry Healthy
 :: Container control-center Started
```

To validate that the environment started up, let's do 2 things.

1. From your terminal, run `docker-compose ps` (or `docker compose ps`) to view running containers.

```
# run from stream-processing-workshop/local dir
> docker compose ps
NAME                SERVICE                STATUS
broker              broker                 running (healthy)
control-center      control-center         running
schema-registry     schema-registry        running (healthy)
zookeeper           zookeeper              running (healthy)
```

After a few minutes, if the state of any component isn't **Running** or **Up**, run the `docker compose up -d` command again, or try `docker compose restart <image-name>`, for example:

```
docker-compose restart control-center
```

2. If all of the containers seem to be running (healthy), navigate to [Control Center](http://localhost:9021) at `http://localhost:9021` and poke around a bit. Explore as much as you want but you should at a minimum see these few things -
  1. A single cluster named "controlcenter.cluster"
  2. When clicking on the "controlcenter.cluster" cluster, 1 Broker
  3. When clicking on Topics, No Topics
    - If you unselect "Hide internal topics", you should see the topics used by CC itself to manage its own state. Kafka and supporting tools heavily rely on Kafka itself for state, which is pretty cool!

## 2) Produce / Consume Records - CLI

Ok - your environment is up (see previous section if it's not). Now we're going to leverage a few of the CLIs (Command Line Interfaces) to interact with the cluster. The Kafka CLIs offer a variety of capabilities and are a quick way to explore key concepts without any layers of abstraction.

You might be thinking "Oh Em Gee, I didn't download the CLIs!" -- well you're in luck, they come pre-baked on the Kafka Broker Docker Container that **you're already running** so we're going to `docker exec` onto the broker and run the commands from inside that container 🙋

1. So first, create a Kafka Topic by running the command below in your terminal. The `docker exec broker` portion of the command is what will transport you into the `broker` container before running the actual `kafka-topics --create` command. The backslashes (`\`) are simply a way to break a command into multiple lines to make it more readable.

```
docker exec -it broker \
kafka-topics --bootstrap-server broker:9092 \
              --create \
              --topic test.topic

--

Created topic test.topic.
```

2. The topic exists, now produce a record.

Enter the command below and then type text + hit enter. Once again, we first `exec` onto the broker before utilizing the `kafka-console-producer` CLI.

```
docker exec -it broker \
kafka-console-producer --bootstrap-server broker:9092 \
    --topic test.topic

--

>hello
>world
>bye

# CONTROL+C to exit
```

The previous command produced records **without keys**, but we know keys are important when ordering matters!

To produce events with keys, we'll use a similar command as before but add a couple more properties. Enter the command below and then type text + hit enter. The text before the comma will be the record key and the text after the comma will be the record value.

```
docker exec -it broker \
kafka-console-producer \
    --topic test.topic \
    --bootstrap-server broker:9092 \
    --property parse.key=true \
    --property key.separator=","

--

>hello,world1
>bye,world2

# CONTROL+C to exit
```

3. There are records on `test.topic`, now lets consume them.

We'll leverage the `kafka-console-consumer` CLI to do this. Enter the below command to consume the records. The `--from-beginning` will ensure that our consumer starts at `earliest` on the topic. By default, consumers start at `latest`. Take out the `--from-beginning` and your consumer will only consume records that arrive after it starts listening.

```
# ATTN WINDOWS USERS - remove the --tty from the command below
docker exec -it --tty broker \
kafka-console-consumer --bootstrap-server broker:9092 \
    --topic test.topic \
    --group test-group \
    --from-beginning

--

hello
world
bye
world1
world2

# CONTROL+C to exit
^CProcessed a total of 5 messages
```

Hold up, the record keys are missing 🤔

To include keys, let's modify the command and add a couple more properties. Run the below command to reconsume the events and include keys (if exists). Non-existent keys will be printed out as "null".

```
# ATTN WINDOWS USERS - remove the --tty from the command below
docker exec -it --tty broker \
kafka-console-consumer \
  --topic test.topic \
  --group test-group-with-keys \
  --bootstrap-server broker:9092 \
  --from-beginning \
  --property print.key=true \
  --property key.separator="-"

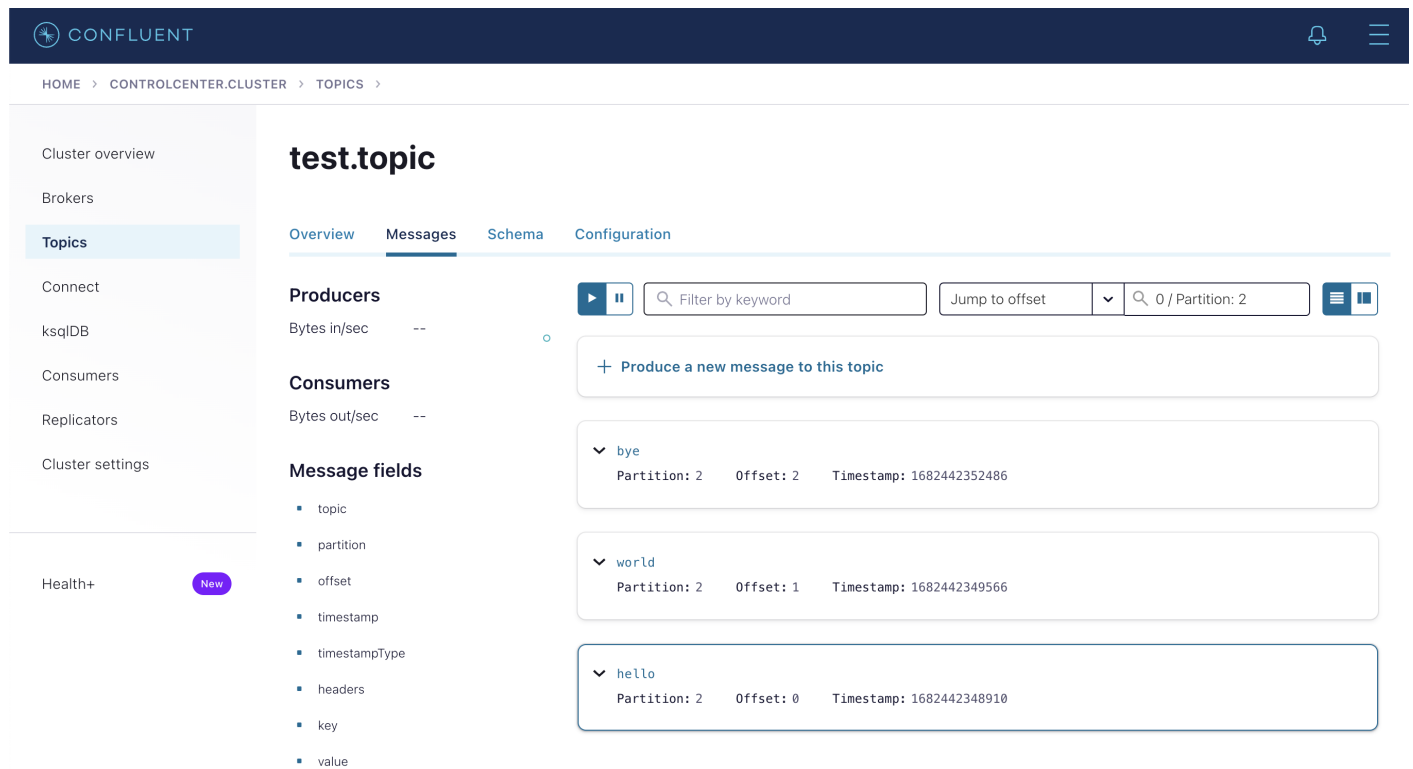
--

null-hello
null-world
null-bye
hello-world
bye-world

# CONTROL+C to exit
^CProcessed a total of 5 messages
```

4. We're all hackerz now, using CLIs to kill the mainframe and bring down the internet (or produce and consume a few records).  
If you get tired of hacking, there's an easy way to explore and interact with your cluster - Control Center.

Open up Control Center (<http://localhost:9021>) and see if you can find `test.topic` and find your messages. Ask a neighbor if you're having trouble finding the Topics page or the records within the topic (hint 'Jump to offset').



The screenshot shows the Confluent Control Center interface. The sidebar on the left contains navigation links: Cluster overview, Brokers, Topics (highlighted), Connect, ksqlDB, Consumers, Replicators, and Cluster settings. The main content area is titled 'test.topic' and has tabs for Overview, Messages (selected), Schema, and Configuration. Under the Messages tab, there are sections for Producers, Consumers, and Message fields. The Message fields section lists: topic, partition, offset, timestamp, timestampType, headers, key, and value. On the right, there are controls for filtering by keyword, jumping to an offset, and a button to 'Produce a new message to this topic'. Below these are three message entries:

- bye**: Partition: 2, Offset: 2, Timestamp: 1682442352486
- world**: Partition: 2, Offset: 1, Timestamp: 1682442349566
- hello**: Partition: 2, Offset: 0, Timestamp: 1682442348910

(Optional) Consumer Group Exploration

We talked about consumer groups enabling seamless distributed processing. Lets inspect your consumer group in a little more in depth.

We're going to list out all groups and then describe our specific group. This will tell us details about the topics/partitions it's consuming, the beginning/ending offset of each partition, and where the group is at on it. These details factor into the 'lag' (the number of records between current offset and end offset) a consumer current has. Lag is an important metric when monitoring Kafka applications.

Run the commands in the code snippet below in order. Each command has a comment above it explaining what we're doing.

```
#####
# list all consumer groups (we should see 'test-group' in the list)
docker exec -it broker \
kafka-consumer-groups \
  --bootstrap-server broker:9092 \
  --list

--

# output
test-group
test-group-with-keys
_confluent-controlcenter-7-3-3-1
_confluent-controlcenter-7-3-3-1-command

#####
# describe the 'test-group' consumer group
docker exec -it broker \
kafka-consumer-groups \
  --bootstrap-server broker:9092 \
  --group test-group \
  --describe

--

# output with GROUP, CONSUMER-ID, HOST, CLIENT-ID removed for brevity
TOPIC      PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
test.topic 0           2               2               0
test.topic 1           3               3               0
test.topic 2           0               0               0

#####
# reset a consumer group back to "earliest" (beginning) on 'test.topic'
docker exec -it broker \
kafka-consumer-groups \
  --bootstrap-server broker:9092 \
  --group test-group \
  --reset-offsets \
  --to-earliest \
  --topic test.topic \
  --execute

--

# SUCCESS output with GROUP removed for brevity
TOPIC      PARTITION  NEW-OFFSET
test.topic 0           0
test.topic 1           0
test.topic 2           0

# ERROR output you will receive if console-consumer was not terminated
Error: Assignments can only be reset if the group 'test-group' is inactive, but the current state is
```

Stable.

```
#####
# describe the 'test-group' consumer group
docker exec -it broker \
kafka-consumer-groups \
  --bootstrap-server broker:9092 \
  --group test-group \
  --describe


--
# output with GROUP, CONSUMER-ID, HOST, CLIENT-ID removed for brevity
TOPIC      PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
test.topic 0          0               2               2
test.topic 1          0               3               3
test.topic 2          0               0               0

# if you restarted the consumer above and rerun the consumer-group describe command you should see the lag
is back to 0
```

### 3) Produce / Consume Records - Java

So far, we've only worked in the CLI. This is a nice way to quickly get to know Kafka but for longer running apps you're going to need to dive into a client library using your choice language.

Today we're going to look at Java, the most heavily supported language in the Kafka ecosystem.

Go read through this  [SimpleKafkaProducer](#). The comments on the code will give you an overview of what's happening.

To run the code,

1. Open the project in the IDE of your choice and run the class. It has everything it needs to run, connected to your local Kafka cluster.
2. Feeling brave? We can run the class directly for your terminal as well using the command below. Run this command from the project's root directory.

```
# run from the root of stream-processing-workshop
./gradlew -Pstream=org.improving.workshop.simple.SimpleKafkaProducer run

# ATTN WINDOWS USERS! You need a space after the -P as shown below
./gradlew -P stream


--

... app logs

Message sent successfully to topic: test.topic, partition: 0, offset: 5

... more logs
```

If you see the "Message sent successfully" log, a record was produced! The producer will shut down once it's produced the record.

Now go look at this  [SimpleKafkaConsumer](#). Read through the comments of the code to get an overview of what's happening.

Running this class can be done the same as the SimpleKafkaProducer (in your IDE or terminal).

```
# run from the root of stream-processing-workshop
./gradlew -Pstream=org.improving.workshop.simple.SimpleKafkaConsumer run
```

```
# ATTN WINDOWS USERS! You need a space after the -P as shown below
./gradlew -P stream

--

... app logs

RECORD RECEIVED: key = key-bdca7a82-acee-4ed4-a0fb-0a0d51466fbc, value = value-2023-05-
05T17:38:32.357256Z, partition = 0, offset = 5

# CONTROL+C to exit
```

The consumer is a bit clingy and will run until *eliminated* (Ctrl + C) so be assertive and take care of it.

## 4) Stop Kafka (Confluent Platform)

To summarize, we -

- Ran a Dockerized Kafka Environment
- Produced Records with both the CLI & a Java App
- Consumed Records with both the CLI & a Java App
- Got to know the Control Center

Now, lets tear it all down!

Just like before, **from the directory containing the docker-compose.yaml file**, run `docker compose down`

Depending on your version of Docker Compose, this command may be `docker-compose down` (with a dash between docker-compose instead of a space)

```
> cd local
> docker compose down
```

