Kafka-docker-composer

Simple tool to create a docker-compose.yml file for failover testing

Introduction

Confluent has published official Docker containers for many years. They are the basis for deploying a cluster in Kubernetes using CFK, and one of the underpinning technologies behind the Confluent Cloud.

For testing, the containers are convenient for quickly spinning up a local cluster with all the components required, such as the Confluent Schema Registry or the Confluent Control Center. You configure each container via a set of environment variables, using the internal docker network and hostname resolution to reference the various components such as the Kafka broker. This is easiest done using docker-compose, which will create a network for your cluster and allow you to keep all configurations in a single document.

This is easily done if you have a single ZooKeeper or KRaft controller and a single Kafka broker in your cluster.

The problem with a docker-compose file begins when you try to configure a larger cluster, for example, for failover testing. Suppose you want to convince yourself (or demonstrate) how the Kafka cluster gracefully handles the loss of a broker. In that case, you need to configure multiple Kafka brokers in your docker-compose.yml file.

You need to be careful not to reuse the same hostname or external port number, and when referencing, for instance, your bootstrap servers, you need to ensure to use the correct hostnames and ports. If you copy/paste and edit your entries, you may mistype or forget an entry. The cluster might still come up but will be degraded.

Of course, you can create a template for a 3 ZooKeeper / 3 Broker cluster and reuse that, but what if you want to test 4 or 6 brokers instead?

This is the situation I found myself in a few years ago with a customer who wanted to test out various configurations. It became very tedious to reconfigure the various docker-compose files for the different scenarios. So, I wrote a Python script that generates the docker-compose file for me: kafka-docker-composer.

How?

Python has a handy templating engine called Jinja2 that allows me to create a template document with variables in it I can populate through my application.

This template simply lists loops through all configured services and populates the required field in the final docker-compose.yml file for each container I want to run. Each container encapsulates a service as a Confluent Server or a ZooKeeper instance, a Schema Registry, and so on.

For each entry, I define the container image to be run as well as the host and container name. Each instance also has a whole set of optional parameters that are defined in the application:

- Health checks so I can define dependencies that will wait for the prerequisites to be healthy.
- Dependencies, which define the order in which containers are created. If health checks are defined as a dependency, this will also be listed here.
- The environment variables that are used for configuring the service within the container.
- Ports that are mapped to the docker host.
- Volumes that are mounted to inject additional files, such as Connector plugins or metrics configurations for monitoring using Prometheus.

The application kafka_docker_composer.py takes a list of arguments and calculates how the template should be populated. It creates the dependencies between the different components, ensures that names and ports are unique, the advertised listeners are correctly set up, and that dependent services like the Schema Registry of Kafka Connect point to the corresponding Confluent Servers.

There are many different configurations, such as using ZooKeeper or KRaft, all controlled by a set of arguments or a configuration file. Here is the help overview

```
❯ python3 kafka_docker_composer.py -h
usage: kafka_docker_composer.py [-h] [-r RELEASE] [--with-tc] [--shared-mode] [-b BROKERS]
[-z ZOOKEEPERS] [-c CONTROLLERS] [-s SCHEMA_REGISTRIES] [-C CONNECT_INSTANCES] [-k
KSQLDB_INSTANCES] [--control-center] [--uuid UUID] [-p] [--kafka-container KAFKA_CONTAINER]
[--racks RACKS]
                                [--zookeeper-groups ZOOKEEPER_GROUPS] [--docker-compose-file
DOCKER_COMPOSE_FILE] [--config CONFIG]

Kafka docker-compose Generator

options:
  -h, --help            show this help message and exit
  -r RELEASE, --release RELEASE
                        Docker images release [7.6.0]
  --with-tc             Build and use a local image with tc enabled
  --shared-mode         Enable shared mode for controllers
  -b BROKERS, --brokers BROKERS
                        Number of Brokers [1]
  -z ZOOKEEPERS, --zookeepers ZOOKEEPERS
```

```
                        Number of ZooKeepers [0] - mutually exclusive with controllers
  -c CONTROLLERS, --controllers CONTROLLERS
                        Number of Kafka controller instances [0] - mutually exclusive with
ZooKeepers
  -s SCHEMA_REGISTRIES, --schema-registries SCHEMA_REGISTRIES
                        Number of Schema Registry instances [0]
  -C CONNECT_INSTANCES, --connect-instances CONNECT_INSTANCES
                        Number of Kafka Connect instances [0]
  -k KSQLDB_INSTANCES, --ksqldb-instances KSQLDB_INSTANCES
                        Number of ksqlDB instances [0]
  --control-center      Include Confluent Control Center [False]
  --uuid UUID           UUID of the cluster [Nk018hRAQFytWskYqtQduw]
  -p, --prometheus      Include Prometheus [False]
  --kafka-container KAFKA_CONTAINER
                        Container used for Kafka, default [cp-server]
  --racks RACKS         Number of racks among which the brokers will be distributed evenly
[1]
  --zookeeper-groups ZOOKEEPER_GROUPS
                        Number of zookeeper groups in a hierarchy [1]
  --docker-compose-file DOCKER_COMPOSE_FILE
                        Output file for docker-compose, default [docker-compose.yml]
  --config CONFIG       Properties config file, values will be overridden by command line
arguments
```

Examples and Use Cases

This is all easier explained through some use cases.

Simplest cluster

```
❯ python3 kafka_docker_composer.py -b 1 -z 1
Generated docker-compose.yml
```

We just created a docker-compose file for the simplest case: 1 ZooKeeper and 1 Broker using the latest Confluent Docker images, as time of writing 7.6.0.

Not convinced? Try it out

```
❯ docker compose up -d
❯ docker compose ps --format "table {{.Name}}\t{{.Image}}\t{{.Status}}"
NAME           IMAGE                              STATUS
kafka-1        confluentinc/cp-server:7.6.0       Up 9 minutes (healthy)
zookeeper-1    confluentinc/cp-zookeeper:7.6.0    Up 9 minutes
```

You might have to use “docker-compose” instead of “docker compose” on your platform.

The broker ports start with 9091 for the first broker, use kafka-topics to create and list topics:

```
❯ kafka-topics –bootstrap-server localhost:9091 --list
❯ kafka-topics –bootstrap-server localhost:9091 --create --topic test-topic \
    --replication-factor 1 --partitions 1
```

KRaft Controller

ZooKeeper is deprecated; modern versions of Kafka prefer KRaft. Just change the option from zookeeper to controller:

```
❯ python3 kafka_docker_composer.py --brokers 1 --controllers 1
Generated docker-compose.yml
```

Look inside the generated docker-compose.yml file to see which environment variables you must set to create a Controller-Broker pair successfully. Do you really want to set this by hand?

A more realistic cluster

Creating single broker setups is nice, but more is needed to show the power of this tool. A minimum standard cluster consists of three controllers and three brokers:

```
❯ python3 kafka_docker_composer.py -b 3 -c 3
Generated docker-compose.yml
❯ docker compose up -d
❯ docker compose ps --format "table {{.Name}}\t{{.Status}}\t{{.Ports}}"
NAME            IMAGE                          STATUS
controller-1    Up About a minute             9092/tcp, 0.0.0.0:19091->19091/tcp
controller-2    Up About a minute             9092/tcp, 0.0.0.0:19092->19092/tcp
controller-3    Up About a minute             9092/tcp, 0.0.0.0:19093->19093/tcp
kafka-1         Up About a minute (healthy)   0.0.0.0:9091->9091/tcp,
0.0.0.0:10001->10001/tcp, 9092/tcp, 0.0.0.0:10101->8091/tcp
kafka-2         Up About a minute (healthy)   0.0.0.0:9092->9092/tcp,
0.0.0.0:10002->10002/tcp, 0.0.0.0:10102->8091/tcp
kafka-3        Up About a minute (healthy)   0.0.0.0:9093->9093/tcp, 9092/tcp,
0.0.0.0:10003->10003/tcp, 0.0.0.0:10103->8091/tcp
```

Note that each broker has its own externally visible port mapped to the host so that you can access each broker individually.

What about the other ports? They are for the JMX agents if you want to configure Prometheus and Grafana for your little cluster:

```
❯ python3 kafka_docker_composer.py -b 3 -c 3 -p
Generated docker-compose.yml
❯ docker compose up -d
❯ docker compose ps --format "table {{.Name}}\t{{.Status}}\t{{.Ports}}"
NAME           IMAGE                           STATUS
controller-1   Up About a minute             9092/tcp, 0.0.0.0:19091->19091/tcp
controller-2   Up About a minute             9092/tcp, 0.0.0.0:19092->19092/tcp
controller-3   Up About a minute             9092/tcp, 0.0.0.0:19093->19093/tcp
grafana        Up 8 seconds             0.0.0.0:3000->3000/tcp
kafka-1        Up About a minute (healthy)   0.0.0.0:9091->9091/tcp,
0.0.0.0:10001->10001/tcp, 9092/tcp, 0.0.0.0:10101->8091/tcp
kafka-2        Up About a minute (healthy)   0.0.0.0:9092->9092/tcp,
0.0.0.0:10002->10002/tcp, 0.0.0.0:10102->8091/tcp
kafka-3        Up About a minute (healthy)   0.0.0.0:9093->9093/tcp, 9092/tcp,
0.0.0.0:10003->10003/tcp, 0.0.0.0:10103->8091/tcp
prometheus     Up 8 seconds             0.0.0.0:9090->9090/tcp
```

As you can see, Prometheus is exposed on port 9090 and Grafana on port 3000. Try out Grafana by pointing your browser to http://localhost:3000. The user and password are admin/admin. The dashboards need updating; you can always find the latest versions in this repository.

The exporter configuration files and dashboards are in the **volumes** directory, as are the exporter jar, so you do not have to download anything separately.

Other components

In addition to the brokers and controllers, you can add Confluent Schema registries, Kafka Connect worker nodes, KSQLdb nodes, and the Confluent Control Center to the mix. You will probably up the memory of your docker environment, specifically, if you run this on your notebook with Docker Desktop.

My Docker Desktop is configured with 8 cores and 16 GB of memory, which gives me ample room to run a large cluster in Docker compose.

The health checks are built for this purpose. If, for example, the schema registry starts up before the broker is finished booting, it will fail since it cannot create its topic, and it will not try again. Ensuring the brokers are up and running and ready to receive clients ensures that the dependent components do not fail on startup.

Connector Plugins

One specific component is the Connect cluster. This cluster comes with a bare-bone set of connector plugins installed. Still, it turned out that by mapping a volume containing unzipped connector plugin jar files before starting up the cluster, I could easily use the same setup to test various connectors cheaply. I have added a few connector plugins as examples, such as the Dategen Connector, which is useful to act as a producer for testing without external dependencies.

Multi-DC scenarios

The original purpose of the kafka-docker-composer was to test failover scenarios in multiple data centres. For this purpose, there are two additional options: racks and zookeeper groups.

Specify the number of racks for the number of unique racks or data centres you want to test with. A typical example is to choose 3 racks, which is a common setup for production clusters. The configured brokers will then be assigned round-robin to the racks. This is particularly useful if you configure more brokers than the number of racks to test the distribution of partitions across the different racks.

You can then test what happens if a data centre goes down by using standard docker-compose methods to stop the containers. Attach a producer and consumer to the cluster to convince yourself that the cluster is still accessible and capable of processing requests. This is how I demonstrate to colleagues, partners, and customers the resilience features of Apache Kafka.

The other option is for configuring ZooKeeper groups, specifically for a 2-DC scenario with hierarchical groups, for which you will need 6 ZooKeeper instances minimum. The tool will calculate the distribution and set up the docker-compose file accordingly.

Further features and outlook

Looking through the help list, you might notice a few other arguments that have not been discussed.

Shared mode

This is an experimental feature for upgrading KRaft Controllers to full brokers. This is not officially supported for production environments, but if you want to play with it, you can use this argument. If nothing else, it will show you which additional environment variables you need for this setup.

UUID

A cluster running with KRaft needs a UUID to identify membership for controllers and brokers. I have pre-created and hardcoded such a UUID, but if you want to change this value, use this option.

TC

I have built the infrastructure for it but have yet to do any testing with it. This option enables a build for all components to create a new image that contains the **tc** tool that can be used to inject latency. There is a Medium article that explains a little more about this feature if you are interested.

I have verified that tc is running successfully in each image, but I have not done any latency testing yet. If you want to try this out and share your setup and experience, please drop me a line.

Security

I have yet to add any authentication or authorization features to this tool, mostly because I have other tools to test and demonstrate security. Still, it might be a worthwhile project, certainly for SASL/PLAIN or even SASL/SCRAM. TLS certificates are a bit trickier because I'd need a new image to generate these. I want the whole project to start with a single **docker-compose up**, not a script.

The same is true for Kerberos and LDAP for RBAC, which would probably require a Samba service in a separate container and some configuration. Let me know in the comments or file an issue in GitHub.

Conclusion

I have been using and extending kafka-docker-composer for the last 5 years whenever the need arose to demonstrate how to set up a cluster in Docker. The main purpose of this tool was to show how resilient a Confluent Cluster is even during a large outage.

Lately, I have used the same tool to teach myself KRaft, experiment with it and use the setup for connector testing and development.

I'd like to know what you will use this tool for. Let me know in the comments or by contacting me directly.

Happy hacking!