

ACIT 3855 – Lab 8

Containerization (Docker and Docker Compose)

Instructors

- Mike Mulder (mmulder10@bcit.ca)
- Tim Guicherd (tguicherd@bcit.ca)

Due date: demo and submission by end of next class.

Purpose

- Use Docker to containerize our microservices into self-contained images
- Use Docker Compose to deploy our microservices

Part 1 – Requirements and Dockerfile

Create Dockerfiles for each of your Python based microservices. You will need a `requirements.txt` that includes all the 3rd party packages required to run each service.

Option 1: Use the same `requirements.txt` file for all services

Since your services have similar requirements, you may want to take advantage of Docker's layered builds and use the same dependencies on all your services. Think about the advantages and drawbacks of that solution.

Option 2: Use a different `requirements.txt` for each service

Create your `requirements.txt`

For this lab, it may be easier to let Python resolve dependencies rather than pinning specific version numbers - even if it is not the recommended way. Your `requirements.txt` may not have version numbers, only package names. Think about the advantages and drawbacks of each option.

All services will require:

- `connexion[flask]`
- `connexion[uvicorn]`
- `connexion[swagger-ui]`

- you should be able to install all by using `connexion[flask,uvicorn,swagger-ui]`

The receiver service also needs:

- `pykafka`

The storage service also needs:

- `sqlalchemy`
- `mysqlclient`
- `pykafka`

The processing service also needs:

- `httpx`
- `apscheduler`

The analyzer service also needs:

- `httpx`
- `pykafka`

Note: the requirements may depend on the way you implement your microservices. Use the above as a guide, and adjust to your needs!

Create a `Dockerfile`

Create a Dockerfile for your receiver service. You should:

- use the `python` base image (https://hub.docker.com/_/python)
- copy the requirements.txt to your container
- install the dependencies from your requirements.txt file
- run your Connexion application (i.e., app.py) when the container is run

Here is an example:

```
FROM python:3

LABEL maintainer="tguicherd@bcit.ca"

RUN mkdir /app

# We copy just the requirements.txt first to leverage Docker cache
# on `pip install`
COPY ./requirements.txt /app/requirements.txt

# Set the working directory
```

```

WORKDIR /app
# Install dependencies
RUN pip3 install -r requirements.txt
# Copy the source code
COPY . /app

# Change permissions and become a non-privileged user
RUN chown -R nobody:nogroup /app
USER nobody

# Tells on which port the service listens in the container
EXPOSE 8080

# Entrypoint = run Python
ENTRYPOINT [ "python3" ]

# Default = run app.py
CMD [ "app.py" ]

```

You will use a similar Dockerfile for each one of your services.

Part 2 – Building and running containers with Compose

At the root of your project folder (i.e. not in any of the services folders), create (or update) the `docker-compose.yml` file. Add each one of your services, and mark them as `build` services (rather than using an image). Make sure you keep the definitions for Kafka, ZooKeeper and MySQL in the Compose file!

```

services:
  receiver:
    build:
      context: receiver
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
  storage:
    build:
      context: storage
      dockerfile: Dockerfile
    ports:
      - "8090:8090"

```

Build the services

Run `docker compose up -d --build`. All images that need to be built will be, and Docker will spin up your services. Look at the log files, and Docker output in the terminal. If necessary, run `docker compose up` (without `-d`) to prevent the containers from going into the background.

Make changes to the code

- All your `app.py` used to run and be accessible on `localhost`.
- They now run in a container, with a dedicated network stack. **You can't run them on localhost anymore!**
- Make sure the `app.py` listens on all network interfaces: `app.run(port=8080, host="0.0.0.0")`.
- Make sure the `EXPOSE`d port in the Docker Compose file matches the one you have in your service.
 - `EXPOSE` does not do anything, apart from letting container users know which port is supposedly used in the container.
 - You may want to add port forwarding to all your containers in the Docker Compose file for easier debugging.

Make changes to the configuration files

- Your services are not available on `localhost` anymore.
- But they run within a Docker Compose network, which takes care of DNS resolution.
- That means the **service names** will resolve to IP addresses in the containers.
- If your `docker-compose.yml` has `kafka`, `zookeeper`, `processing` and `storage` as **service names**:
 - `processing` can access storage by using `http://storage:8090/<....>`
 - `storage` can access `kafka` by using `kafka:9092`
 - make sure you change your Kafka listener to the appropriate service name (probably `kafka`) instead of `localhost`!
 - **DO NOT USE IP ADDRESSES IN YOUR CONFIGURATION FILES!** Docker containers are transient and IP addresses change
- Note: you don't need to run your services on different ports anymore - they are running on different containers, hence different IP addresses. You can always forward them as necessary in the Docker Compose file.

You may need to drop your containers and recreate them (including volumes) if you run into issues when setting up the platform: `docker compose rm -v`.

Leverage `docker compose`

Because you are using Compose, you should **NEVER** use regular `docker` commands for this project, or use the container IDs. This is very time consuming, and prone to errors.

USE `docker compose <SERVICE NAME> action`

For example:

- `docker compose exec receiver bash`
- `docker compose exec -u 0 receiver bash`
- `docker compose logs receiver`
- `docker compose logs receiver -n 10`
- `docker compose logs receiver -n 10 -f`
- `docker compose stop receiver`
- `docker compose rm receiver`
- `docker compose rm receiver -v`

Set up dependencies

Your containers depend on one another. For example, you can't run `receiver` or `storage` without `kafka`. You can't run `processing` without `storage`, etc.

Make sure you make changes to your `docker-compose.yml` to describe the dependencies between your services. For example, the following will make `storage` dependent on the services `mysql` and `kafkaqueue`. **Make sure to adjust it to your setup - use the COMPOSE SERVICE NAMES.**

```
storage:
  build:
    context: storage
    dockerfile: Dockerfile
  depends_on:
    - mysql
    - kafkaqueue
```

Part 3 - Test and clean up

Thoroughly test your services and their interactions. Once everything works as expected, you can clean up your platform.

Final file structure

Your project should have a structural similar to the following:

```
.
├── config
│   ├── log_config.yml
│   ├── receiver_config.yml
│   ├── <.....>
│   └── storage_config.yml
```

```

├─ data
│   ├── database
│   │   └─ [...] MySQL files
│   ├── kafka
│   │   └─ [...] kafka files
│   ├── processing
│   │   └─ processing.json
├─ logs
│   ├── receiver.log
│   ├── storage.log
│   └─ <.....>
├─ docker-compose.yml
├─ storage
│   ├── app.py
│   ├── Dockerfile
│   ├── db.py
│   ├── models.py
│   ├── openapi.yml
│   └─ requirements.txt
└─ receiver
    ├── app.py
    ├── Dockerfile
    ├── requirements.txt
    └─ snowreport.yml

```

- Each service has a separate folder for its *code*.
- All *configuration files* are in a dedicated folder.
- Each service has its own *configuration file*.
- All *log files* are in a dedicated folder.
- Each service has its own *log files*.
- Services may use named volumes and bind mounts for data persistence.

Network setup

Only forward ports for services that are publicly accessible:

- `receiver`
- `processing`
- `analyzer`

Persistent volumes: named and bind mounts

Make sure you understand the distinction [between Docker named volumes and bind mounts](#).

We will use:

- **named volumes** for ZooKeeper data
- **bind mounts** for all other data: kafka, database, configuration files and logs

Persistent volume setup: data

Set up volume persistence for all your data-oriented containers:

- The Kafka container uses `/kafka`.
- MySQL uses `/var/lib/mysql`.
- The processing service uses a JSON file.
 - Create a bind-mount volume, and mount it in your processing container
 - Change the configuration to write the JSON file in the directory where the persistent volume is mounted

Persistent data: ZooKeeper and Kafka

- The Kafka cluster has a `cluster ID` to identify itself with ZooKeeper.
- ZooKeeper keeps track of this cluster ID in its data files.
- When you remove named volumes, the ZooKeeper data is erased.
- The `cluster ID` remains in the `meta.properties` file of Kafka.
- When ZooKeeper restarts, the cluster ID it uses does not match the old one used by Kafka.
- The easiest fix is to remove the `meta.properties` file in Kafka's data directory.
- You should create a script to make it all happen in one swift command.

The following commands may be helpful debugging your Kafka + ZooKeeper:

- `docker compose logs kafka -f`
- `docker compose logs zookeeper -f`
- `docker compose exec kafka bash`, then:
 - `kafka-topics.sh --describe --bootstrap-server kafka:9092` (check the partition ID)
 - `kafka-console-consumer.sh --bootstrap-server=kafka:9092 --topic events --partition 0 --offset earliest` (match the partition ID with the value in the previous command)

Persistent volume setup: configuration files

- In your `config` folder, create one folder for each service.
- Put all the configuration files in the relevant folder. This includes at least the `app_conf.yaml`.
- Using `docker-compose.yml`, make these files accessible in the container at the appropriate location.
- Your logging configuration file may be the same for all services - no need to duplicate it.

Persistent volume setup: log files

- Make sure each service has a log file that is accessible from the Docker host.

Warning: file permissions and users

Depending on how you setup your volumes, you may have permission issues, or bootstrapping issues. The typical issues are:

- wrong / inconsistent permissions on files / folders
 - your services likely run as `nobody`.
 - bind mounts will use the permissions set on the Docker host.
 - they may be compatible, but likely not (especially on Windows)
- The solution is **never** to run your services as root, or to `chown -R 777` your files.
- Instead, you should think about your paths and permissions in advance.
- You may have better results if you work on a Linux system, since filesystem permissions will be more coherent.

Grading

- Walkthrough the Docker and Docker Compose setup: 6 marks
 - `Dockerfile`s
 - `docker-compose.yml` file
 - volumes setup
 - port forwarding / network setup
 - only required ports are forwarded
 - services use Docker internal network
- Demo of data persistence: 2 marks
 - change configuration files
 - check logs
 - stop all services, remove the containers and restart all services: all data must persist (kafka, database, JSON stats)
- Demo of your platform with jMeter: 2 marks

Submit the following to get your marks:

- your `docker-compose.yml` file
- a legible screenshot of the `docker compose ps` command showing all services up and running

Total: 10 marks