Documentation on workflow

This assignment is an example of using python3, flask, requests, and docker to make a complete microservice architecture implementation.

**Get started with the application**

• Extract the folder named microservices

• Directory structure :

micro services-flask demo

├── docker-compose.yml

├── book.csv

├── content

├── Dockerfile

├── app.py

└── requirements.txt

├── user

├── Dockerfile

├── app.py

└── requirements.txt

└── user\_interaction

├── Dockerfile

├── app.py

└── requirements.txt

• Run **sudo docker-compose up**

• There are four different docker images that are build for this demo.

• postgres:12

• user

• user\_interaction

• content

• To see whether the containers are up and running open next terminal and type

**sudo docker ps -a**

• There are four container name :

• Postgres 12

• User

• User-interaction

• Content

Overview on the directory files

• In the microservices directory there are 3 more internal directory for each separate services,then there is Documentation file ,book.csv and one docker-compose.yml file.

• Docker compose ​​is a tool that was developed to help define and share multi-container applications. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down. The big advantage of using Compose is you can define your application stack in a file, keep it at the root of your project repo.

• There is one book.csv file used to upload the content in content service.

• Next there are 3 services: directory user,user-interaction,content.

• User directory content .

• app.py : Contains all necessary routes for running user service as flask application.

• Dockerfile : Used to build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands to call on the command line to assemble an image.

• requirements.txt file : Used to install all python and flask necessary packages that are required to run the app.

• User\_interaction directory content .

• app.py : Contains all necessary routes for running user service as flask application.

• Dockerfile : Used to build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands to call on the command line to assemble an image.

• requirements.txt file : Used to install all python and flask necessary packages that are required to run the app.

• Content directory content .

• app.py : Contains all necessary routes for running user service as flask application.

• Dockerfile : Used to build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands to call on the command line to assemble an image.

• requirements.txt file : Used to install all python and flask necessary packages that are required to run the app.

Application architecture :

**Description on the containers**

User service container –

• Contain a Dockerfile, requirements.txt and app.py file for flask app.

• In app.py I used the flask library for defining routes and different methods in application , then I used Flask-SQLAlchemy for ORM and to get more flexibility in writing different sql query and then I used request libraries to make a call and communication between different services .

• I also used logging in every service file that will record every event happening inside the container. Every time a container build up and running it will create a record.log file inside the container service directory to record every event and error happening inside the flask app.

• List of routes in user service :

• POST : localhost:80/user — for creating user

• GET : localhost:80/user – for fetching all user from db

• GET : localhost:80/user/<id> – for fetching user by id

• PUT : localhost:80/user/<id> – for updating used by id

• DELETE : localhost:80/user/<id> – Delete user by id

• I also used a custom error function inside the app to handle the exception and error handling during request.

User Interaction service container –

• This container also contains the basic setup as the user service container did.

• List of routes in user interaction service :

• POST : localhost:81/user\_interaction — for creating user

• GET : localhost:81/user\_interaction – for fetching all user from db

• GET : localhost:81/user\_interaction/<id> – for fetching user by id

• PUT : localhost:81/user\_interaction/<id> – for updating used by id

• PATCH : localhost:81/user\_interaction\_update/<id> –Used for updating the read and like events

• DELETE : localhost:81/user\_interaction/<id> – Delete user by id

• To check if a user exists I used get user by user\_id call to user service.

• GET : http://user:80/user/<id> – Get a used detail if user exit or 400 not found

• To sort max user-intraction with content

• GET : localhost:81/top\_user\_interaction – Return list of content id with user make a interaction and how many user interact with it

• I also used a custom error function inside the app to handle the exception and error handling during request.

Content service container –

• This container also contains the basic setup as the other container did.

• List of routes in content interaction service :

• POST : localhost:82/upload — for uploading csv file

• GET : localhost:82/get\_content – for fetching all user from db

• GET : localhost:82/content/<id> – for fetching user by id

• PUT : localhost:82/content/<id> – for updating used by id

• DELETE : localhost:82/content/<id> – Delete user by id

• For fetching top content :

• GET : localhost:82/top\_content :

• This will call GET : http://user\_interaction:80/top\_user\_interaction

• For fetching top content :

• For sorting recent content :

• GET : localhost:82/get\_recent\_content – return recent content added

Future Scope

• In this assignment I simply configure my Flask app to run with the line app.run(host="0.0.0.0", port=80).

• There are 3 disadvantages of doing this - 1 big and 2 little:

• The main limitation of relying on app.run() is that it uses Flask’s built-in server, which was built for local development and not for use in production. It cannot handle much traffic and isn’t very robust. Apache and Nginx are battle-tested, high quality, robust web servers that are very widely used in production.

• If you run python app.py in an ssh session, your app will close when you close your ssh connection.

• If your server reboots, the app will not start unless you ssh in again and restart it. One way to fix this is by making an upstart job to start the app, but again, you are still using Flask’s little development server.

• For this problem i used Nginx with Gunicorn together because the major advantage of using Nginx and Gunicorn together is that in addition to being a web server, Nginx can also proxy connections to Gunicorn which brings good performance benefits along with the capability to handle a large number of connections with very little CPU usage and memory cost.

Postman api collection

https://www.getpostman.com/collections/f759399f20b6fd424ace

I will also add a json collection with the zip file

Api Documentation link :

Link : https://documenter.getpostman.com/view/15582474/UVeCQoNJ