17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation: 19/05/2020
Evaluator(s): Prof. Venkatesh Prasad
Submission ID: 1161
Automated submission score: 0

SNo	Name	USN	Class/Section
1	Aarya Arun	PES1201700009	C
2	Indu A Rallabhandi	PES1201700795	C

*Automated submission score 0 due to repeated SSH failure

Introduction

RideShare is a cloud based web-app backend built using flask and docker containers hosted on AWS instances. With this, one can add a user to the database, add a ride after validating the user's existence, delete a user, delete a ride, and get details of existing rides as well as joining existing rides if they are a user. The many stages of this project included writing the relevant APIs for users and rides related database queries and modifications, creating containers for both users and rides, and finally creating a Database as a Service by creating an orchestrator that uses master and slave workers that overall function as a scalable database.

RideShare backend also includes other APIs like clearing up the database/resetting it, counting the number of requests sent to it, listing all the users, etc. The rides container must send a request to a user-related API in order to check if the user exists. Along with this, the APIs were tested using Postman to check for the required return values which consisted of helpful messages and appropriate status codes depending on the values found or returned from the database.

The persistent database that we used for our assignments leading up to the project was MySQL, but it was changed to SQLite for the project for more simplicity.

Related work

The sites we referred to for the project are:

- Mysql tutorial: https://dev.mysql.com/doc/
- Rabbitmq tutorial: https://www.rabbitmq.com/getstarted.html
- Flask tutorial: https://flask.palletsprojects.com/en/1.1.x/tutorial/
- Zookeeper tutorial: https://zookeeper.apache.org/doc/r3.6.0/index.html
- Docker tutorial: https://docs.docker.com/
- Docker-compose tutorial: https://docs.docker.com/compose/install/
- SQLite tutorial: https://www.sqlite.org/docs.html

ALGORITHM/DESIGN

Our system is designed such that it has three instances: one for users, one for rides and one for the orchestrator. The rides instance contains the rides container, the users instance has the users container. The orchestrator instance has 5 containers - The orchestrator container, rabbitmq container, zookeeper container, master container and slave container. There are two target groups, one for the rides instance and one for the users instance. The two target groups are used to create a load balancer. The conditions for the load balancer are as follows: If the route is /api/v1/users* then the load balancer forwards the request to the user target group. Else the request is forwarded to the rides target group.

TESTING

Here are some of the challenges faced during testing:

1. 500 internal server error during submissions, especially during the first assignment. Despite having good internet issues it could not get fixed.
2. Repeated SSH failure for automated submissions, which we did not know how to fix even after running chmod on the .pem files
3. During the first assignment we had a lot of key errors in submission which we had to rectify
4. Making the load balancer work during the third assignment was very difficult

CHALLENGES

Here are some of the challenges we faced:

1. Using mysql. As mysql required its own server to be run, we initially used to SSH into an instance a second time to start the server before sending any queries. This became a problem for the last assignment where scaling was to be expected hence we shifted to SQLite which was file-based and easier

2. Many times when we installed packages like pika and kazoo, on running docker-compose build and docker-compose up, the output would show that there was no such package like pika or kazoo and give import errors. This had to be fixed
3. Returning the right datatypes. While returning data through the orchestrator it was very difficult to get the datatypes right and make sure that what was returned finally to Postman was the correct response
4. Memory errors. Many times after introducing the rabbitmq containers we had the instances either hang or terminate the docker containers due to memory errors. By using docker volume prune, docker container prune, and docker system prune, we were able to solve this issue.

Contributions

APIs - Aarya

Building and running containers - Indu

RabbitMQ container - Indu

Orchestrator and workers- Aarya

CHECKLIST

SNo	Item	Status
1.	Source code documented	CHECK
2	Source code uploaded to private github repository	CHECK
3	Instructions for building and running the code. Your code must be usable out of the box.	sudo docker-compose up CHECK