A2 - Zoomer (Microservices)

DESCRIPTION

After graduating from UTM, you and a group of your peers decide to take on the ride-sharing market with a revolutionary new web app called *Zoomer.* You (and your partner, if you choose to pair up) are the CTO(s) of *Zoomer*! This app allows you to request rides - matching you with drivers in the nearby area to safely take you to your destination.

Your team has successfully completed the frontend of the application so only the backend APIs are still to be done. As CTO, you will be creating an application with a microservices architecture, containing **Three Microservices** (backend Java APIs). You will be using PostgreSQL, MongoDB, and Neo4J databases for the three microservices respectively, and will be implementing this assignment using the Java Maven framework. You will also implement an **API Gateway** to interface with the three microservices. Remembering your teachings from CSC301, you will also dockerize everything (provided in the starter code).

OBJECTIVE

1. To understand general design and purpose of microservice architecture
2. To understand and implement an API gateway
3. To create REST API endpoints that are supported by Neo4j graph, MongoDB, and PostgreSQL databases
4. To utilise git, git flow, and code style correctly
5. To further understand how docker works

MARKING OUTLINE

Microservices	75%
API Gateway	10%
Testing	10%
Git Flow	5%

ENVIRONMENT SETUP

This assignment uses the same Java and Maven versions from A1 so if you've already set those up you should be ready to go. You will also need Docker/Docker Desktop which you should already have installed from A1. Finally, use the link below to view the API docs.

- https://documenter.getpostman.com/view/14483503/UVsLSRqT

The assignment setup should be the same as A1. You can either choose to develop locally or in Docker. If you're using Docker, simply run docker-compose up to get the containers up and running. If you're working locally, just import each microservice into your IDE and run them the same as in A1. If you are working locally, you may also need to download and install MongoDB and PostgreSQL and run the DBs locally. We've also provided environment variables just like in A1 so that the microservices will connect to their respective databases when the microservices are being run locally without changing the connection url (refer to the A1 handout for more info on the environment variables).

Also, remember to frequently check the piazza for any updates or clarifications on A2.

Submission Information

1. A1 will be submitted through GitHub classrooms, the link can be found on the website
2. You may work alone or in teams of two
3. Make sure your final draft is on your main/master branch before the assignment deadline
4. Make sure your repos file/folder structure is unchanged
5. Make sure your whole assignment runs fully and correctly in Docker
6. Make sure team.md (provided to you in the starter code) is filled out

Starter Files

Do **not** change any code that is already given to you in the starter files (including the pom.xml, dockerfile, docker-compose file, etc) and do not change the file/folder structure of the starter code. We have provided comments in every place where you need to add code. You may also add extra files or import statements as you see fit except in the API Gateway folder (see comments in the code for more info).

Note for M1 Mac Users

If you're using an M1 Mac and your Docker containers aren't working properly, you could try to uncomment the line '`#platform: linux/amd64`' in the docker-compose file and restart your docker containers which might fix the problem.

Data Access Object Pattern (DAO)

You will be following the Data Access Object (DAO) Pattern for this assignment as well. This means all interactions with the database must be from a single object. This object class is given to you in each microservice and you must populate the class with the database methods that you will need. If you make any database transactions without using the DAO, you may lose marks.

Frontend

The frontend is completely for you to play around with and to test your code through a real frontend. You will have no work to do in the frontend so you can choose to completely ignore the frontend if you so wish. The frontend is also a nice sanity check, as it should work as intended with the right API.

Git Usage

You must use Git flow for this assignment and you will be evaluated on correct usage of Git flow. This means you will use a development branch along with feature branches to implement features. You will have to decide what you consider to be a feature. You must also use pull requests any time you merge branches.

Continuous Integration / Continuous Delivery Testing

This part will be the same as A1. For each endpoint that you implement, you should have one test to test a successful response (200 status code) and one test to test a fail response (4XX status code). You don't need to write tests for the endpoints that are already given in the starter code. All of these tests should be requests that go to the API Gateway meaning that they should **NOT** be going directly to the microservice. You also won't lose marks if tests fail, they are more so for you to test your own code. You can name them however you want but your naming should be clear with which endpoint the test is for and if the test is testing a 200 response or a 4XX response. The tests for a certain microservice should go in the AppTest.java file under the src/test folder of that respective microservice. The test API requests must originate from outside of Docker and go to your server running inside Docker, meaning you will run your tests from outside the Docker containers to test your servers running inside the Docker container. Your tests only have to check the response code and response body and you don't have to test if the database is updated correctly. It is also fine if your tests rely on previous tests.

API Gateway

This portion of the assignment will require you to implement an API Gateway. Here's a helpful video to explain what an API Gateway is: https://www.youtube.com/watch?v=1vjOv_f9L8I.

The Gateway will simply be another HTTP server that acts as a proxy server between the user and the microservice APIs. The Gateway should route the user's requests to the services that can handle those requests and should route the microservice's response back to the requestor.

The requests that are sent to the gateway should be exactly the same as the requests that would be sent normally to the services i.e. the URI, request body, URL parameters, and request method of the request that is sent to the gateway should all be the same as the request that would be sent to the APIs directly. In this sense, the gateway will essentially be forwarding the requests to the correct host at the correct port number and sending the response back to the requestor.

Hint: If a service running in a docker container wants to send a request to/connect to a service running on another docker container, you wouldn't use 'localhost'. The host name that you would use would be the name of the actual destination container and the port number would be the port that the destination service is running on INSIDE the container, NOT the port number that it is binded to on the host machine.

API Requirements

The documentation of the three APIs is given on the Postman link on the website. Your job will be to implement the endpoints that are marked as **TODO**. Most of the location and user microservices are already completed so you will only have to implement two endpoints for location and user microservice. As for TripInfo microservice, you will have to implement the whole service yourself according to the documentation.

In the Postman documentation, we've given you plenty of examples for each endpoint to show you what the expected responses look like. Remember the examples might not cover every edge case, so be sure to thoroughly read the endpoint description and the 'Expected Responses' and 'Edge Cases' sections to make sure you're handling every case correctly. For any other edge cases that aren't mentioned in the docs, you are free to handle them however you want.

Two endpoints, particularly /trip/confirm and /trip/driverTime require you to fetch data from other microservices. Do not directly access another microservices database to fetch this data, rather, you will need to send requests to the endpoints you've created to get the required data. You also don't need to send these requests through the API Gateway. This will be the only two cases of communication between services (other than the API Gateway) so for every other endpoint, it will only deal with its own microservice and database.

Database Schemas

Below are the database specifications. Please do not change any of these in your code.

- Neo4j schema:
    - "**user**" nodes:
        - **uid** (String) - The user's ID
        - **longitude** (Float) - The user's current longitude coordinate
        - **latitude** (Float) - The user's current latitude coordinate
        - **is_driver** (Boolean) - Whether the user is a driver or not
        - **street** (String) - The current street the user is on
    - "**road**" nodes:
        - **name** (String) - The name of the road
        - **has_traffic** (Boolean) - Whether the road has traffic
    - "**road**" nodes are connected by the "**ROUTE_TO**" relationship which has:
        - **travel_time** (Integer) - Time taken to travel on the route
        - **has_traffic** (Boolean) - Whether the route is busy

- MongoDB Schema:
    - **_id** - The ObjectId of the document (i.e. a unique identifier for each document). This id is auto-generated by Mongo when a document is inserted into the DB
    - **distance (Integer)** - The distance of the trip
    - **totalCost (String)** - The total cost of the trip
    - **startTime (Integer)** - The start time of the trip (unix time in seconds)
    - **endTime (Integer)** - The end time of the trip (unix time in seconds)
    - **timeElapsed (Integer)** - The time spend for the entire trip (endTime - startTime)
    - **driver (String)** - The uid of the driver in this trip
    - **passenger (String)** - The uid of the passenger in this trip
    - **Note** The database name should be 'trip' and the collection name should be 'trips'

- Postgres Schema:
    - **uid** - The user's unique identifier
    - **email** - The user's email
    - **password** - The user's password (plain text).
    - **prefer_name** - The user's name
    - **rides** - Number of rides that a user took
    - **isDriver** - Boolean, true if user is driver, false otherwise

Attention: Each database should only be accessible to its own microservice; If there is a need to visit another service's database, you should use the endpoint that you built for that service.

Frequently Asked Questions

Can I change the docker-compose or the dockerfile file? - Don't add anything to these files. If you want to comment stuff out to make the containers start up faster, then sure, but remember to restore the file to its original form since we will be testing your code using the exact docker-compose and dockerfiles that we give you on the starter code.

Can we add imports? - Yes, but not where it's explicitly stated not to (check starter code)

Can we make more classes/files? - No, everything you need should be in the starter code

Can we add more methods? - Helper methods are fine, but don't change the given methods

Can we use other Java/Maven versions? - No

Do I have to use Docker? - No, but the code you hand in should all work in Docker

Can we use multithreading or some other imports to handle inter-service communication? - No

Do I have to check if a user or user id exists/doesn't exist in the other databases? - No

In which cases do we return a 500 and how do we test 500 error cases? - This should be sent for unexpected errors other than the ones already mentioned in the documentation, for example, when a database error or database connection issue occurs, you should send a 500. Look at the given endpoints to see when you should send back a 500.

Do we have to use Neo4j GDS for calculating the shortest path in navigation? - This is given to you in the docker-compose file so you can use it but you don't need to if you wish

Can we use built-in neo4j functions? - Yes

Do the string user ids have a format - No, you can assume they can use any characters

How do I access the postgres db running in Docker - use command ‘docker exec -it postgres psql’

Does the order of the fields in the JSON requests and responses matter? - No

Why can't I connect to Neo4j sometimes? - This may be due to Neo4j not starting up fully due to the time it takes to download and install the GDS library. You will have to wait until you see the Neo4j container fully started up, then everything should work as normal. If you want, you may comment out the GDS part on the docker-compose file (see the first question above)

Why does the frontend hang infinitely? - This is a weird behaviour with how docker composes, so you need to restart the container and that should fix it