17CS352: Cloud Computing – Assignment 1

RideShare on AmazonAWS

During this semester for the cloud computing course, you will develop the backend for a cloud based *RideShare* application**,** that can be used to pool rides.
The *RideShare* application allows the users to create a new ride if they are travelling from point A to point B. The application should be able to

- Add a new user
- Delete an existing user
- Create a new Ride
- Search for an existing ride between a source and a destination
- Join an existing ride
- Delete a ride

The entire application will be deployed on Amazon Web Services using the AWS educate starter account preloaded with $75 of credit that has been provided to you.

Scope of Assignment 1

For this assignment, you should complete the backend processing of *RideShare* using REST APIs on the AWS instance. There is no need to create any Front End.

When creating items on the AWS instance, you can store it in any database of your choice. This part will be addressed in further assignments. You need not necessarily need to store it on a database too. It is enough to store this somehow on the filesystem in whatever format you want.

Deliverables

1. Each one of the APIs given below must be implemented with proper status codes
2. These APIs must be exposed on the public IP address, so that we can run a test script to test for correctness of the functioning of the API. Hence it is important that you must stick to the specification. Major focus of the credits must be on this.
3. Please submit a one page report based on the template. (Template will be shared at a later date)

Marks: 10
Due Date: Feb 1, 2020
Evaluation Criteria

- APIs with proper HTTP status codes (8 marks)
- Deployed on AWS with a static IP on port 80 (1 mark)
- Over and above specification like deploying over a reverse proxy and app server, automated testing, load testing etc (1 mark)

RideShare: REST API specification

- For each API endpoint, the Response body (sent by you) must be in the corresponding JSON format
- For each API endpoint, the Request body (sent by testing suite) will be in the corresponding JSON format
- From the list of relevant HTTP response codes, you must decide which one to send for a given request body. All response codes given for each endpoint will be tested against by the suite. Response codes are for both success and failure cases.
- You must implement the APIs using the given endpoints.
- Note that below, `{}` represents a JSON associate map and `[]` represents a JSON array.
- Do not assume a max size for any array, string, number. Use appropriate data structures in your backend for this. We may test against your APIs with extremely long strings/numbers.
- The “source” and “destination” fields are enums, corresponding to the different localities in Bangalore. Hence they will be integers in the HTTP Requests. The enum of the localities will be shared with you.
- The last two APIs don’t have any strict specification, all the db operations must happen through those APIs. Do not embed any db operations directly in any other API. This is needed for further assignments. Use the last two APIs in all the other APIs to perform any db operation.
- You can use any backend framework in any programming language. Eg. flask, django, Nodejs, Go, Spring Boot etc.
- You can use any database. Eg PostgreSQL,MySQL, MongoDB, sqlite, etc
- The last two APIs are going to be further expanded in upcoming assignments, for now ensure that all database requests go through those two queries only.

1. **Add user**

   Route: `/api/v1/users`
   HTTP Request Method: `PUT`
   Relevant HTTP Response Codes: `201, 400, 405 (500 also if the service is not available?)`
   Request: `{`

```
"username": "userName",
"password": "3d725109c7e7c0bfb9d709836735b56d943d263f"
```

   `}`
   Response: `{}`
   Comments:

   1) The username in the request body must be unique (case-sensitive), otherwise send the appropriate response code from the given list.
   2) The password field must be a SHA1 hash hex (40 chars long, hex digits only, case-insensitive), otherwise send the appropriate response code from the given list.

2. **Remove user**

   Route: `/api/v1/users/{username}`
   HTTP Request Method: `DELETE`
   Relevant HTTP Response Codes: `200, 400, 405`
   Request: `{}`
   Response: `{}`
   Comments:

   1) `username` in the route must exist, otherwise send the appropriate response code from the given list.

   Example:

   A call to `/api/v1/users/xyz` should delete user "xyz".

3. **Create a new ride**

   Route: `/api/v1/rides`
   HTTP Request Method: `POST`
   Relevant HTTP Response Codes: `201, 400, 405`
   Request: `{`

```
// username of the user who is creating the ride
"created_by" : "{username}",

// Timestamp of the ride start time
"timestamp" : "DD-MM-YYYY:SS-MM-HH",

// Source of the ride
"source" : "{source}",
```

```
            // Destination of the ride
            "destination" : "{destination}"
        }
```

Response: `{}`

Comments:

a. Given username must exist, otherwise send appropriate status code.

4. List all upcoming rides for a given source and destination

Route: `/api/v1/rides?source={source}&destination={destination}`

HTTP Request Method: `GET`

Relevant HTTP Response Codes: `200, 204, 400, 405`

Request: `{}`

Response:

```
[
    {
        // unique unsigned number
        "rideId": 1234,

        // username of the user who created the ride
        "username": "{username}",

        // timestamp of the ride start time
        "timestamp": "DD-MM-YYYY:SS-MM-HH"
    },
    {
        ...
    },
]
```

Comments:

1) Source and the Destination in the url params must exist, otherwise send the appropriate response code from the given list.
2) The `rideID` must be globally unique.
3) Timestamp must be in given format.
4) Username must exist.

Example:

A call to `/api/v1/rides?source=21&destination=11` should list all the upcoming rides from source 21 to destination 11.

5. List all the details of a given ride

Route: `/api/v1/rides/{rideId}`
HTTP Request Method: `GET`
Relevant HTTP Response Codes: `200, 204, 405`
Request: `{}`
Response: `{`

```
            // Globally unique rideId
            "rideId" : "{rideId}",

            // username of the user who created the ride
            "created_by" : "{username}",

            // Username of all the users associated with the ride
            "users" : ["{username1}", "{username1}", ..],

            // Timestamp of the ride start time
            "timestamp" : "DD-MM-YYYY:SS-MM-HH",
            "source" : "{source}",
            "destination" : "{destination}"
          }
```

Comments:

b. Given rideId must exist, otherwise send appropriate status code.

Example:

A call to `/api/v1/rides/21` should return all the details of the ride in the given format.

6. Join an existing ride

Route: `/api/v1/rides/{rideId}`
HTTP Request Method: `POST`
Relevant HTTP Response Codes: `200, 204, 405`
Request: `{`

```
            // Username of the user who is trying to join the
ride
            "username" : "{username}"
        }
```

Response: `{}`
Comments:

c. Given rideId and the username must exist, otherwise send appropriate status code.

Example:

A call to `/api/v1/rides/21` should return “200 OK” status code.

7. Delete a ride

Route: `/api/v1/rides/{rideId}`
HTTP Request Method: `DELETE`
Relevant HTTP Response Codes: `200, 405`
Request: `{}`
Response: `{}`
Comments:

d. Given rideId must exist, otherwise send appropriate status code.

Example:

A call to `/api/v1/rides/21` should return “200 OK” status code and the ride must be deleted

8. Write to db

Route: `/api/v1/db/write`
HTTP Request Method: `POST`
Relevant HTTP Response Codes:
Request:

```
{
        // You can change this message body according to the needs of the database.
        “insert” : “data”,
        “column” : “column name”,
        “table”  : “table name”
    }
```

Response: `{}`
Comments:

1) All the db write operations must be through this API, use the message request and response as per your discretion. This API will be used in the further assignments.
2) Use the request message body to build the SQL query(or NoSQL if used)

9. Read from db

Route: `/api/v1/db/read`
HTTP Request Method: `POST`
Relevant HTTP Response Codes:
Request:

```
{
        “table”: “table name”,
```

```
            "columns": ["column name",],
            "where": "column=value"
        }
```

Response: `{}`

Comments:

1) `All the db read operations must be through this API, use the message request and response as per your discretion. This API will be used in the further assignments.`
`2) Use the request message body to construct the SQL/NoSQL database query.`

RideShare: Relevant documentation for the assignment

All the links are just for reference, you are not confined to these methods.

- You can use any framework for creating the APIs. Flask, Django,Nodejs, Spring Boot, GO, Ruby on Rails etc are some popular ones. Some frameworks are easier to get started with while some may seem difficult, they offer many features like an ORM etc.
- For integrating flask with databases, "flask SQLAlchemy" is the most popular ORM. You can use any method of storing data though.
    - https://flask-sqlalchemy.palletsprojects.com/en/2.x/
- Simply running the flask code using `flask run` or `python file.py` creates a development server. This server is not production ready because it has multiple security vulnerabilities and does not perform well when multiple concurrent requests are sent. This should only be used locally, for development. This is not the right way for deploying flask in production. In an ideal production deployment the flask code must be run by an application server (like gunicorn) which must run behind a web server (apache/nginx/caddy) using WSGI. The web server may also serve as a reverse proxy.
    - https://stackoverflow.com/questions/26861761/why-use-gunicorn-with-a-reverse-proxy
    - https://stackoverflow.com/questions/38982807/are-a-wsgi-server-and-http-server-required-to-serve-a-flask-app
    - https://www.javacodemonk.com/part-2-deploy-flask-api-in-production-using-wsgi-gunicorn-with-nginx-reverse-proxy-4cbeffdb
- Create REST APIs in flask. There is a python module named "Flask RESTful" which creates better REST APIs but is slightly difficult to use.

    - https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask
    - https://kite.com/blog/python/flask-restful-api-tutorial/
    - https://medium.com/python-pandemonium/build-simple-restful-api-with-python-and-flask-part-1-fae9ff66a706
- Deploy REST APIs on AWS. The two most common methods of deploying web applications are either using NGINX+gunicorn or using Apache2+mod_wsgi. Both have their own pros and cons. NGINX+gunicorn is considered to be the “modern” way but maybe harder to setup and configure for “newbies”. You can feel free to use any other way to deploy given that you are not running the flask application directly.
    - https://medium.com/ymedialabs-innovation/deploy-flask-app-with-nginx-using-gunicorn-and-supervisor-d7a93aa07c18
    - https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-gunicorn-and-nginx-on-ubuntu-18-04
    - https://www.bogotobogo.com/python/Flask/Python_Flask_HelloWorld_App_with_Apache_WSGI_Ubuntu14.php
    - https://stackoverflow.com/questions/18048318/apache-mod-wsgi-vs-nginx-gunicorn
    - https://www.peterbe.com/plog/nginx-gunicorn-vs-mod_wsgi
- HTTP status codes
    - https://developer.mozilla.org/en-US/docs/Web/HTTP/Status