CMPT 756.211 Team-D Term Project Submission

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Team | Team-D | | | | |
| Section | G100 | | | | |
| Team Member Name | Madana Krishnan V K | Arnab Roy | Abhishek Sundarraman | Gayatri Ganapathy | Anchal Jain |
| Email (SFU) | mvadakan@sfu.ca | arnabr@sfu.ca | asundarr@sfu.ca | gganapat@sfu.ca | anchalj@sfu.ca |
| Email (G-Suite/Gmail) | madana.1rn15cs056@gmail.com | thearnabroy@gmail.com | abhishek.sundarraman@gmail.com | gayatrig.vesit@gmail.com | anchalj.2018@gmail.com |
| Github ID | MadanKrishnan97 | Roy-2605 | abhisheksundarraman | Gayatri-2017 | anchalj2018 |
| Additional Notes |  | | | | |

# 

# Section 1 - Problem Statement

## Problem Domain & Application

### **Great Feasts**

Given the fast paced environment that we are living in and the advent of the technologies that have made our lives dependent on it, we propose the solution for a food delivery service “Great Feasts”. Especially during the Pandemic era, ordering food online at home has increased, and almost become a common thing thus making it one of the most essential services required to be available all the time.

Hence, we are building an application synonymous to the famous UberEats, for connecting the Users who are essentially the users of the application, to the Restaurants, and help them perform the most essential operations. In any Food Ordering and Delivering application, the vital services are creating records for users and restaurants (one-time operation), ordering a food item, making a payment, checking the discounts and finally assigning the order to a driver for delivery of the food. Our application hence focusses on these tasks.

We attempt to provide a reliable, scalable and available Food Delivery service to the users using the concepts learnt in Cloud & Distributed System course.

## 

## Tools and Technologies used

For building a robust application, we intend to use the following tools in our project:

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Tool** | **Use Case** |
|  | Docker | To create containerized applications within Kubernetes. |
|  | Azure Kubernetes | Container orchestration to manage the various Docker containers. |
|  | AWS DynamoDB | To store the persistent records of the entities of the application. |
|  | Istio | A service mesh for secure connection within the distributed system. |
|  | REST API calls | Used for CRUD operations on various entities of the application |
|  | Gatling | For generating the traffic load for load-testing purposes. |
|  | Grafana | For Analyzing and Testing the load performance of our application. |
|  | Prometheus | For obtaining real-time metrics of load. |
|  | Flask | For building a Web-based application. |

We also plan to use the following Programming languages

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Language** | **Use Case** |
|  | Python | For Flask applications. |
|  | Shell Scripts | For executing the makefiles. |
|  | Scala | For Gatling scripts. |

The goal of this application is to allow users to perform basic operations related to Food ordering and Delivery using the following entities.

### Entities:

The entities in our application are:

1. **User:**

A user is a customer who primarily orders food and makes payment transactions. Also, the user views discounts on the orders made.

1. **Restaurants:**

A food outlet consisting of various food items and their prices.

1. **Order:**

Stores the Order information of one food item in a Restaurant made by a User.

1. **Payment:**

Stores payment-related information made by a User for an Order.

1. **Deliveries:**

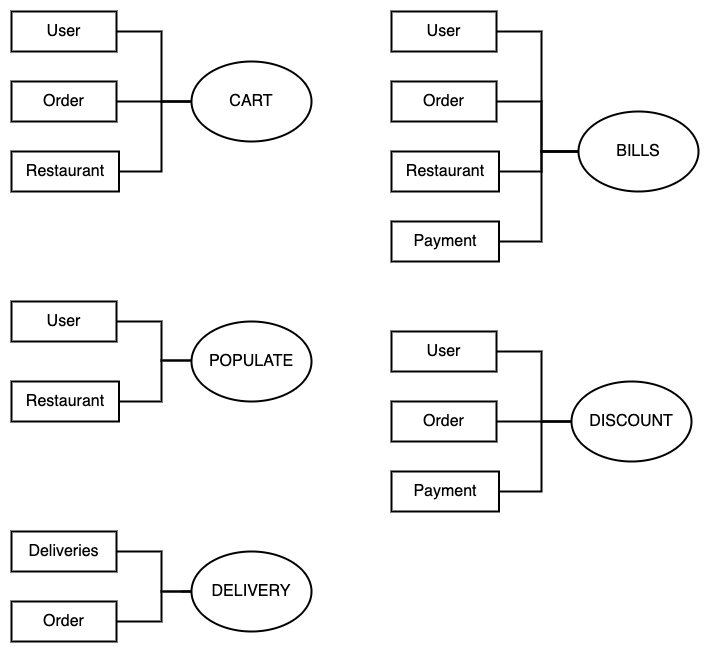
Stores Delivery related information for an Order.

Operations

We have the following 5 operations defined:

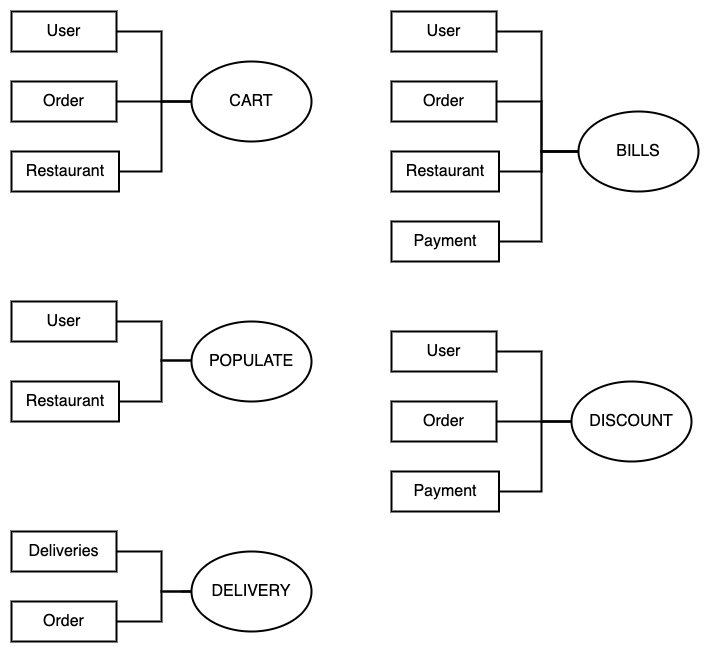
1. **POPULATE**

The admin User can populate restaurant and user details.



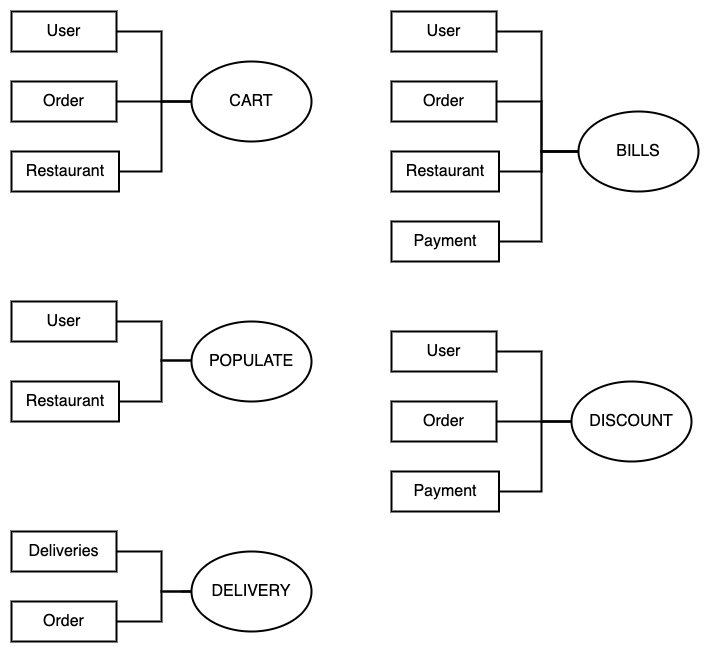
1. **CART**

The User can order food from a Restaurant



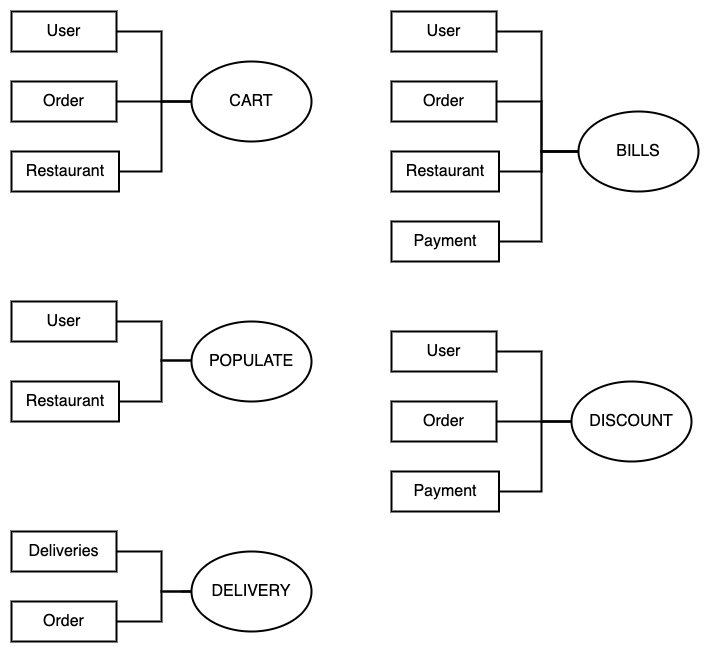
1. **BILLS**

The User can pay the Restaurant



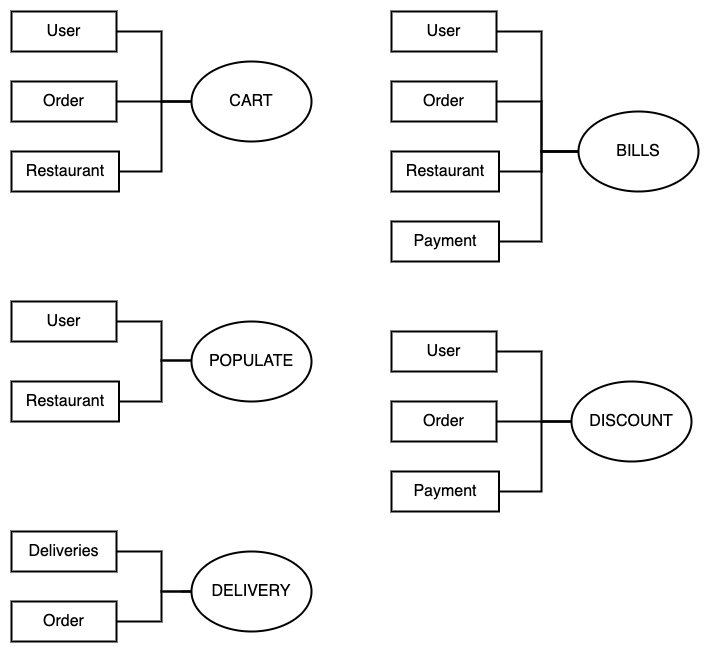
1. **DISCOUNT**

Check if a discount/offer is available for a specific order



1. **DELIVERY**

Each Order is mapped to a Delivery



## Specification of REST API (Microservices Contract)

The following section shows the different REST APIs that will be used in the microservice architecture. Each of the calls represent one operation among the entities involved in the application.

Version: v1

Service: Gateway

Visibility: Public

Domain: Ingress-gateway

Serialized Data/Content-Type: json/xml

|  |  |
| --- | --- |
| **API** | **Description** |
| /api/v1/populate (s1) | Populate Users and Restaurant service URL (30001) |
| /api/v1/cart (s2) | Adding food to Cart from a Restaurant by a User service URL (30002) |
| /api/v1/bills (s3) | Making a Bill Payment by a User for an Order in a Restaurant URL (30003) |
| /api/v1/discount (s4) | Checking Discounts by a User for an Order URL (30004) |
| /api/v1/delivery (s5) | Restaurant assigns an Order to a Delivery Person URL (30005) (db-30000) |

Version: v1

Service: Populate

Visibility: Private

Domain: Populating User, Restaurant

Serialized Data/Content-Type: json/xml

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **API** | **Description** | **Request Body/Parameters** | **Response Body** | **HTTP Response Code** | **Error Codes** | **Request Example** | **Response Example** |
| PUT - /api/v1/populate/login | Login | JSON:{"user\_id": <user\_id>} | Hash of user context suitable for passing to other calls | 200 | 500 | PUT  http://52.139.21.27/api/v1/populate/login | <token> |
| PUT - /api/v1/populate/logoff | Logoff | JSON: {"jwt":<token>} | None | 200 | 500 | PUT  http://52.139.21.27/api/v1/populate/logoff | { Message: ok } |
| PUT - /api/v1/populate/user/<user\_id> | Update user details | Body: {“user\_name”: <user name>, “user\_email”: <user\_email>, “user\_phone”: <user\_phone> }  Header: <User-context token> | OK response | 200 | 500 | PUT http://52.139.21.27/api/v1/populate/user/<user\_id> | { Message: ok } |
| POST - /api/v1/populate/user/ | Add new user | Body: { “user\_name”: <user\_name>, “user\_email”: <user\_email>, “user\_phone”: <user\_phone>}  Params: None | { “user\_id”: string } | 200 | 500 | POST http://52.139.21.27/api/v1/populate/user | {"user\_id":"d555a8cc-8e2d-4579-a15e-0edc986b7690"} |
| DELETE - /api/v1/populate/user/<user\_id> | Delete a user | Header: User-context token  Params: <user\_id> | JSON of response from aws | 200 | 500 | DELETE http://52.139.21.27/api/v1/populate/user/<user\_id> | { ResponseMetadata: <> } |
| GET - /api/v1/populate/user/<user\_id> | Retrieve user details | Header: <User-context token>  Params: “user\_id”<user\_id> | JSON of User entity | 200 | 500 | GET http://52.139.21.27/api/v1/populate/user/<user\_id> | {User Object for given <user\_id>} |
| PUT - /api/v1/populate/restaurant/<restaurant\_id> | Update restaurant details | Body: {“restaurant\_name”: <restaurant\_name>, “food\_name”: <food\_name>, “food\_price”: <food\_price>, }  Header: User-context token  Params: <restaurant\_id> | OK response | 200 | 500 | PUT http://52.139.21.27/api/v1/populate/restaurant/<restaurant\_id> | { Message: ok } |
| POST - /api/v1/populate/restaurant/ | ADD new restaurant | Body: {“restaurant\_name”: <restaurant\_name>, “food\_name”: <food\_name>, “food\_price”: <food\_price>}  Params: None | { “restaurant\_id”: string } | 200 | 500 | POST http://52.139.21.27/api/v1/populate/restaurant | {"restaurant\_id":"1a5defcc-78d6-4b8d-aca3-02f263321ff1"} |
| DELETE - /api/v1/populate/restaurant/<restaurant\_id> | Delete a restaurant | Header: <User-context token>  Params: <restaurant\_id> | JSON of response from aws | 200 | 500 | DELETE http://52.139.21.27/api/v1/populate/restaurant/<restaurant\_id> | { ResponseMetadata: <> } |
| GET - /api/v1/populate/restaurant/<restaurant\_id> | Retrieve restaurant details | Header: <User-context token>  Params: <restaurant\_id> | JSON of Restaurant entity | 200 | 500 | GET http://52.139.21.27/api/v1/populate/restaurant/<restaurant\_id> | {Restaurant Object for given <restaurant\_id>} |

Version: v1

Service: orders (s2)

Visibility: Private

Domain: Adding food items to cart for the order

Serialized Data/Content-Type: json/xml

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **API** | **Description** | **Request Body/Parameters** | **Response Body** | **HTTP Response Code** | **Error Codes** | **Request Example** | **Response Example** |
| POST - /api/v1/cart | Add an order to the cart | Body: { “user\_id”: <user\_id>, “restaurant\_id”: <restaurant\_id>, “food\_name”: <food\_name>}  Header: <User-context token> | {:order\_id”:string} | 200 | 500 | POST http://52.139.21.27/api/v1/orders/ | {"order\_id":"a5781b90-6678-40fb-8965-533b665a9f82"} |
| DELETE - /api/v1/orders/<order\_id> | Delete an order from the cart | Header: <User-context token>  Params: <order\_id> | JSON of response from aws | 200 | 500 | DELETE http://52.139.21.27/api/v1/orders/<order\_id> | { ResponseMetadata: <> } |
| GET-  /api/v1/orders/<order\_id> | Retrieve order details | Header: <User-context token>  Params: <order\_id> | JSON of Order entity | 200 | 500 | GET http://52.139.21.27/api/v1/orders/<order\_id> | {Order Object for given <order\_id>} |

Version: v1

Service: bills

Visibility: Private

Domain: Payment for the food items in the cart

Serialized Data/Content-Type: json/xml

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **API** | **Description** | **Request Body/Parameters** | **Response Body** | **HTTP Response Code** | **Error Codes** | **Request Example** | **Response Example** |
| POST - /api/v1/bills | Add bill for the food ordered | Body: { “payment\_method”: <payment\_method>, “discount\_applied”: <discount\_applied>, “payment\_amount”: <payment\_amount>, “food\_name”: <food\_name>, “user\_id”: <user\_id>, “order\_id”: <order\_id>,”restaurant\_id”: <restaurant\_id> }  Header: User-context token | {“payment\_id”:string} | 200 | 500 | POST http://52.139.21.27/api/v1/bills/ | {"payment\_id":"e46b40fb-b8a7-4d11-b727-f8425c432e25"} |
| DELETE - /api/v1/bills/<payment\_id> | Delete bill from an order | Header: <User-context token>  Params: <payment\_id> | JSON of response from aws | 200 | 500 | DELETE http://52.139.21.27/api/v1/bills/<payment\_id> | { ResponseMetadata: <> } |
| GET-  /api/v1/bills/<payment\_id> | Retrieve payment details | Header: <User-context token>  Params:  <payment\_id> | JSON of Payment entity | 200 | 500 | GET http://52.139.21.27/api/v1/bills/<payment\_id> | {Payment Object for given <order\_id>} |

Version: v1

Service: Discount

Visibility: Private

Domain: Show discount

Serialized Data/Content-Type: json/xml

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **API** | **Description** | **Request Body/Parameters** | **Response Body** | **HTTP Response Code** | **Error Codes** | **Request Example** | **Response Example** |
| GET - /api/v1/discount/show\_discount/<params> | Retrieve discount applied against a bill for an order | Params:: { “payment\_id”: <payment\_id>, “order\_id”: <order\_id>, “Payment ID”:payment\_id}  Header: User-context token | JSON of discount applied | 200 | 500 | GET http://52.139.21.27/api/v1/discount/show\_discount?payment\_id=e46b40fb-b8a7-4d11-b727-f8425c432e25&order\_id=a5781b90-6678-40fb-8965-533b665a9f82&user\_id=d555a8cc-8e2d-4579-a15e-0edc986b7690 | { “10” } |

Version: v1

Service: Delivery

Visibility: Private

Domain: Delivery details

Serialized Data/Content-Type: json/xml

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **API** | **Description** | **Request Body/Parameters** | **Response Body** | **HTTP Response Code** | **Error Codes** | **Request Example** | **Response Example** |
| POST - /api/v1/delivery | Add new delivery | Body: { “order\_id”: <order\_id>, “driver\_name”: <driver\_name>, “predicted\_delivery\_time”: <predicted\_delivery\_time>}  Header: User-context token | {delivery\_id: :string} | 200 | 500 | POST http://52.139.21.27/api/v1/delivery/ | {"delivery\_id":"0fed398b-b5be-4e16-a4c9-bebf9dc42422"} |
| DELETE - /api/v1/delivery/<delivery\_id> | Delete a completed delivery | Params: <delivery\_id>  Header: User-context token | JSON of response from aws | 200 | 500 | DELETE http://52.139.21.27/api/v1/delivery/<delivery\_id> | { ResponseMetadata: { … etc. } } |
| GET-  /api/v1/delivery/<delivery\_id> | Retrieve delivery details | Params: <delivery\_id>  Header: User-context token | JSON of Deliveries entity | 200 | 500 | GET http://52.139.21.27/api/v1/delivery/<delivery\_id> | {Deliveries Object for given <delivery\_id>} |

Version: v1

Service: db

Visibility: Private

Domain: Datastore

Serialized Data/Content-Type: json/xml

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **API** | **Description** | **Request Body/Parameters** | **Response Body** | **HTTP Response Code** | **Error Codes** | **Request Example** | **Response Example** |
| GET - /api/v1/datastore/read | Read an object | Param: object-type, object-key | JSON of aws response | 200 | 500 | GET https://host:5002/api/v1/datastore/read?objtype=user&objkey=123 | { "Count": 1, "Items": [ { "User Name": "John", “User email”: “[john@sfu.ca](mailto:john@sfu.ca)”, “User Phone”: “user\_phone”, "user\_id": "9b235bcb-0d10-415f-ab9f-9ce23da2a119" } ], "ResponseMetadata": { "HTTPHeaders": { "connection": "keep-alive", "content-length": "165", "content-type": "application/x-amz-json-1.0", "date": "Sat, 12 Sep 2020 18:16:15 GMT", "server": "Server", "x-amz-crc32": "196980578", "x-amzn-requestid": "AOGKN903DF66VLU3GEBEEO8DK3VV4KQNSO5AEMVJF66Q9ASUAAJG" }, "HTTPStatusCode": 200, "RequestId": "AOGKN903DF66VLU3GEBEEO8DK3VV4KQNSO5AEMVJF66Q9ASUAAJG", "RetryAttempts": 0 }, "ScannedCount": 1 } |
| POST - /api/v1/datastore/write | Write an object | Body: objtype, object-key(s) | ID of new entity | 200 | 500 | POST https://host:5002/api/v1/datastore/write | { "user\_id": "9b235bcb-0d10-415f-ab9f-9ce23da2a119" } |
| DELETE = /api/v1/datastore/delete | Delete an object | Param: objtype, object-key | JSON of aws response | 200 | 500 | DELETE https://host:5000/api/v1/datastore/delete?objtype=user&objkey=123 | { "ResponseMetadata": { "HTTPHeaders": { "connection": "keep-alive", "content-length": "2", "content-type": "application/x-amz-json-1.0", "date": "Sat, 12 Sep 2020 18:13:04 GMT", "server": "Server", "x-amz-crc32": "2745614147", "x-amzn-requestid": "N7R6LO93FFBDH1A5GRRL55LBS7VV4KQNSO5AEMVJF66Q9ASUAAJG" }, "HTTPStatusCode": 200, "RequestId": "N7R6LO93FFBDH1A5GRRL55LBS7VV4KQNSO5AEMVJF66Q9ASUAAJG", "RetryAttempts": 0 } } |
| PUT - /api/v1/datastore/update | Update an object | Params: objtype, objkey  Body: object-key(s) | JSON of aws response | 200 | 500 | PUT https://host:5002/api/v1/datastore/update?objtype=user&objkey=123 | { "ResponseMetadata": { "HTTPHeaders": { "connection": "keep-alive", "content-length": "2", "content-type": "application/x-amz-json-1.0", "date": "Sat, 12 Sep 2020 18:13:04 GMT", "server": "Server", "x-amz-crc32": "2745614147", "x-amzn-requestid": "N7R6LO93FFBDH1A5GRRL55LBS7VV4KQNSO5AEMVJF66Q9ASUAAJG" }, "HTTPStatusCode": 200, "RequestId": "N7R6LO93FFBDH1A5GRRL55LBS7VV4KQNSO5AEMVJF66Q9ASUAAJG", "RetryAttempts": 0 } |

## Database Schema (DynamoDB)

Table: User

|  |  |  |
| --- | --- | --- |
| **Tag** | **Value** | **Comment** |
| user\_name | string |  |
| user\_email | string |  |
| user\_phone | string |  |
| user\_id | string | Unique id not interpreted by DynamoDB |

Table: Order

|  |  |  |
| --- | --- | --- |
| **Tag** | **Value** | **Comment** |
| food\_name | string |  |
| user\_id | string | Unique id not interpreted by DynamoDB |
| restaurant\_id | string | Unique id not interpreted by DynamoDB |
| order\_id | string | Unique id not interpreted by DynamoDB |

Table: Restaurant

|  |  |  |
| --- | --- | --- |
| **Tag** | **Value** | **Comment** |
| restaurant\_name | string |  |
| food\_name | string |  |
| food\_price | float |  |
| restaurant\_id | string | Unique id not interpreted by DynamoDB |

Table: Payment

|  |  |  |
| --- | --- | --- |
| **Tag** | **Value** | **Comment** |
| payment\_method | string | cash/card |
| discount\_applied | float | Show how much discount applied(percentage) |
| payment\_amount | float | Final payable amount |
| food\_name | string | Name of food ordered |
| user\_id | string | Unique id not interpreted by DynamoDB |
| order\_id | string | Unique id not interpreted by DynamoDB |
| restaurant\_id | string | Unique id not interpreted by DynamoDB |
| payment\_id | string | Unique id not interpreted by DynamoDB |

Table: Deliveries

|  |  |  |
| --- | --- | --- |
| **Tag** | **Value** | **Comment** |
| driver\_name | string |  |
| predicted\_delivery\_time | float | The estimated delivery time for the order; can be added to order table |
| order\_id | string | Unique id not interpreted by DynamoDB |
| delivery\_id | string | Unique id not interpreted by DynamoDB |

# Section 2 - Github Repo Guide

|  |  |
| --- | --- |
| **Path** | **Note** |
| /code | Code folder for the different microservices |
| /code/<Placeholder>  (Placeholder for s1,s2,s3,s4,s5 and db) | Code Folder containing app.py, Dockerfile and other relevant files for the different microservices (s1,s2,s3,s4,s5,db) |
| /code/gatling | Folder containing the code and configuration files used by Gatling for load testing |
| /code/gatling/resources | Csv files used during simulations |
| /code/gatling/simulations/proj756 | Code for defining coverage and load simulations |
| /code/helper | Folder for any additional helper code |
| /IaC | Infrastructure as Code folder containing the different scripts/yaml/makefiles used in this project |
| /IaC/Makefiles | Folder for the different makefiles used (az.mak for azure, k8s.mak for Kubernetes, api.mak for API endpoints) |
| /IaC/cluster | Folder containing the cluster information |
| /IaC/cluster/yamls | Folder for the different yaml files for the different microservices |
| /IaC/cluster/cloudformationdynamodb.json | Cloudformation template for the different DynamoDB tables |
| /IaC/tpl-vars.txt | Configuration file used in the application |
| /docs | Folder containing the final report and any other supporting documentation |
| /readme\_images | Image folder containing the images used in README.md |
| /scripts | Scripts folder for any additional scripts that might be required in this application (scripts for running gatling simulations etc.) |
| /scripts/gatling | Folder containing DockerFile and MakeFile to run gatling simulations. |
| /README.md | README file detailing the problem domain, tools used, system architecture etc. |

# Section 3 - Reflection on Development

In this term project, our team of 5 members are working on developing a simplified food delivery system. The goal of this project is to apply all the distributed systems based technologies that we have learned over the course and integrate them into our project workflow. We are also trying to replicate the scrum methodology as far as possible. Currently, we have a decentralized Git repository where each of the members have their own branches in which we can work independently. We are using the Github issue tracker and project board to assign tasks to the members. Each of the members are taking ownership of their own modules. Additionally, our work is completely transparent and we can monitor each other's progress through the Git commits.

At the start of the project, we have defined the items to be added in the product backlog based on the project requirements. The tasks have been distributed and assigned to the different team members based on their competencies. We have been able to successfully complete the API development and perform 1 round of Gatling simulations as of milestone 2. In addition, since there is no Product Owner/Scrum Master here, the development team i.e. us will take the final call in case of any disputes through a majority vote. We review each other’s work to ensure a seamless workflow without impacting each other’s deliverables.

As a part of the scrum methodology, it is better to have a clear idea regarding what the final goal of the project is at the onset of the development. Currently, we have implemented the different microservices that interact with the different entities in the database and now, we are working on the simulation testing using Gatling. Our next step is to perform load simulation testing and reproduce failures to check how our system behaves in those cases.

# Section 4 - Analysis

In this part, we shall share the results of our analysis after running simulations with the different API’s distributed across the different microservices. We have run two types of simulations: a coverage simulation where we test the workings of the different API calls and a load simulation where we test how the system performs when we add load to the system and its response to failures. We are defining different scenarios based on the simulation type. In the subsequent sections, we shall talk about the different testing approaches for each simulation using Gatling and the results.

4.1 Coverage simulation

In this simulation, we are using Gatling to define custom scenarios in order to test the workings of the different APIs across the different microservices. This simulation is performed from the perspective of a single user, as we are not interested in the load test for this part. We test the different API calls to create (POST) , update (PUT), retrieve (GET) and delete (DELETE) records across the different entities (users/restaurants/orders/payments/deliveries). We have implemented different scenarios for the different types of API calls to check if they are working as intended.

We were able to successfully perform the GET API calls for each of the entities for a single user which indicates that the GET API calls are functional. A typical run output of the simulation after running the Gatling scripts is as follows:-

> RUser 0 (OK=1 KO=0 )

> RRestaurant 0 (OK=1 KO=0 )

> ROrders 0 (OK=1 KO=0 )

> RDiscount 0 (OK=1 KO=0 )

> RBills 0 (OK=1 KO=0 )

> RDelivery 0 (OK=1 KO=0 )

As seen from the status (OK=1 and KO=0), the GET API requests for each of the entities using Gatling was successful.

The results of the Grafana dashboard after running the simulations are as follows:-



Fig 1: Grafana dashboard for GET API calls using Gatling simulations

We can observe that the requests were successful with no errors after letting the simulation run with one user for at least 30 minutes. We also observe that the db service requests have a lower response team compared to the service requests for the other microservices. The next step is to verify the results for the other API calls (PUT, POST and DELETE) in order to complete this coverage simulation.

4.2 Load simulation

In this simulation, we perform a load test on the entire system to verify how the system performs when multiple users are accessing the system in parallel. We design a custom load scenario that integrates the different API calls across the different microservices to replicate a real-world use case scenario. In this scenario, a user who is using the application for the first time needs to sign up (using the ‘Add user’ API). We assume that the restaurants have already been added using the ‘Add Restaurant’ API call and so, for the purpose of this simulation, we will be querying an existing restaurant. After adding the user, the user needs to login (using the ‘Login’ API) before he can. The user then selects some food that he wishes to order and adds them to the cart (using the ‘Add Order’ API). The user can then view the order details using the ‘Get Order’ API call if he wants to. After the order is submitted, the system then generates the bill for that order (using the ‘Add Bill’ API). The user can view the final amount that he needs to pay using the ‘Get Bill’ API. He can also check if any discounts were applied to the final amount using the ‘Show Discount’ API call. Finally, once the order is ready, it is assigned to a delivery person (using the ‘Add deliveries’ API). The user can then check the status of the delivery and the estimated delivery time through this application (using the ‘Get deliveries’ API). This scenario contains an integration of the different API calls across the different microservices from a single user perspective.

We then add more users who perform all these operations on the system in parallel. We also try to simulate failures where one of the pods running one of the microservices is forcefully shut down to see how our system responds to such failures. Depending on the service pod that is shut down, the impact due to the errors can be either big or small. For example, in the above use case, if the Kubernetes pod running the ‘orders’ service fails, the user is unable to submit any new orders and so the rest of the services for ‘payment’, ‘discount’ and ‘deliveries’ will not work as expected (They need an order to be submitted before they can do their respective activities). On the other hand, if the Kubernetes pod running the ‘discount’ service fails, the user is unable to check the discount applied against the order. However, the user can still proceed ahead with the payment and the delivery without the discount. Currently, we are still working on defining the aforementioned load scenario and running the different tests. We shall update this section once we have the results of our testing.