FOODLE

Kaushik Mishra and Monjoy Narayan Choudhury 14/12/2023

Problem Statement and Overview

Foodle is a centralized mess management platform that can be used by institutions to handle all mess-related queries. It solves the issue of depending on traditional mailing lists and the lack of feedback collection facilities from students. It acts as a bridge between students, the food committee (involving admin), and any contractor. The application on the user end allows the user to view Menu, view the paid canteen menu, give feedback for meal service (anonymously if wanted) along with generate service request like room service or special food requirements. The admin side allows to update the menu, add/ update new items to the canteen menu, view feedback left by students, check open tickets for services and resolve them accordingly.

The front end of the application is a React frontend and employs a microservices architecture using spring-boot. The database choice is MongoDB. As a proof of concept it is deployed using docker-compose and the deployment is handled using Ansible. We have not used any cloud services as the number of microservices exceeds the free resources given but have ensured that the code can be transferred to any paid cloud with little to no changes.

Use Case Diagram

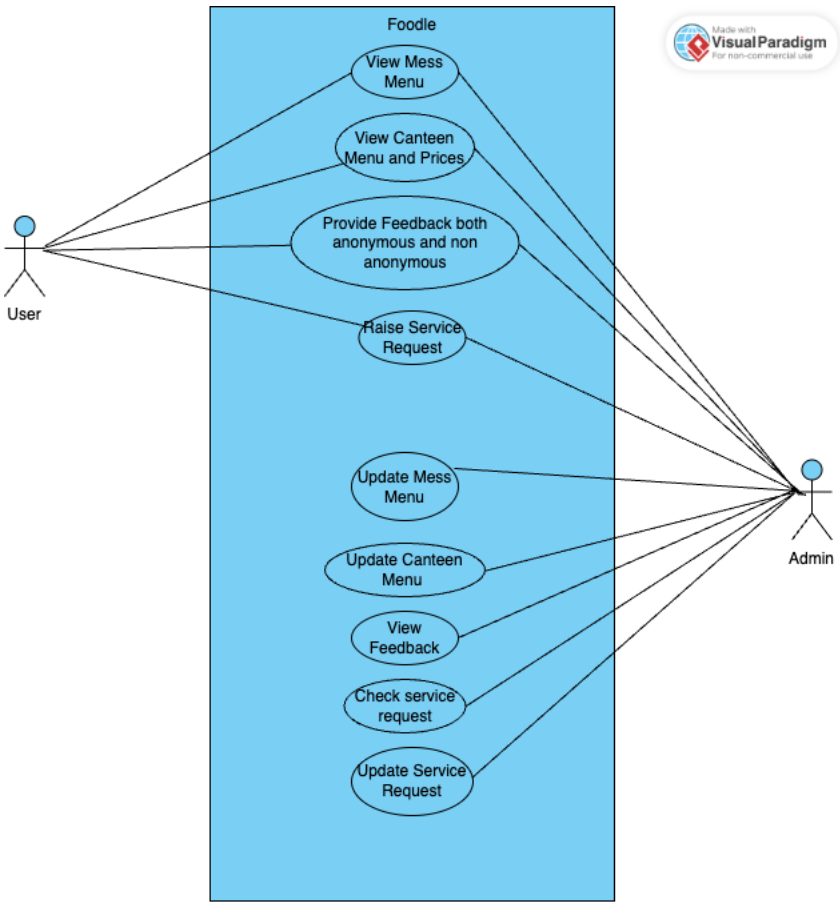

Figure 1: Use Case Diagram

Architecture Diagram

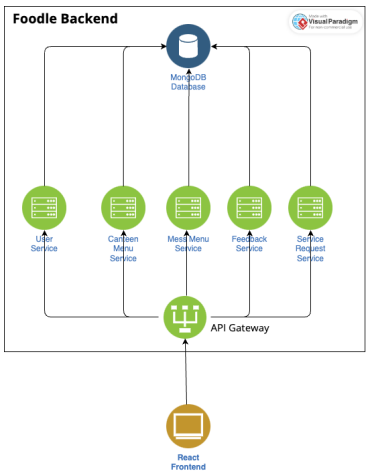

Figure 2: Microservices Architecture of Foodle

We employ a microservice-based architecture for the following reasons:

- Scalability: Microservices allow independent scaling of individual services based on de-

mand. This ensures that resources are allocated efficiently and only to the components that need them, rather than scaling the entire application.

- Flexibility and Agility: Microservices enable independent development, deployment, and scaling of services. This allows development teams to work on and release features independently, reducing dependencies and speeding up the development life cycle.
- Fault Isolation: Isolating services means that a failure in one service does not necessarily affect the entire application. This enhances fault tolerance and makes it easier to identify and address issues without impacting the entire system.
- Technology Diversity: Different services in a microservices architecture can be developed and deployed using different technologies and programming languages. This flexibility allows teams to choose the best tools for the specific requirements of each service.
- Continuous Delivery and Deployment: Microservices enable continuous integration and deployment, allowing for faster and more frequent releases. Teams can deploy changes to specific services without affecting the entire application, reducing the risk associated with large-scale releases.

Repository Links

- GitHub Repository. Refer to corresponding branches for frontend, microservices, and deployment scripts used.
- DockerHub Links:
  - Canteen Service
  - API Gateway
  - Frontend
  - Mess Menu Service
  - Service Request
  - Feedback Service
  - User Service
  - Service Discover/Registry

API Documentation

Base URL: `/api/canteen`

Canteen Service

1. Get All Food Items Endpoint

```
GET /api/canteen
```

Description Retrieve a list of all food items in the canteen.

Response

```
[
  {
    "id": "string",
    "name": "string",
    "description": "string",
    "price": 0.0
  },
  // ... other food items
]
```

2. Get Food Item by ID

Endpoint

```
GET /api/canteen/{id}
```

Parameters

- `id` (path): The unique identifier of the food item.

Description Retrieve details of a specific food item by its ID.

Response

```
{
  "id": "string",
  "name": "string",
  "description": "string",
  "price": 0.0
}
```

3. Add New Food Item

Endpoint

```
POST /api/canteen
```

Request

- Content Type: `application/json`

```
{
  "name": "string",
  "description": "string",
  "price": 0.0
}
```

Description Add a new food item to the canteen menu.

Response

```
{
  "id": "string",
  "name": "string",
  "description": "string",
  "price": 0.0
}
```

4. Delete Food Item

Endpoint

```
DELETE /api/canteen/{id}
```

Parameters

- `id` (path): The unique identifier of the food item to be deleted.

Description Delete a food item from the canteen menu.

Response

- Status Code: `204 No Content`

Feedback Service

Base URL: `/api/feedback` **1. Get All Feedback**

Endpoint

```
GET /api/feedback
```

Description Retrieve a list of all feedback.

Response

```
[
  {
```

```
    "id": "string",
    "name": "string",
    "emailID": "string",
    "FeedbackText": "string",
    "FeedbackImage": "byte[]"
  },
  // ... other feedback items
]
```

2. Get Feedback by ID

Endpoint

```
GET /api/feedback/{id}
```

Parameters

- `id` (path): The unique identifier of the feedback.

Description Retrieve details of specific feedback by its ID.

Response

```
{
  "id": "string",
  "name": "string",
  "emailID": "string",
  "FeedbackText": "string",
  "FeedbackImage": "byte[]"
}
```

3. Add New Feedback

Endpoint

```
POST /api/feedback
```

Request

- Content Type: `application/x-www-form-urlencoded`

```
{
  "name": "string",
  "emailID": "string",
  "FeedbackText": "string",
  "FeedbackImage": "byte[]"
}
```

Description Add new feedback.

Response

```
{
  "id": "string",
  "name": "string",
  "emailID": "string",
  "FeedbackText": "string",
  "FeedbackImage": "byte[]"
}
```

4. Delete Feedback

Endpoint

```
DELETE /api/feedback/{id}
```

Parameters

- `id` (path): The unique identifier of the feedback to be deleted.

Description Delete feedback.

Response

- Status Code: `204 No Content`

Mess Menu Service for daywise data

Base URL: `/api/days` **1. Add Day**

Endpoint

```
POST /api/days
```

Request

- Content Type: `application/json`

```
{
  // ... day properties
}
```

Description Add a new day.

Response

```
{
  "name": "string",
```

```
  "menus": [
    {
      "foodItems": [
        {
          "name": "string",
          "description": "string",
          "foodImage": "byte[]"
        },
        // ... other food items
      ],
      "type": 1
    },
    // ... other menus
  ]
}
```

2. Update Day

Endpoint

```
PUT /api/days/{id}
```

Parameters

- `id` (path): The unique identifier of the day to be updated.

Request

- Content Type: `application/json`

```
{
  // ... updated day properties
}
```

Description Update an existing day.

Response

```
{
  "name": "string",
  "menus": [
    {
      "foodItems": [
        {
          "name": "string",
          "description": "string",
          "foodImage": "byte[]"
        },
```

```
        // ... other food items
      ],
      "type": 1
    },
    // ... other menus
  ]
}
```

3. Get All Days

Endpoint

```
GET /api/days
```

Description Retrieve a list of all days.

Response

```
[
  {
    "name": "string",
    "menus": [
      {
        "foodItems": [
          {
            "name": "string",
            "description": "string",
            "foodImage": "byte[]"
          },
          // ... other food items
        ],
        "type": 1
      },
      // ... other menus
    ]
  },
  // ... other days
]
```

4. Get Day by ID

Endpoint

```
GET /api/days/{id}
```

Parameters

- `id` (path): The unique identifier of the day.

Description Get details of a specific day by its ID.

Response

```
{
  "name": "string",
  "menus": [
    {
      "foodItems": [
        {
          "name": "string",
          "description": "string",
          "foodImage": "byte[]"
        },
        // ... other food items
      ],
      "type": 1
    },
    // ... other menus
  ]
}
```

Service Requests

Base URL: `/api/service-requests`

1. Create Service Request

Endpoint

```
POST /api/service-requests
```

Request

- Content Type: `application/x-www-form-urlencoded`

```
{
  // ... service request properties
}
```

Description Create a new service request.

Response

```
{
  "id": "string",
  // ... service request properties
}
```

2. Get All Service Requests

Endpoint

```
GET /api/service-requests
```

Description Retrieve a list of all service requests.

Response

```
[
  {
    "id": "string",
    // ... service request properties
  },
  // ... other service requests
]
```

3. Get Service Request by ID

Endpoint

```
GET /api/service-requests/{id}
```

Parameters

- `id` (path): The unique identifier of the service request.

Description Get details of a specific service request by its ID.

Response

```
{
  "id": "string",
  // ... service request properties
}
```

4. Update Status

Endpoint

```
PATCH /api/service-requests/{id}
```

Parameters

- `id` (path): The unique identifier of the service request.
- `status` (query): The updated status of the service request.

Description Update the status of a service request.

5. Get Active Service Requests

Endpoint

```
GET /api/service-requests/active
```

Description Retrieve a list of active service requests.

Response

```
[
  {
    "id": "string",
    // ... service request properties
  },
  // ... other active service requests
]
```

User Service

Base URL: `/api/users`

1. Get All Users

Endpoint

```
GET /api/users
```

Description Retrieve a list of all users.

Response

```
[
  {
    "id": "string",
    "emailID": "string",
    "role": "string"
  },
  // ... other users
]
```

2. Get User by Email

Endpoint

```
GET /api/users/{email}
```

Parameters

- `email` (path): The email address of the user.

Description Get details of a specific user by email.

Response

```
{
  "id": "string",
  "emailID": "string",
  "role": "string"
}
```

3. Add User

Endpoint

```
POST /api/users
```

Request

- Content Type: `application/json`

```
{
  "emailID": "string",
  "role": "string"
}
```

Description Add a new user.

Response

```
{
  "id": "string",
  "emailID": "string",
  "role": "string"
}
```

4. Update User Role

Endpoint

```
PUT /api/users/{email}/updateRole
```

Parameters

- `email` (path): The email address of the user.

- `newRole` (query): The new role to be assigned to the user.

Description Update the role of a user.

Response

```
{
  "id": "string",
  "emailID": "string",
  "role": "string"
}
```

User Interface

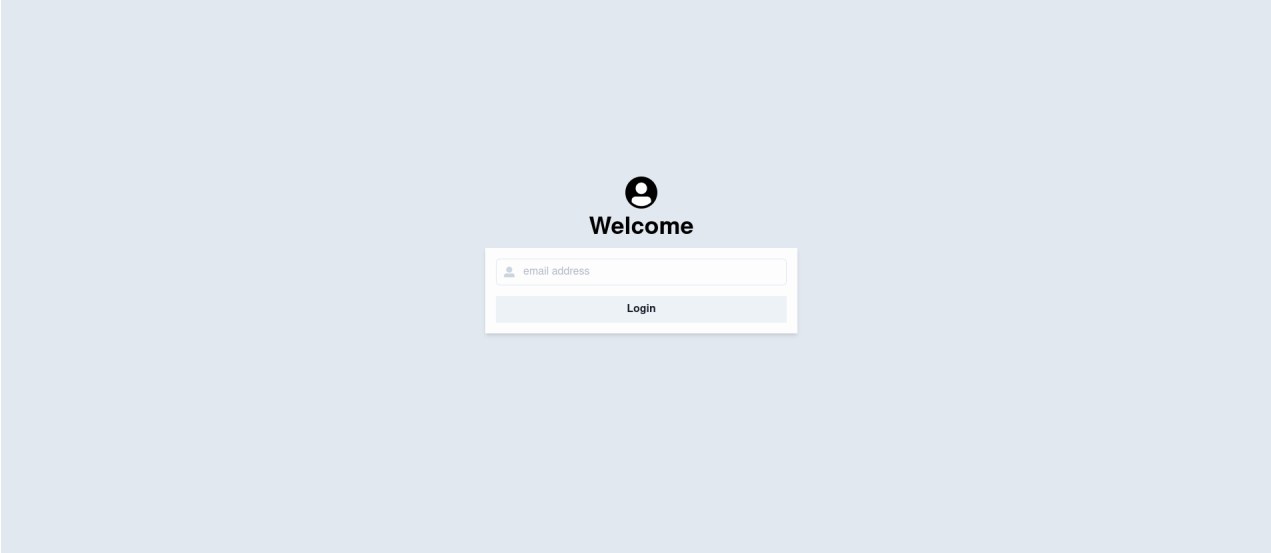

Figure 3: Login Page

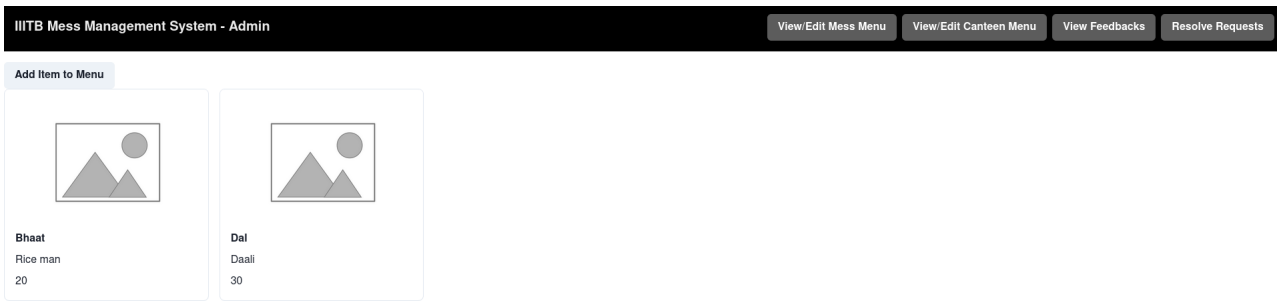

Figure 4: Admin Side Home

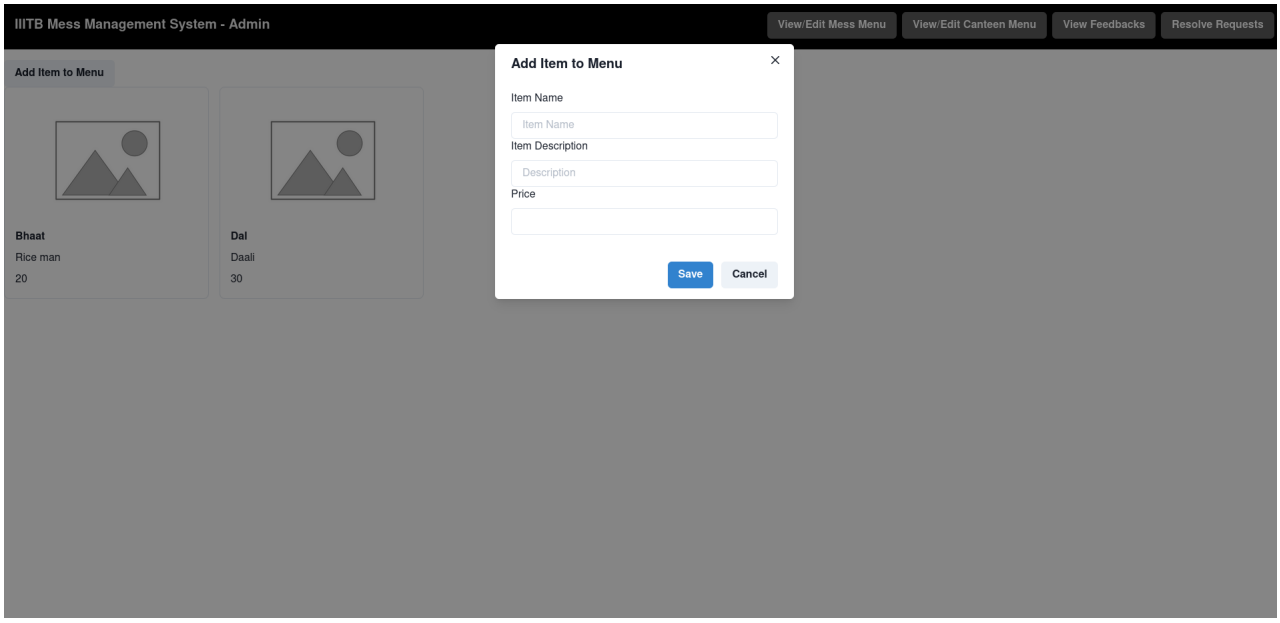

Figure 5: Adding food item

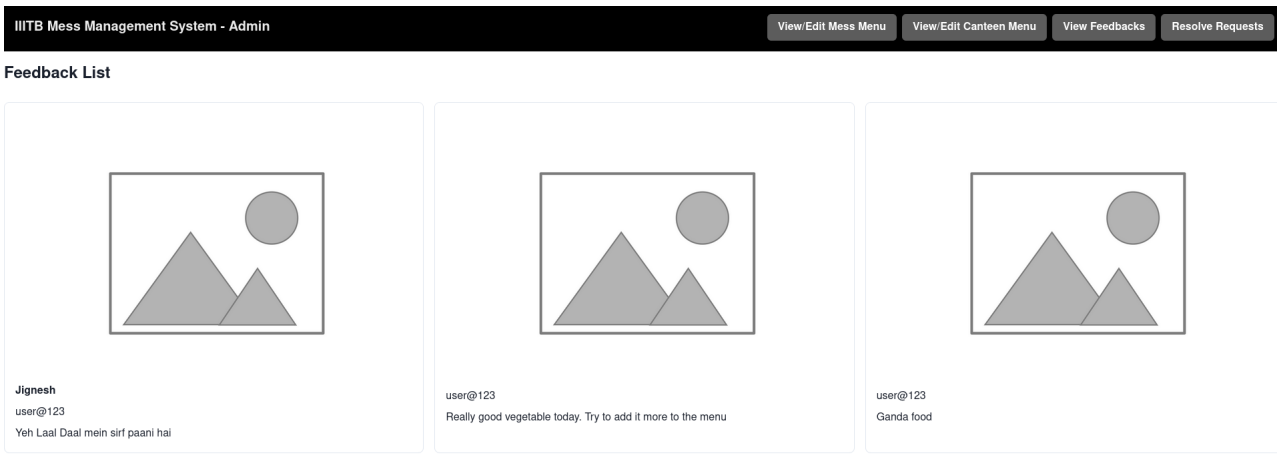

Figure 6: Viewing User Feedbacks

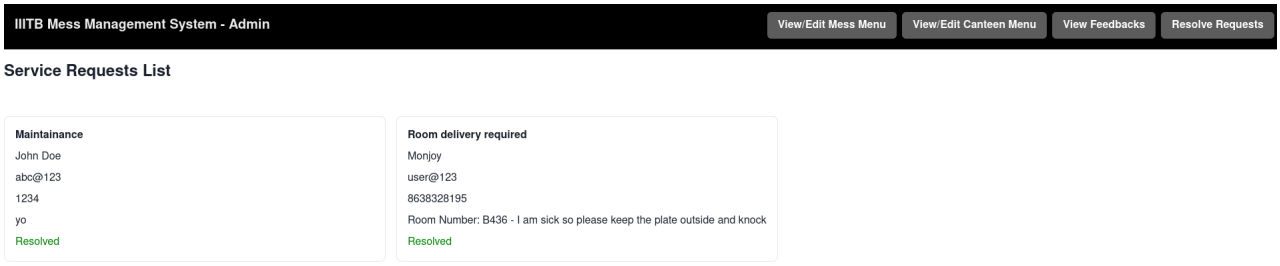

Figure 7: Viewing Service Request and Resolving them

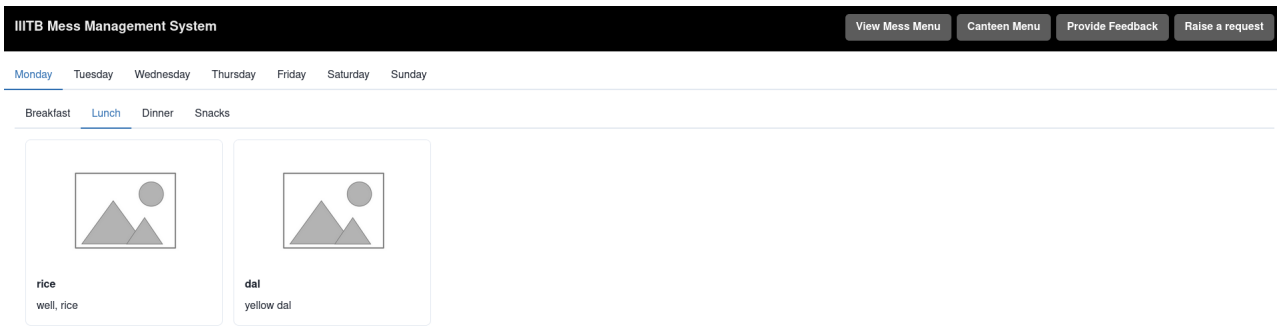

Figure 8: User Login Landing

Figure 9: Viewing Canteen Menu

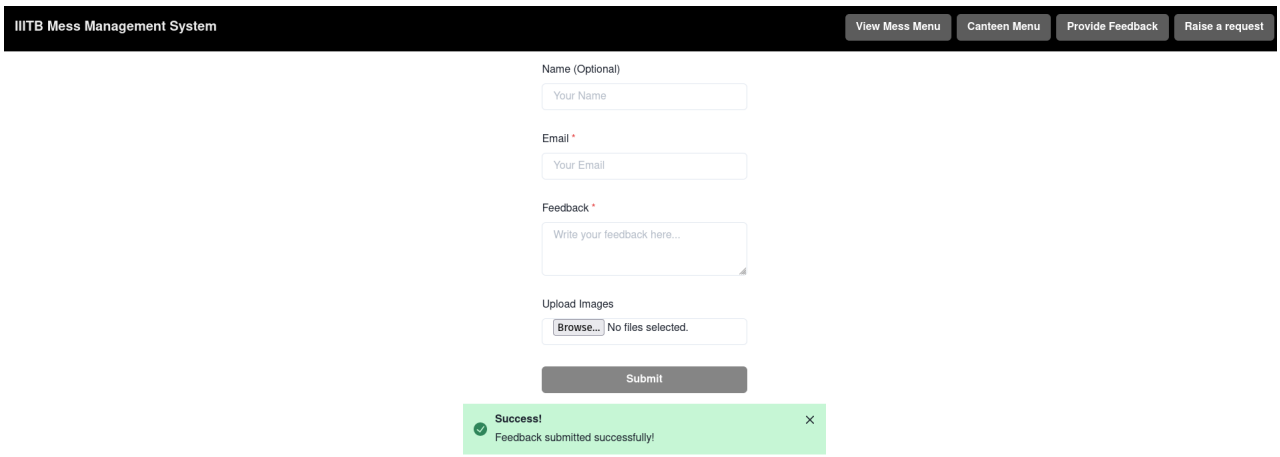

Figure 10: Giving Feedback

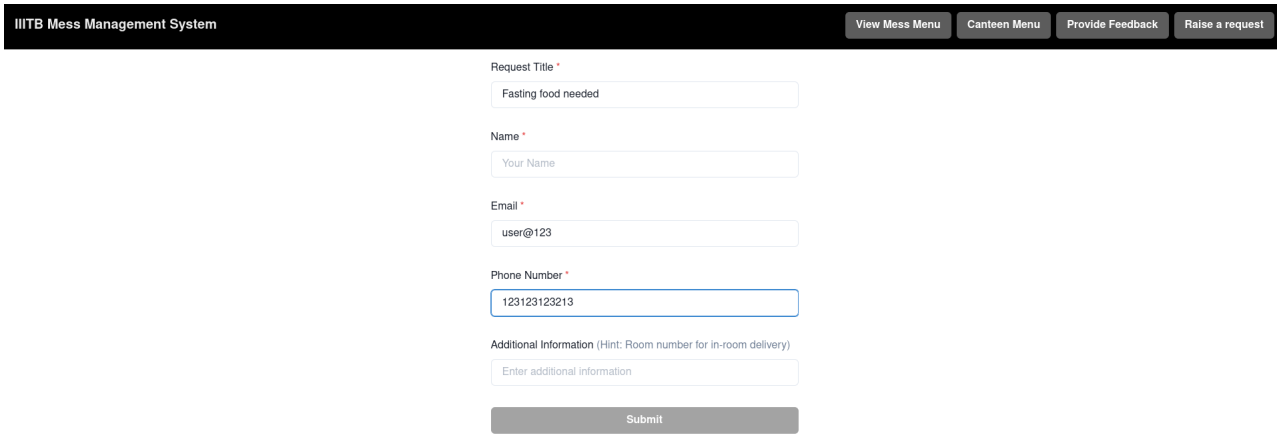

Figure 11: Raising Service Request

Tools used

- git and Github: Used for source code management and version control.
- JUnit: Used for testing the services within various microservices.
- Intellij: IDE suitable for application development in Java with plugins that ease the use of build tools like maven, JUnit, etc.

- GitHub Actions and Jenkins: Used for continuous integration and deployment. Automates the pipeline for testing building and deploying the product. GitHub actions are used for the Continuous Integration and Delivery part while Jenkins is used as a trigger script.
- Docker Compose for handling containers and orchestration.
- Webhook: GitHub webhook is used to trigger a build in Jenkins whenever there is a push to the remote GitHub repository.
- Postman for API testing.
- ngrok: Used to forward requests received at a public IP address to the localhost.
- Maven: Used to automate the build and test processes.
- Docker and Dockerhub: Used to containerize the application and store the container image remotely.
- Ansible: Used to configure hosts easily.
- ELK Stack: Tool used to analyze logs generated by the application for monitoring purposes.

Steps

Setup Project on IDE

1. Install Intellij and create a new project. Right-click on the project and select Add Framework Support. Add Maven. This creates a directory structure where the application code is to be written in /src/main/java/ and unit tests in /src/test/java/.
2. A file called pom.xml is also created. Change the groupId and artifactId appropriately. JUnit is used for testing and log4j is used for logging during runtime. So, add these as dependencies and change the build block in pom.xml.
3. This is repeated for all microservices

Source Code Management

1. Create an account and a new repository at https://github.com/. Click on Code and copy the repository URL.
2. Run the following commands in your terminal:

```
1    git init
2    git add .
3    git commit -m "initial commit"
4    git remote add origin <REPOSITORY_URL>
```

```
5               git remote add upstream <REPOSITORY_URL>
6               git push origin main
```

Testing

Build

Use Maven to build and test the application. In IntelliJ in the Maven tab, select Execute Maven Goal and select the following (the commands can also be run on the terminal).

```
1 mvn compiler:compile
2 mvn compiler:testCompile
```

or

```
1 mvn clean package
```

This removes the target directory (if it exists) and creates the target directory again with the output files and the application is packaged with its dependencies into a *.jar* file. You can run the microservice using this jar file as follows:

```
1 java -jar <jar name>.jar
```

Containerising

- Install docker. Refer link
- Give the non-root user permission to run docker commands. Refer **??**link
- In the root directory of each of the microservice folders, create a Dockerfile as shown:

Figure 12: Dockerfile for microservice

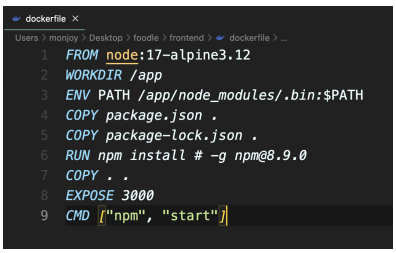

Figure 13: Dockerfile for frontend

- Dockerhub is used to store the docker images remotely so that it can be run on any system.

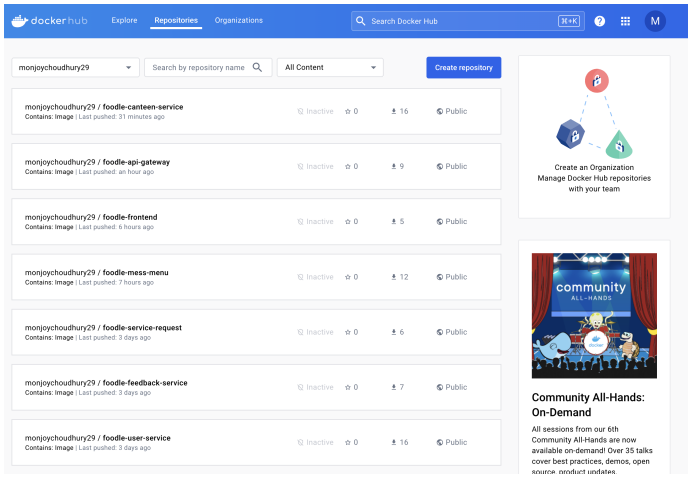

Figure 14: DockerHub

Running Multi-Container Setup - Docker Compose

Docker Compose is a tool that allows you to define and run multi-container Docker applications. It uses a YAML file to describe the services, networks, and volumes needed for your application and can start and stop all containers with a single command. This is useful for managing complex applications with multiple components, and for deploying applications consistently across different environments. We have 2 yml files to first set up the Mongo db database using a container volume for persistence storage while the other docker-compose file handles deploying of microservices. One extra thing to note is that we must keep all the docker containers on the same virtual network provided by docker so that the microservices can communicate with each other using Eureka Client and Server and the API Gateway Microservice.

Listing 1: docker-compose.yml

```
1 version: "3.6"
2 services:
3   frontend:
4     image: monjoychoudhury29/foodle-frontend:latest
5     ports:
6       - "3000:3000"
7     pull_policy: always
```

```
    networks:
      - my-network
    container_name: frontend
  discovery-service:
    image: monjoychoudhury29/foodle-service-discovery:latest
    ports:
      - "8761:8761"
    container_name: discovery-service
    pull_policy: always
    networks:
      - my-network

  api-gateway:
    image: monjoychoudhury29/foodle-api-gateway:latest
    ports:
      - "9191:9191"
    depends_on:
      - discovery-service
    container_name: api-gateway
    pull_policy: always
    networks:
      - my-network

  canteen-service:
    image: monjoychoudhury29/foodle-canteen-service:latest
    ports:
      - "8084:8084"
    depends_on:
      - discovery-service
    container_name: canteen-service
    pull_policy: always
    networks:
      - my-network
    volumes:
      - ./logs:/logs

  feedback-service:
    image: monjoychoudhury29/foodle-feedback-service:latest
    ports:
      - "8082:8082"
    depends_on:
      - discovery-service
    container_name: feedback-service
    pull_policy: always
    networks:
      - my-network

```

```
  mess-menu-service:
    image: monjoychoudhury29/foodle-mess-menu:latest
    ports:
      - "8085:8085"
    depends_on:
      - discovery-service
    pull_policy: always
    container_name: mess-menu-service
    networks:
      - my-network

  user-service:
    image: monjoychoudhury29/foodle-user-service:latest
    ports:
      - "8086:8086"
    depends_on:
      - discovery-service
    pull_policy: always
    container_name: user-service
    networks:
      - my-network

  service-request:
    image: monjoychoudhury29/foodle-service-request:latest
    ports:
      - "8083:8083"
    depends_on:
      - discovery-service
    pull_policy: always
    container_name: service-request
    networks:
      - my-network

networks:
  my-network:
    driver: bridge
```

Listing 2: database-docker-compose.yml

```
version: "3.6"
services:
  mongodb:
    image: mongo
    container_name: mongodb-container
    ports:
      - 27017:27017
```

```
    volumes:
      - mongo:/data
    environment:
      - MONGO_INITDB_ROOT_USERNAME=monjoy
      - MONGO_INITDB_ROOT_PASSWORD=monjoy
    networks:
      - my-network
  mongo-express:
    image: mongo-express
    container_name: mongo_express
    ports:
      - 8081:8081
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=monjoy
      - ME_CONFIG_MONGODB_ADMINPASSWORD=monjoy
      - ME_CONFIG_MONGODB_SERVER=mongodb-container
    networks:
      - my-network
volumes:
  mongo: {}
networks:
  my-network:
    driver: bridge
```

Some notable features of both the files are:

1. Service that denotes a task to be run. Here we put microservice specific container information
2. networks: define the virtual network on which the microservices are running inside the containers in the docker network.
3. environment: Used to define environment variables especially in case of database handling.
4. ports: useful for port forwarding from the docker container to the machine on which the container is running. Port forwarding allows us to check into the running services from our local machine outside the container.
5. pull_policy: defines when an image from docker hub should be checked in and pulled. Keeping it always ensures that docker composes checks for any changes on docker hub and pull and rebuild an image if that's the case.
6. depends_on: is useful especially when you are deploying services where one service needs to be initialized before the other. However, there is a small caveat that only the creation step is followed according to dependencies, the containers don't wait for each

other to start and rather happen continuously. This is where docker compose lapses a bit and some better-complicated orchestration tool like Kubernetes might help.

Deployment using Ansible

We will use Ansible for local deployment. Ansible is a suite of software tools that enables infrastructure as code.

- In Ansible, managed hosts or servers which are controlled by the Ansible control node are defined in a host inventory file. The Ansible inventory file defines the hosts and groups of hosts upon which commands, modules, and tasks in a playbook operate.
- Create an inventory file where we explicitly define our localhost in the inventory file for local deployment.

Figure 15: Inventory File

- Ansible Playbooks offer a repeatable, re-usable, simple configuration management.
- Playbooks consist of one or more plays run in a particular order. A play is an ordered set of tasks run against hosts chosen from your inventory. Plays define the work to be done. Each play contains a set of hosts to configure, and a list of tasks to be executed.

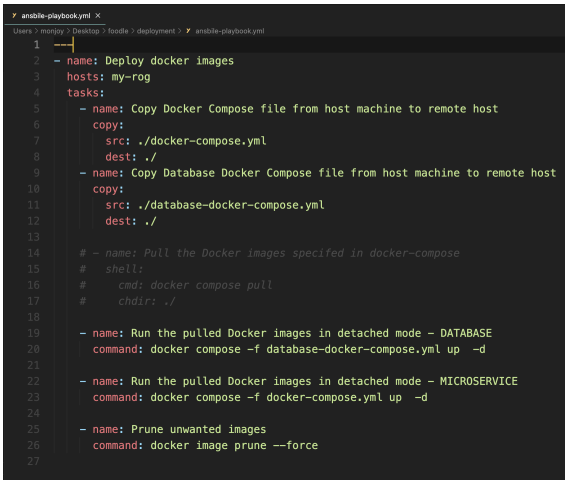

Figure 16: Playbook File

It performs the following steps:

1. Copy Docker Compose file to Remote Host
2. Copy Database Docker Compose file to Remote Host
3. Run Docker Compose for Database in Detached Mode followed by Docker Compose for microservices in detached mode.

4. Prune unwanted images.

Along with script setup, one must also setup SSH and share the necessary keys to connect both the machine (control server and the worker node) together. Once done ansible makes it possible to deploy on any worker node using a control machine anywhere in the world.

Continuous Integration and Delivery Using GitHub Actions

GitHub Actions is a powerful CI/CD (Continuous Integration/Continuous Deployment) platform that can be utilized for a judge platform in the following ways:

1. By configuring GitHub Actions workflows, you can automate the build, test, and deployment processes for the judge platform's codebase.
2. Actions enable you to define custom workflows triggered by events like code pushes or pull requests, ensuring automatic code validation and testing.
3. Integration with GitHub's vast ecosystem allows you to leverage actions from the marketplace to simplify tasks like code linting, unit testing, and containerization.
4. GitHub Actions provides deployment capabilities, enabling you to automatically deploy new versions of the judge platform to target environments such as staging or production.

In our use case, we defined our workflow in the *.github/workflows/main.yml* file. An example of such deployment can be seen below.

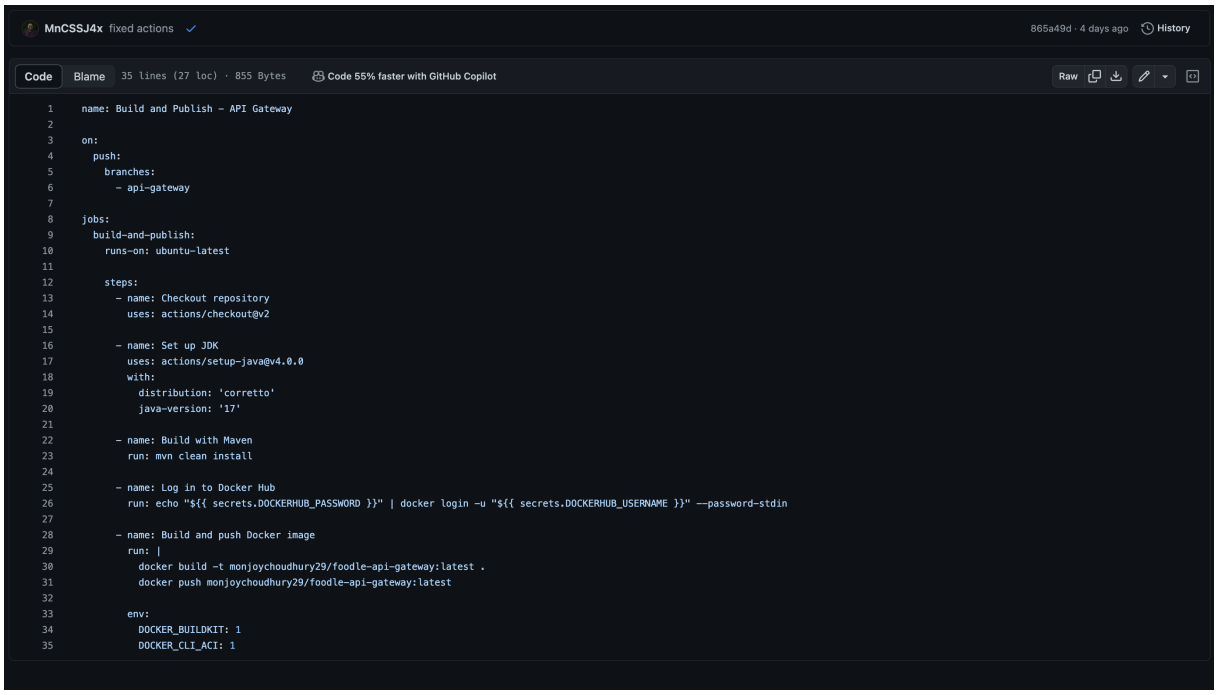

Figure 17: CI/CD pipeline for a microservice

The explanation of the steps in the file is as follows

1. It checks out the GitHub repo for any change. Interstingly as per the lines above, GitHub triggers the checkout internally when it senses a push on a certain branch. In the example above it is the api-gateway branch/

2. In its own Java Virtual Environment on the cloud, GitHub sets up the project and performs the necessary build steps defined in the run block

3. Once done GitHub creates a docker image of the project and pushes it to DockerHub using the secretes credential that we can configure normally on GitHub. This is way simpler to create and manage here in the case of Actions in comparison to Jenkins.

4. Once the push to DockerHub is done GitHub runs additional steps to clean all temporary files generated.

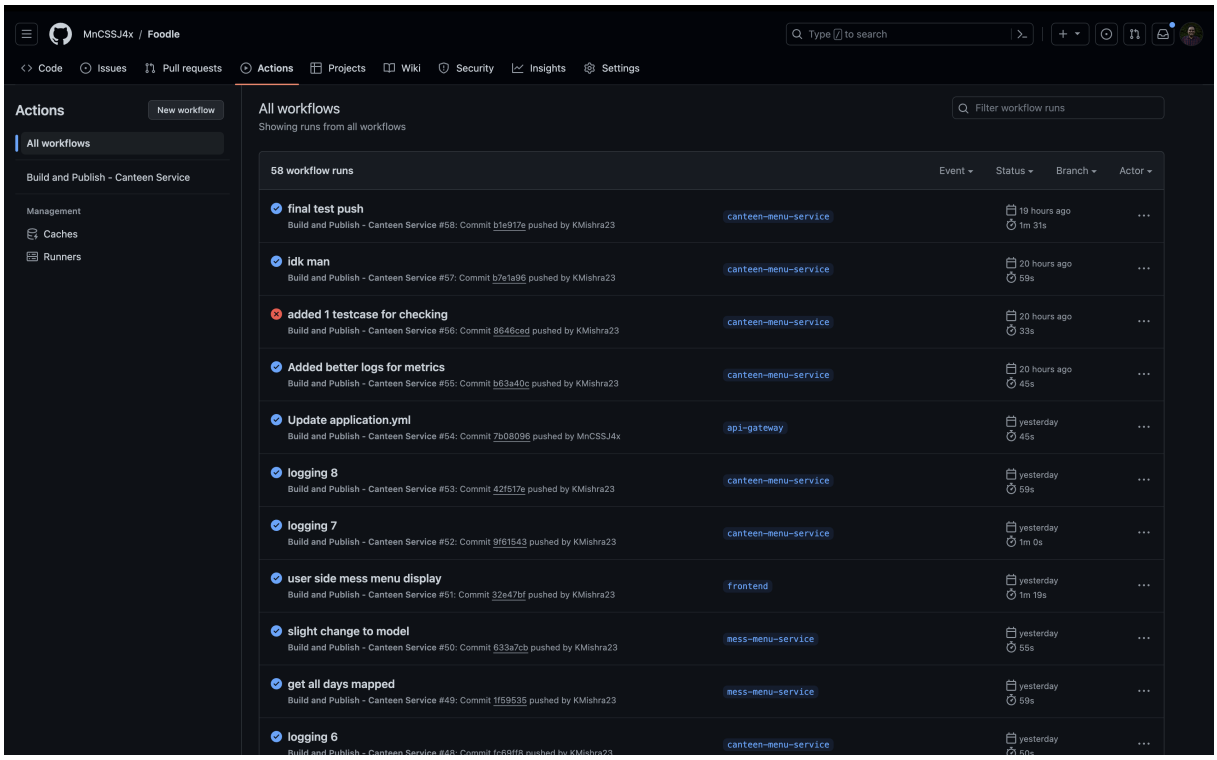

Figure 18: GitHub Actions for project

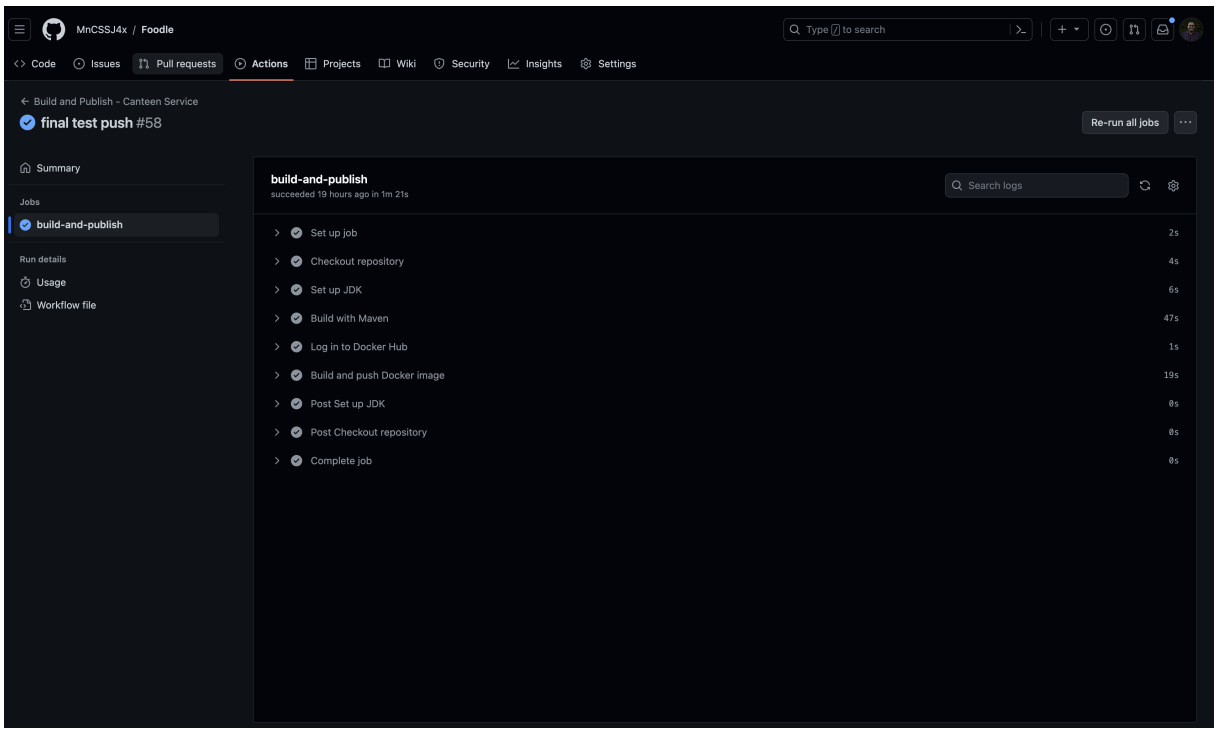

Figure 19: Detailed look on pipeline

Triggering final Deployment Using Jenkins

GitHub actions run the code brought in from source control on their VM which could be hosted anywhere in the world. However, we intend to show a working deployment on local infrastructure as the resources provided by free credits of Azure and AWS are not sufficient to host the number of microservices that we have created. As the control server for Ansible is located locally with the worker node we need an automated pipeline to pull the code from Github on webhook triggers and run the Ansible script. For this, we use Jenkins setup with GitHub Webhook and run a one-stage simple pipeline for running an ansible script.

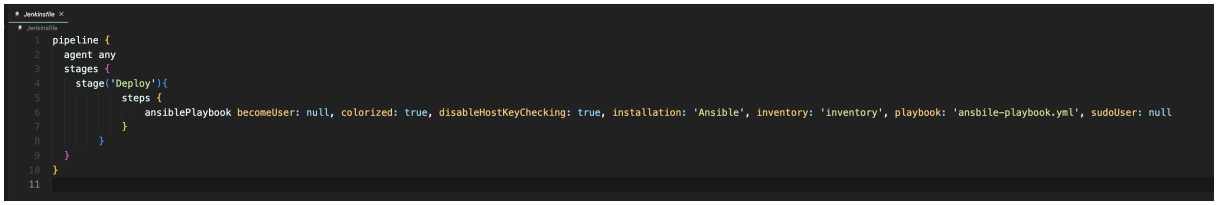

Figure 20: Pipeline Script used

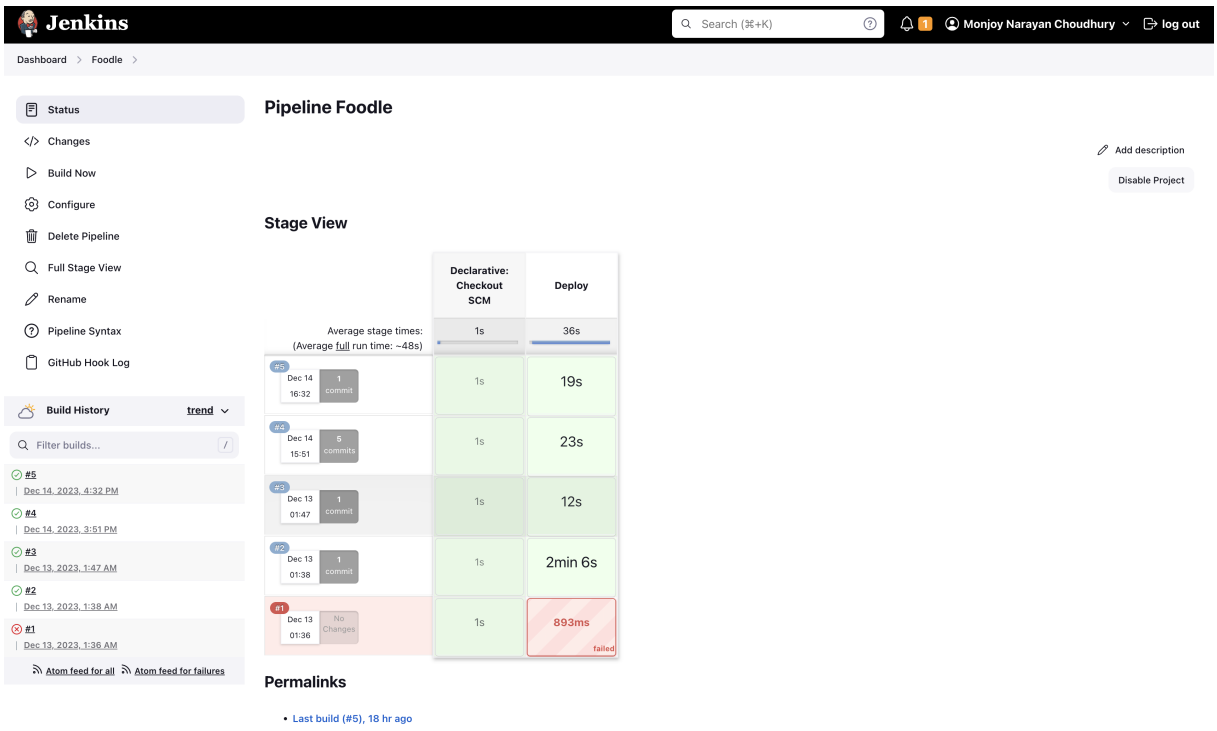

Figure 21: Pipeline for final deployment

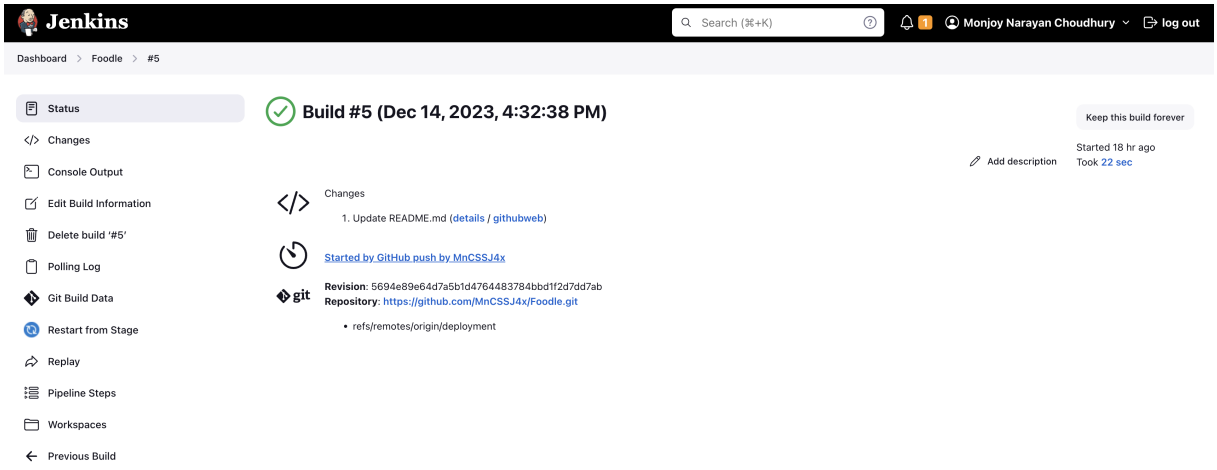

Figure 22: Webhook triggered pipeline

Post Deployment

Final Output on Local Environment

```
(base) monjoynarayanchoudhury@pop-os:~$ docker ps
CONTAINER ID   IMAGE                                              COMMAND                  CREATED        STATUS          PORTS                                                  NAMES
beed43c29974   monjoychoudhury29/foodle-canteen-service:latest    "java -jar canteense…"   19 hours ago   Up 8 seconds    0.0.0.0:8084->8084/tcp, :::8084->8084/tcp              canteen-service
65a90d363de4   monjoychoudhury29/foodle-api-gateway:latest        "java -jar api-gatew…"   22 hours ago   Up 8 seconds    0.0.0.0:9191->9191/tcp, :::9191->9191/tcp              api-gateway
b2ed3f836b4d   monjoychoudhury29/foodle-frontend:latest           "docker-entrypoint.s…"   24 hours ago   Up 9 seconds    0.0.0.0:3000->3000/tcp, :::3000->3000/tcp              frontend
952a044f2ac3   monjoychoudhury29/foodle-mess-menu:latest          "java -jar mess-menu…"   24 hours ago   Up 8 seconds    0.0.0.0:8085->8085/tcp, :::8085->8085/tcp              mess-menu-service
6a6785fde31a   monjoychoudhury29/foodle-feedback-service:latest   "java -jar feedback.…"   3 days ago     Up 9 seconds    0.0.0.0:8082->8082/tcp, :::8082->8082/tcp              feedback-service
99c21a0effe3   monjoychoudhury29/foodle-service-request:latest    "java -jar service-r…"   3 days ago     Up 8 seconds    0.0.0.0:8083->8083/tcp, :::8083->8083/tcp              service-request
65ba96f7bb97   monjoychoudhury29/foodle-user-service:latest       "java -jar userservi…"   3 days ago     Up 8 seconds    0.0.0.0:8086->8086/tcp, :::8086->8086/tcp              user-service
912f66f3f55a   monjoychoudhury29/foodle-service-discovery:latest  "java -jar service-d…"   3 days ago     Up 9 seconds    0.0.0.0:8761->8761/tcp, :::8761->8761/tcp              discovery-service
9f1d1642e7df   mongo                                              "docker-entrypoint.s…"   3 days ago     Up 14 seconds   0.0.0.0:27017->27017/tcp, :::27017->27017/tcp          mongodb-container
8286c25e1ce8   mongo-express                                      "/sbin/tini -- /dock…"   3 days ago     Up 14 seconds   0.0.0.0:8081->8081/tcp, :::8081->8081/tcp              mongo_express
```

Figure 23: Running containers on worker node

Testing

Testing is essential for any software's lifecycle. Making sure that components are working after every incremental change in a devops cycle is absolutely necessary. As this project was not as big, testing here is limited and is only performed on one of the microservices due to time constraints. However, these tests can be easily replicated over the other microservices. Testing is performed for the canteen menu service. It tests the getters and setters of the service as well as emulates a working backend repository by posting to a virtual emulated repository and then evaluating the get request results with the expected results to ensure proper flow of the CRUD operations. We used the JUnit testing framework along with the Mockitto framework for mocking database behavior. Screenshot below showcase the testing results.

```
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.281 s -- in com.eic.canteenservice.CanteenserviceApplicationTests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jar:3.3.0:jar (default-jar) @ canteenservice ---
[INFO] Building jar: /Users/monjoy/Desktop/foodle/canteenservice/target/canteenservice-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot:3.2.0:repackage (repackage) @ canteenservice ---
[INFO] Replacing main artifact /Users/monjoy/Desktop/foodle/canteenservice/target/canteenservice-0.0.1-SNAPSHOT.jar with repackaged archive, adding nested dependencies in BOOT-INF/.
[INFO] The original artifact has been renamed to /Users/monjoy/Desktop/foodle/canteenservice/target/canteenservice-0.0.1-SNAPSHOT.jar.original
[INFO]
[INFO] --- install:3.1.1:install (default-install) @ canteenservice ---
[INFO] Installing /Users/monjoy/Desktop/foodle/canteenservice/pom.xml to /Users/monjoy/.m2/repository/com/eic/canteenservice/0.0.1-SNAPSHOT/canteenservice-0.0.1-SNAPSHOT.pom
[INFO] Installing /Users/monjoy/Desktop/foodle/canteenservice/target/canteenservice-0.0.1-SNAPSHOT.jar to /Users/monjoy/.m2/repository/com/eic/canteenservice/0.0.1-SNAPSHOT/canteenservice-0.0.1-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.598 s
[INFO] Finished at: 2023-12-15T21:02:14+05:30
[INFO] ------------------------------------------------------------------------
```

Figure 24: Test case successes output on maven

Logging and Monitoring

Once deployed Monitoring is critical for maintaining the health and security of the deployment as well as the deployment infrastructure, as it allows us to quickly detect and respond to potential problems before they become critical. However, to make this possible a developer must use appropriate logging mechanisms in various services they are handling to indicate to users the stage, possible exceptions, and possible errors that occur.

Monitoring using log4j2

Logging was performed using Log4j2 to log CRUD actions being performed on the server. It throws out warnings for misses and logs success for hits. This is all generated with the existing logs when running a Maven project. Screenshots below show some log files generated.

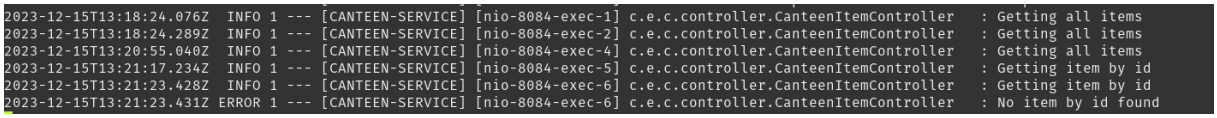

Figure 25: Log outputs for get operations

Figure 26: Log outputs for post operations

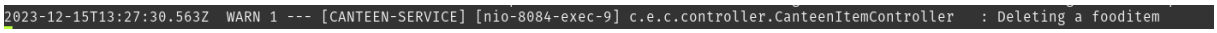

Figure 27: Log outputs for delete operations

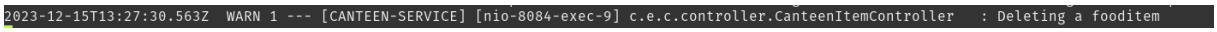

Figure 28: Log outputs for delete operations

Log Analysis and Visualization using the ELK stack

The Elastic stack consists of Elasticsearch, Logstash, and Kibana helps us to achieve the task of parsing and analyzing the logs to get meaningful info out of logs generated by the services and frontend. Elasticsearch is a search and analytics engine. The data processing is handled by Logstash while the users of Elasticsearch may visualize the data using the visual analytics platform Kibana. To get started Login to https://www.elastic.co/ and create a deployment. We can then upload the log file and perform necessary filtering using GROK patterns and visualise this data using a Kibana dashboard.

Log files generated can't be directly used and need to pre-processed before being uploaded into ELK. We used a simple Python script that cleans up the log file, making it usable for ELK directly. The log files before and after processing it are shown in the screenshot below.

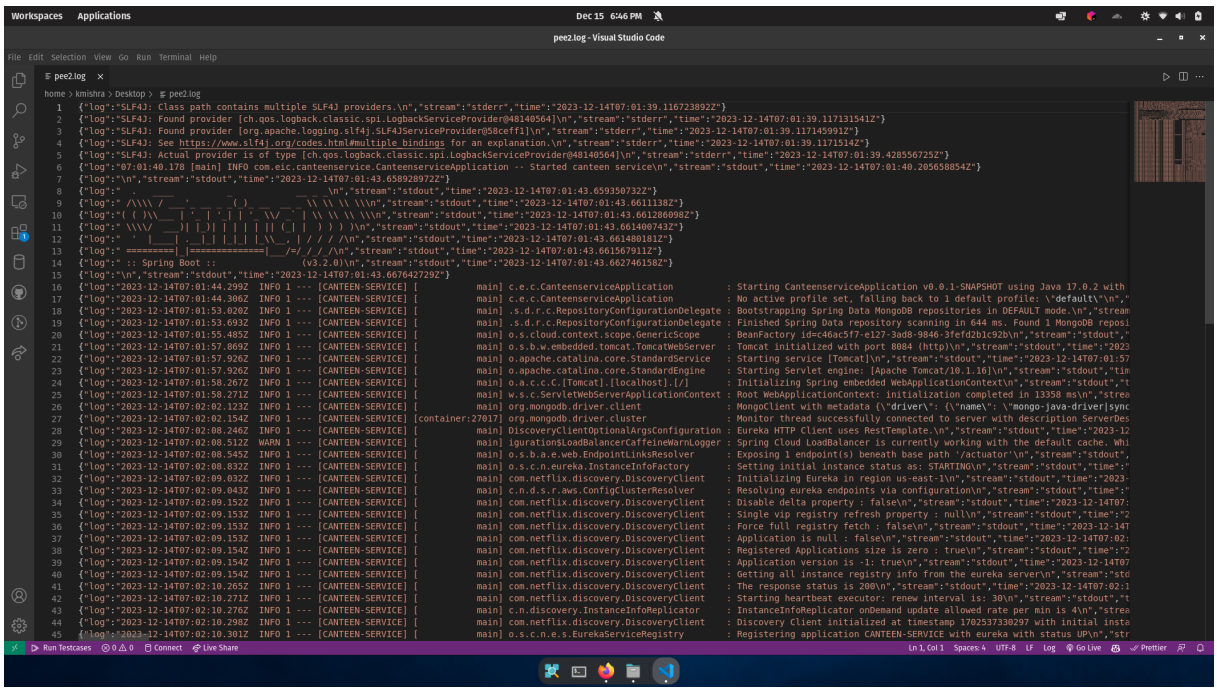

Figure 29: Log File before processing

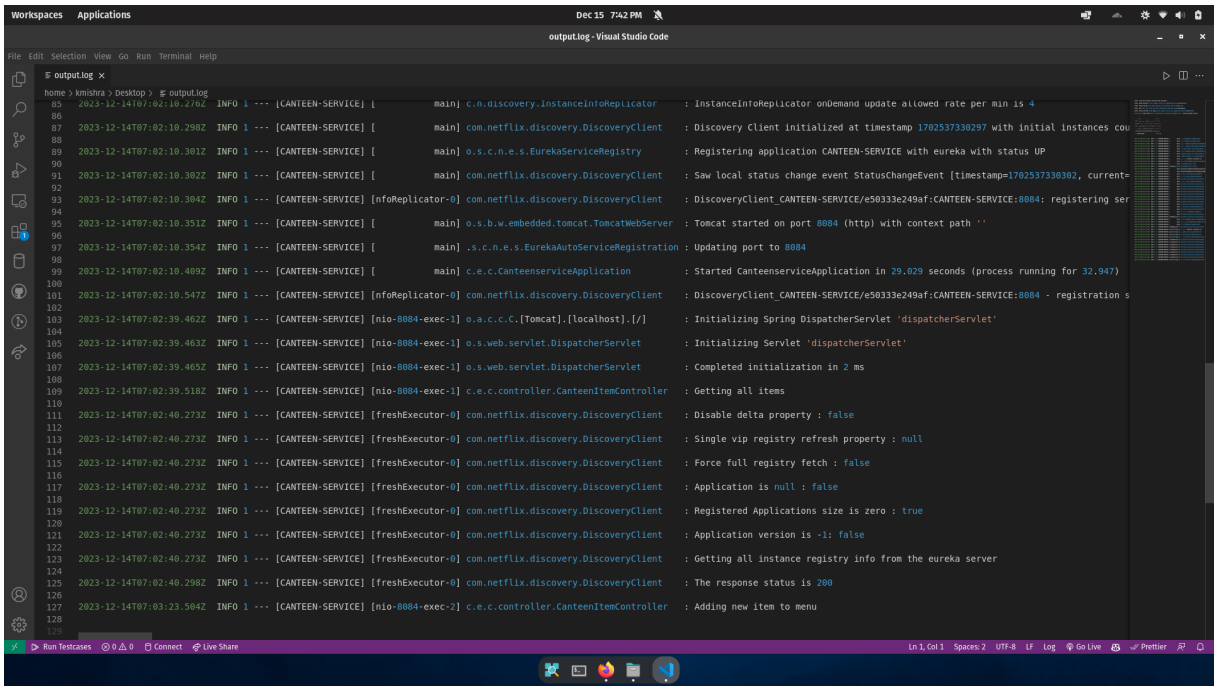

Figure 30: Log file after processing

Then, this processed log file is uploaded to kibana. Kibana automatically creates a suitable GROK pattern for the uploaded log file. We create an index name for it and we can then start visualizing it. Kibana gives us a multitude of options to visualize this log file. Kibana starts by organizing your data using aggregations. Think of it like grouping things together: counting events, calculating averages, or finding unique values. This transformed data becomes the

"building blocks" for visualizations. Kibana then applies its analytical prowess to uncover hidden patterns and trends within our data. Some of these are shown below.

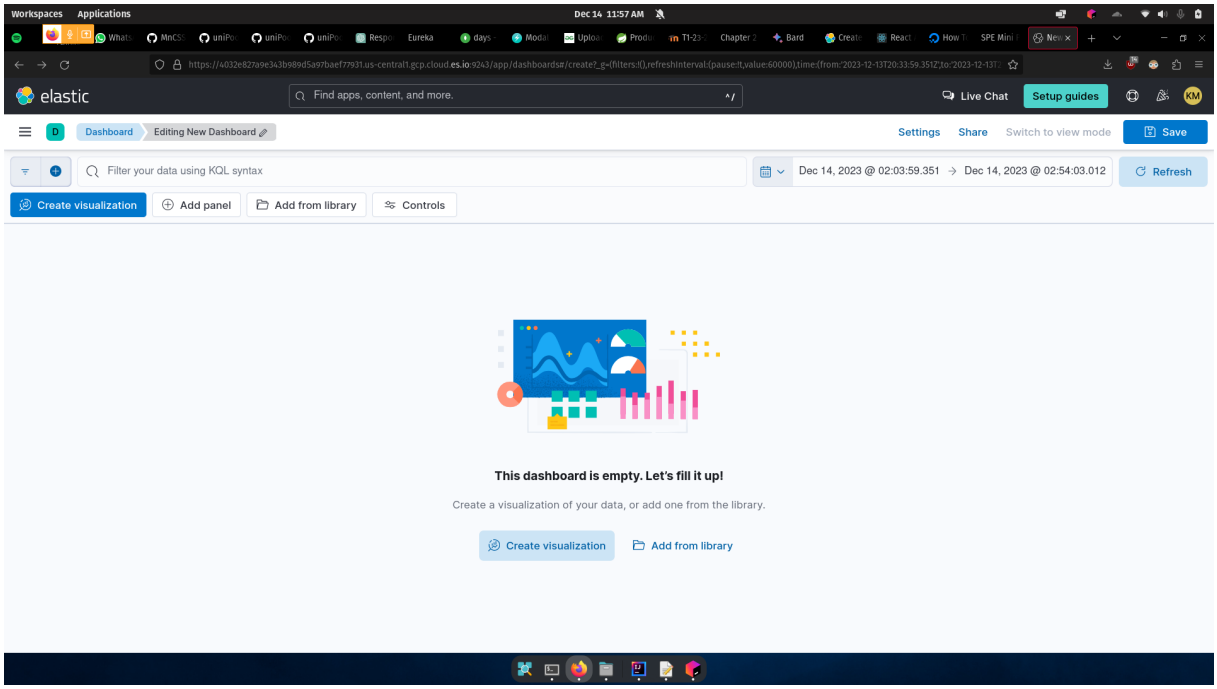

Figure 31: Elastic Dashboard

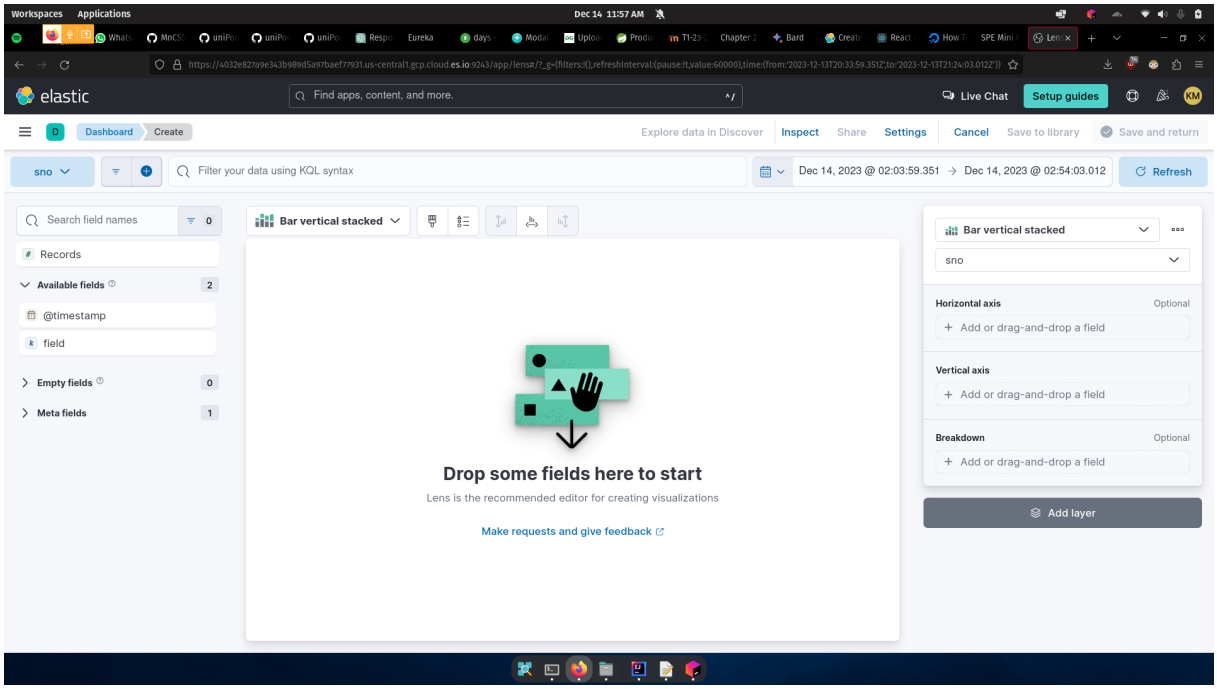

Figure 32: Kibana Dashboard

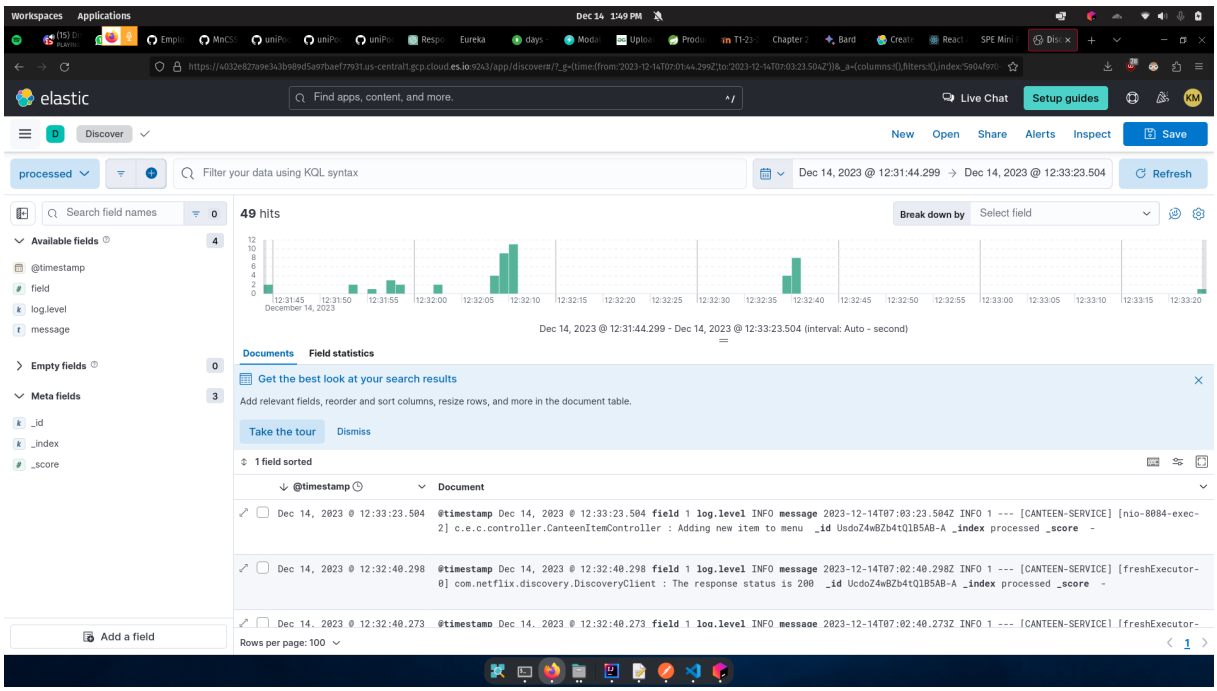

Figure 33: Indexing results

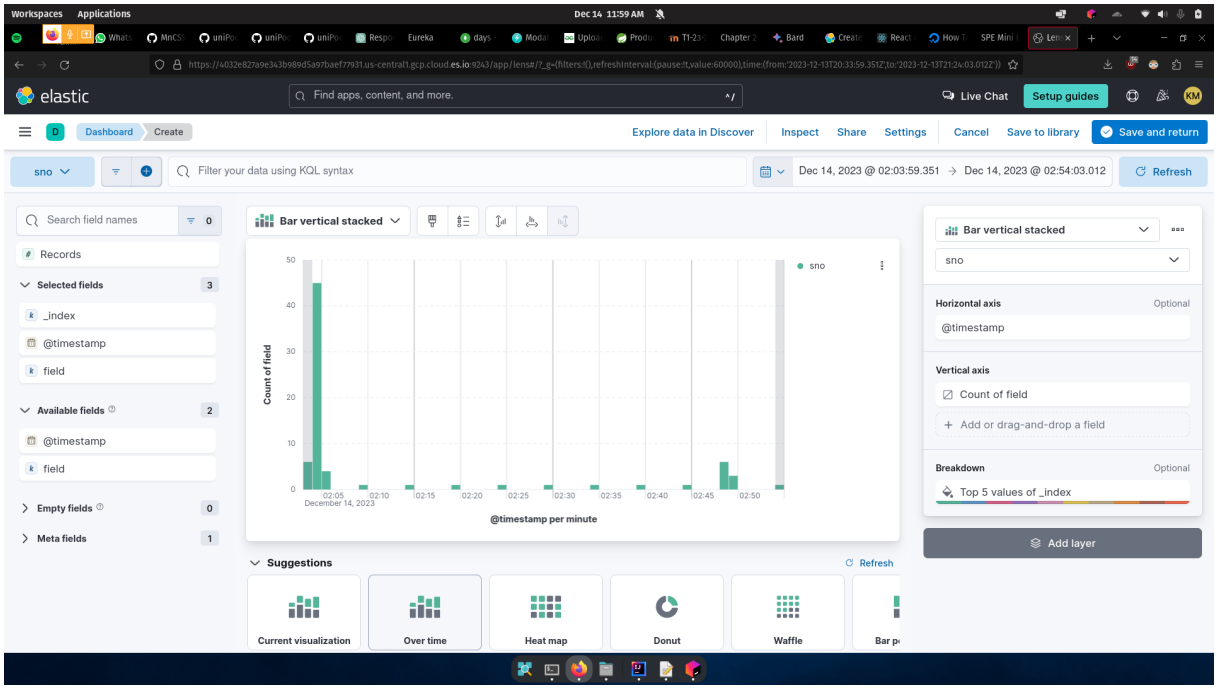

Figure 34: Bar Graph visualization

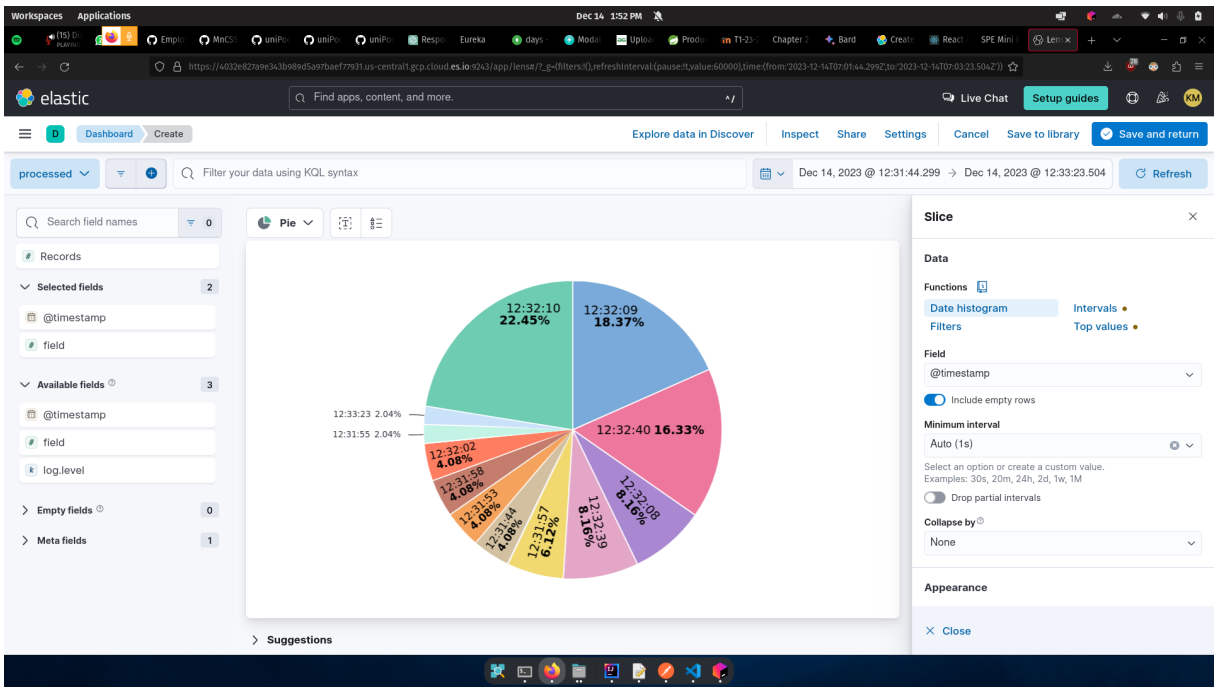

Figure 35: Pie chart showing log outputs per timestamp

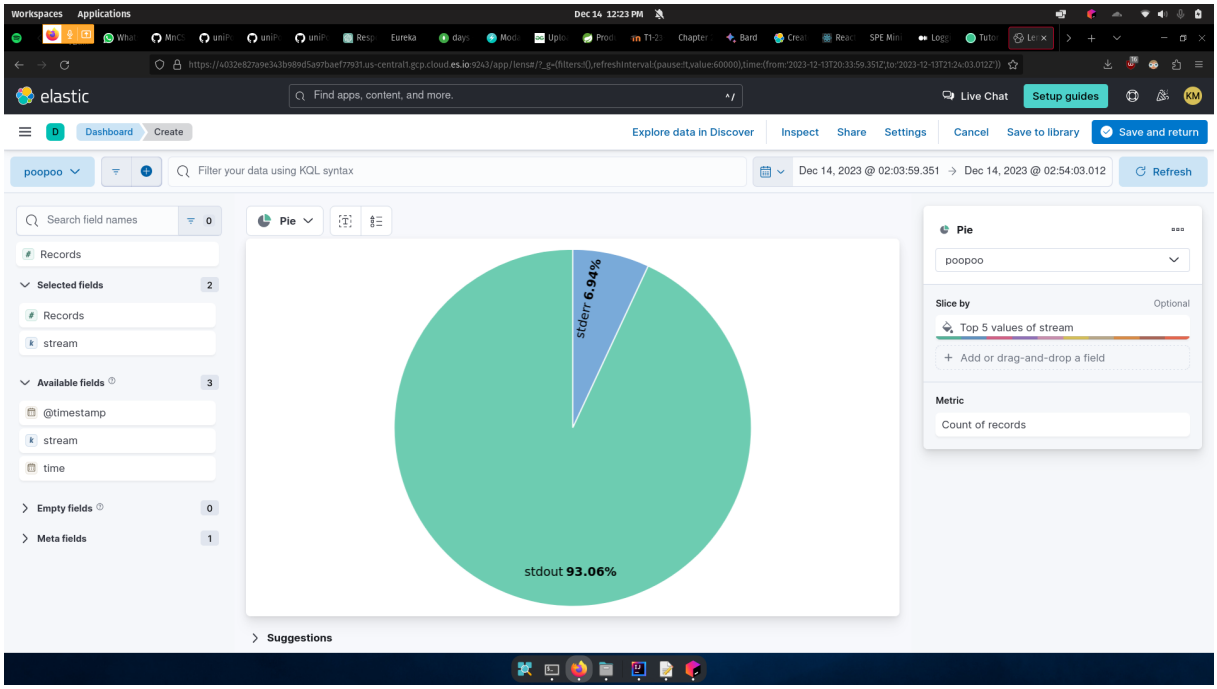

Figure 36: Pie chart comparing stderr vs stdout logs

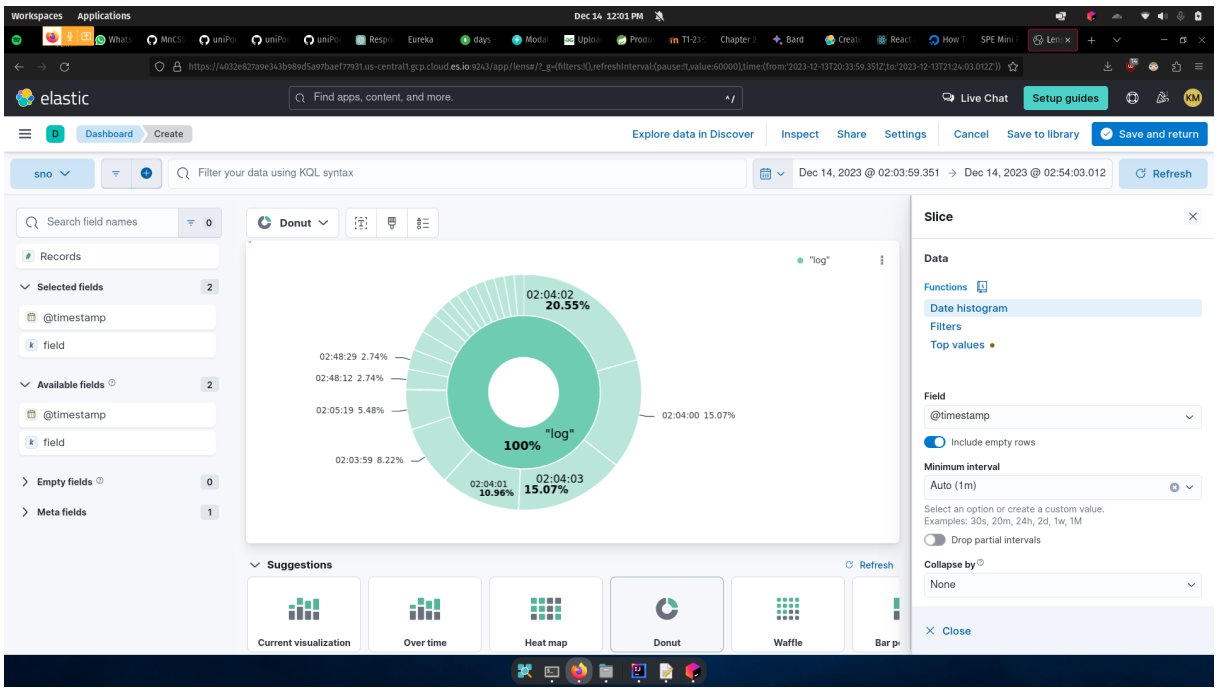

Figure 37: Donut Visualization

Challenges Faced

- Using microservice-based architecture was an entirely new venture for the both us. The usual challenges of learning software and getting familiarized with the APIs were present. Once setup, however, the microservice model proved to be a massive help for parallel development, debugging and testing.
- Setting up Communication between microservices within a containerized application: Initially we had the microservices running with the help of IntelliJ IDEA which made it easy to test and develop. However, in the later half of the project when we had to containerise microservices, the connection parameters changed. Also since the database setup and the microservice setup were handled by two separate docker-compose files triggered separately using Ansible, we had no clue initially about what URLs to replace the initial localhost URLs with and how to club both of these separate composes together. Upon exploring we understood the Docker virtual network and how compose allows to define the same network to allow any number of containers to be part of a virtual network anytime and communicate with each other.
- Creation of Worker and Control node using Ansible: Setting Ansible was easy, but setting the connection using SSH was not free and full of obstacles. Since both of our nodes were on separate operating systems, the worker node running a Linux Distribution while the control node running macOS the openSSH setup was a bit different than the documentation provided to us. However, the right Google searches helped us to come to the required output.

- Testing a lot of microservices: Since we are a two-person team and have more than one microservice per member it became very difficult to first think about the test cases and then implement the test cases as testing in spring-boot can be done using various approaches. Also in terms of Integration Testing and other coverage testing, we weren't able to put in much effort due to lack of time and knowledge of testing springboot microservices.

Future Scope

- One of the primary goals of this project would be to make it complete on the basic functionality side. Cleaning up the frontend and making it more intuitive as a QOL change.
- Making a mobile app so that anyone can use it on the go. This could be done using flutter as it supports React and would allow us to port some of the code easier.
- Bulk addition of menu items for the canteen menu would be an additional goal that would speed up setting up the app for any new college.
- Excel sheet integration would allow users to download the mess menu and store it for offline access. It would also allow the mess manager to upload the menu as a single Excel sheet as opposed to manually entering for each food item.
- Push Notifications for both Students and Managers and Admins regarding menu updates, unavailability of items, resolution of issues, etc.
- Section that would allow to display of the special menus, or fasting menus during festivals.
- On the deployment side since it is locally deployed on machines/bare-metal server computers, one can switch to a complete cloud-native environment where the microservices architecture shines with load-balancing features and high availability.
- Allow a pre-order service for the canteen where the canteen beforehand knows orders of some users and can help speed up the process of delivering food preventing long queues as the regular customers have fix orders beforehand rather than browsing the menu and picking them.

References

- https://www.guru99.com/elk-stack-tutorial.html
- https://medium.com/@AlexanderObregon/enhancing-logging-with-log-and-slf4j-in-spring-boot-applications-f7e70c6e4cc7
- https://chakra-ui.com/ -> for minimal react code to build a responsive website
- Class Notes slides and material provided.