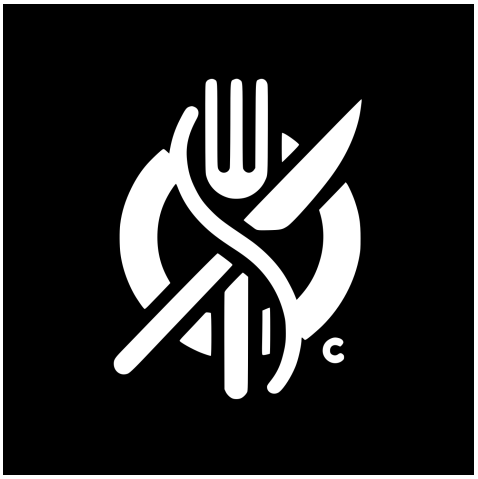
Design Document

Dine Master Pro



Canim

Date	15/01/2025
Version	5.0
State	Unompleted
Author	Samuil Kozarov - product developer

Version history

Version	Date	Author	Description	State	Approved by (teachers only)
0.1	29/09/2024	Samuil Kozarov	Document layout is created, including the main headings	Rough draft (TBA)	
0.2	01/10/2024	Samuil Kozarov	C1 and C2 diagrams are created and added	Rough draft (TBA)	
0.3	02/10/2024	Samuil Kozarov	C3 diagram is created and added	Rough draft (TBA)	
0.4	05/10/2024	Samuil Kozarov	Document purpose and architecture constraints and decisions are added	Rough draft (TBA)	
0.5	09/10/2024	Samuil Kozarov	C1, C2 and C3 diagrams are updated	Rough draft (TBA)	
0.6	11/10/2024	Samuil Kozarov	Some changes for the architecture explanation are made	Rough draft (TBA)	
1.0	11/10/2024	Samuil Kozarov	Final revision before submission	Draft 1.0 (TBA)	Yes
1.1	01/11/2024	Samuil Kozarov	Updating the design decisions	Draft 1.0 (TBA)	
2.0	08/11/2024	Samuil Kozarov	Updating the c4 diagrams	Draft 1.0 (TBA)	
2.1	12/11/2024	Samuil Kozarov	Updating the tools used for routing provider	Draft	
2.2	22/11/2024	Samuil Kozarov	Adding CI/CI pipeline explanations and security design decisions	Draft	
3	27/11/2024	Samuil Kozarov	Update the c4 diagrams	Draft	

4	17/12/2024	Samuil Kozarov	Update the c4 diagrama	Draft	
5	05/01/2025	Samuil Kozarov	Update the c4 diagrama and front end pipeline diagram	Draft	

*TBA - To Be Accepted

Distribution

Version	Date	Receivers and Reviewers
1.0	11/10/2024	Frank Coenen and Bart Rabeling
2.0	08/11/2024	Frank Coenen and Bart Rabeling
3.0	29/11/2024	Frank Coenen and Bart Rabeling
4.0	19/12/2024	Frank Coenen and Bart Rabeling
5.0	16/12/2025	Frank Coenen and Bart Rabeling

Table of Contents

Stakeholders and team members

Name	Abbreviation	Role and functions	Availability
Samuil Kozarov *s.kozarov@student.fontys.nl* *+31 736862736*	*S.K.*	*Product developer*	*Availability: The whole duration of the project* *Days & Hours:* - *Monday (9:00 - 12:00)* - *Wednesday (9:00 - 12:00 | 13:00 - 16:00)* - *Friday (9:00 - 12:00)*
Michael Smith *m.smith@samauto.org* *+31 682 75 9594*	*M.S.*	*Business owner*	*Availability:* *Monday - Friday from 10:00a.m - 11:30p.m.*
Bart Rabeling *b.rabeling@fontys.nl* *+31 885074484*	*B.R*	*Teacher / Tech Support*	*Availability: The whole duration of the project* *Days & Hours:* - *Monday (9:00 - 12:00)* - *Wednesday (9:00 - 12:00)*
Frank Coenen *f.coenen@fontys.nl* *+31 885074348*	*F.C*	*Teacher / Tech Support*	*Availability: The whole duration of the project* *Days & Hours:* - *Wednesday (13:00 - 16:00)* - *Friday (9:00 - 12:00)*

Introduction

Document Purpose

This Software Design Document provides an overview of the system design and describes the architecture of Dine Master Pro, a food ordering system. Dine Master Pro gives its customers the opportunity to browse a menu with numerous items, place orders and track their status (progress). The system also provides item related management functionalities and role-based responsibilities overviews. The main tools used for developing the system are **Java Spring Boot** for the backend, **React** for the frontend and **MySQL** for the database. This document is intended for Project Managers, Software Engineers, and anyone else who will be involved in the implementation of the system.

Architecture

Architecture constraints and design decision

1. Spring Boot

Spring Boot is the required framework to be used which perfectly aligns with the needs of the system. It is efficient in building scalable and maintainable web services and offers an easygoing development experience by reducing configuration complexity, while providing robust support for creating **RESTful APIs** - an essential to the application's architecture. These APIs facilitate seamless communication between the backend and frontend components. Spring Boot naturally supports the implementation of **SOLID** principles, which are essential for building **maintainable** and **scalable** applications. Through its architecture, the framework promotes **Single Responsibility Principle (SRP)** by separating concerns into layers like controllers, services, and persistence. The **Open-Closed Principle (OCP)** is reinforced by allowing easy extension of components through inheritance and configuration without modifying the existing code. The **Liskov Substitution Principle (LSP)** is adhered to by supporting interface-based development, allowing components to be replaced or extended with minimal impact. The **Interface Segregation Principle (ISP)** is facilitated by encouraging smaller, specific interfaces for all use cases within the business layer and the repositories, promoting modularity. Following the rules of the segregation principle (*"A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use"*), the service layer architecture is designed in such a way so that each service operation is a different use case and has separate interface and service class that

implements the interface. This segregation is particularly important for future code refactoring as it guarantees an easy finding of each logic provider class and clear view of all operations made by the class. Lastly, the **Dependency Inversion Principle (DIP)** is a core aspect of Spring Boot through its **Inversion of Control (IoC)** and **dependency injection**, allowing for flexibility.

2. Cloudinary

Cloudinary will be used as an **external software system** responsible for storing images, communicating with both the **backend application** and **frontend application**. It offers a robust and feature-rich platform specifically designed for **media management**, which provides a **scalable cloud storage**, allowing the application to handle large volumes of images without worrying about storage limitations or infrastructure. Its powerful **image optimization** features automatically compress, resize, and deliver images in the most efficient formats, **improving application performance**. The **backend** will communicate to **Cloudinary** through the **Image Upload service class**, which will be responsible for saving the images to the platform. On the other side, the **frontend** will also communicate to **Cloudinary** in order to fetch the already stored images in the platform. This segregation will provide a **secure image management**.

3. React

React JavaScript library will be used for the frontend of the application due the client's requirements and due to its efficiency in building **dynamic**, **responsive**, and **maintainable** user interfaces. React's **component-based architecture** promotes reusability and modularity, making development faster and more scalable and aligns with the **Don't Repeat Yourself (DRY)** principle. I am using **React Router** for seamless navigation between different views, enabling the creation of a **single-page application** (SPA) that delivers a smooth user experience without full page reloads. **Axios** is integrated for handling HTTP requests, making it easy to interact with the backend **RESTful APIs**, ensuring efficient data fetching and state management. This stack ensures that the frontend remains highly performant, maintainable, and aligned with modern web development standards, meeting both client requirements and delivering a seamless user experience.

4. MySQL

MySQL will be used as a **database** due to the past experience the developer has with it and due to its reliability, **performance** and **scalability**. It is a widely used relational database management system and is well-suited for handling structured data and complex queries, ensuring that the application's data is stored efficiently and can be easily retrieved.

5. Flyway

In this application, **Flyway** is used as the database migration tool to manage and version-control MySQL schema changes. This choice ensures consistency and maintainability across environments by automating schema migrations (a schema is first created from start to end and when application is started, it is executed) and allowing each change to be versioned and tracked over time. By integrating Flyway, the development and deployment processes are streamlined, as schema changes are applied automatically, reducing the risk of manual errors and ensuring the database remains synchronized with the application code. It also saves some time. In case of an error, the sql commands can be easily tracked in the migration versions

6. Zustand

In this front-end application, **Zustand** is used for state management due to its simplicity, efficiency, and minimal setup requirements. Zustand offers an intuitive API that allows for easy configuration of global state which prevents unnecessary prop-drilling. What's more it requires less time to set up than React Redux and it's easier to use. This simplicity is ideal for the application, as it lets me manage shared data across the components in the application. Zustand's built-in reactivity ensures that components re-render only when the specific state they depend on changes, which optimizes performance.

7. Leaflet

Leaflet is used as a map provider in the front end as it provides customizable maps in a lightweight and efficient way. Leaflet's straightforward API allows me to quickly add features like markers, pop ups, and custom layers, adapting the map to meet the needs of my application. Lastly, it is free for use and is one of the best cost-effective solutions.

8. Nominatim

I use **Nominatim** for getting location data because it provides a simple, fast, and reliable way to perform geocoding and reverse geocoding using OpenStreetMap data. Nominatim allows me to easily convert place names into geographic coordinates and vice versa, which is crucial for applications that need location-based functionality for some of the features. It's an open-source solution with good coverage and accuracy, making it cost-effective and highly customizable. Additionally, its straightforward API integration fits seamlessly into the application, offering quick access to location data without unnecessary complexity.

9. Geoapify

I use **Geoapify** (Routing API) for calculating the distance between two points on the map because it offers a flexible, and accurate routing solution based on OpenStreetMap data. Geoapify provides fast, efficient calculations for many transportation types such as car, truck, scooter, bicycle, train,etc. which are needed for some of the features of my application. It can be easily integrated through its API and is a cost-effective solution. Its a paid api, but for the current needs of my application I can use the free version.

10. Tailwind CSS

Tailwind CSS will be used for the frontend styling of this project because it provides a highly efficient, utility-first approach (a methodology many small and purpose-specific classes are used for building) that allows for **quick styling** and **easy readability**. With Tailwind, all styles are applied directly within the **HTML**, making it easy to see the applied styles immediately, which accelerates development and debugging. This approach is especially beneficial if the project scales and additional developers are involved in the future, as Tailwind promotes a standardized and consistent styling methodology. Last, but not least, it is a good opportunity to learn it as a new styling language.

11. SMTP Gmail Service

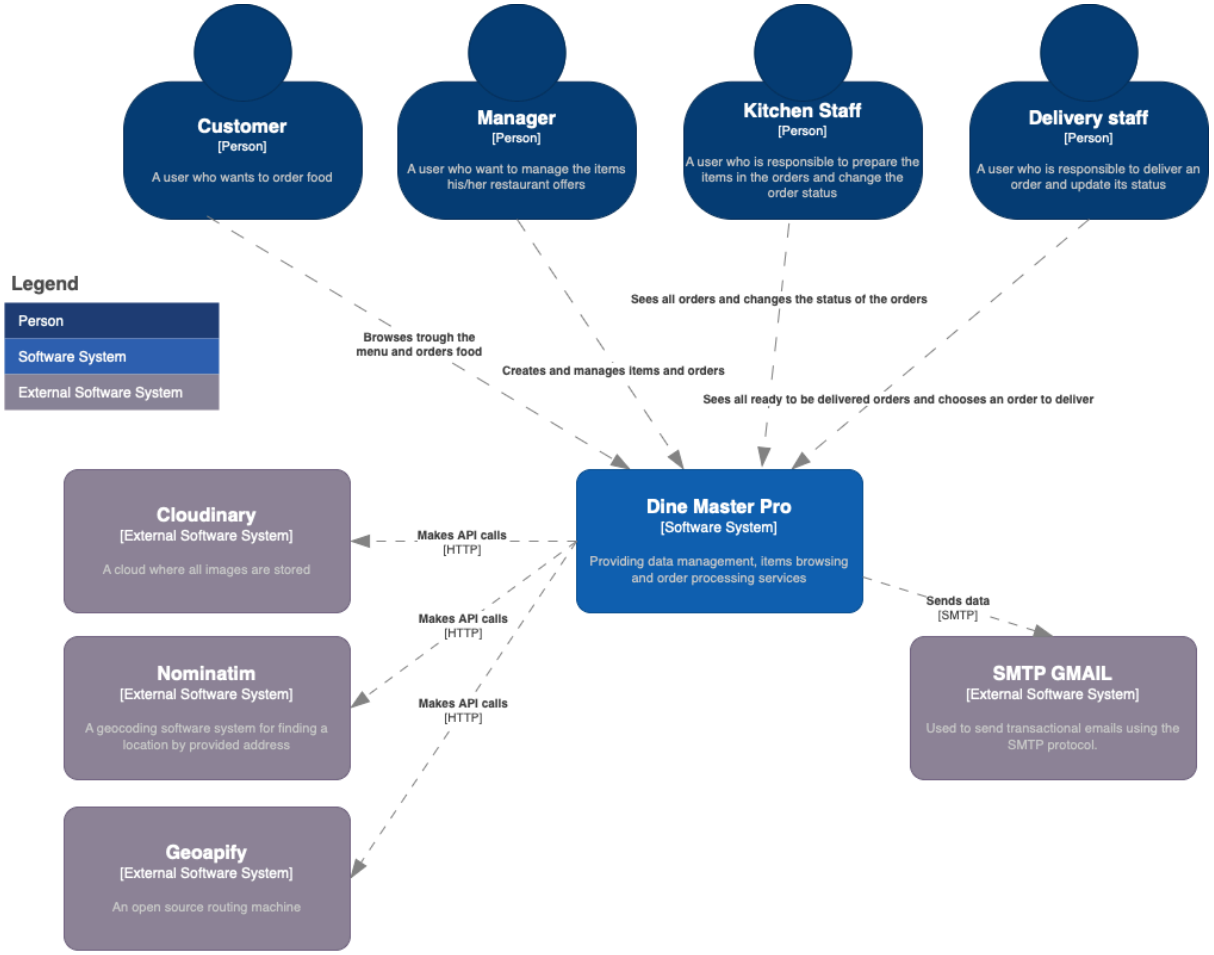
The **Gmail SMTP service** is used to handle email functionality in the application, specifically for sending password recovery emails. The **Spring JavaMailSender** simplifies the integration, enabling seamless email delivery through

a well-documented API. This service is chosen because it does not require a dedicated email server or complex setup.

C4 Model Overview

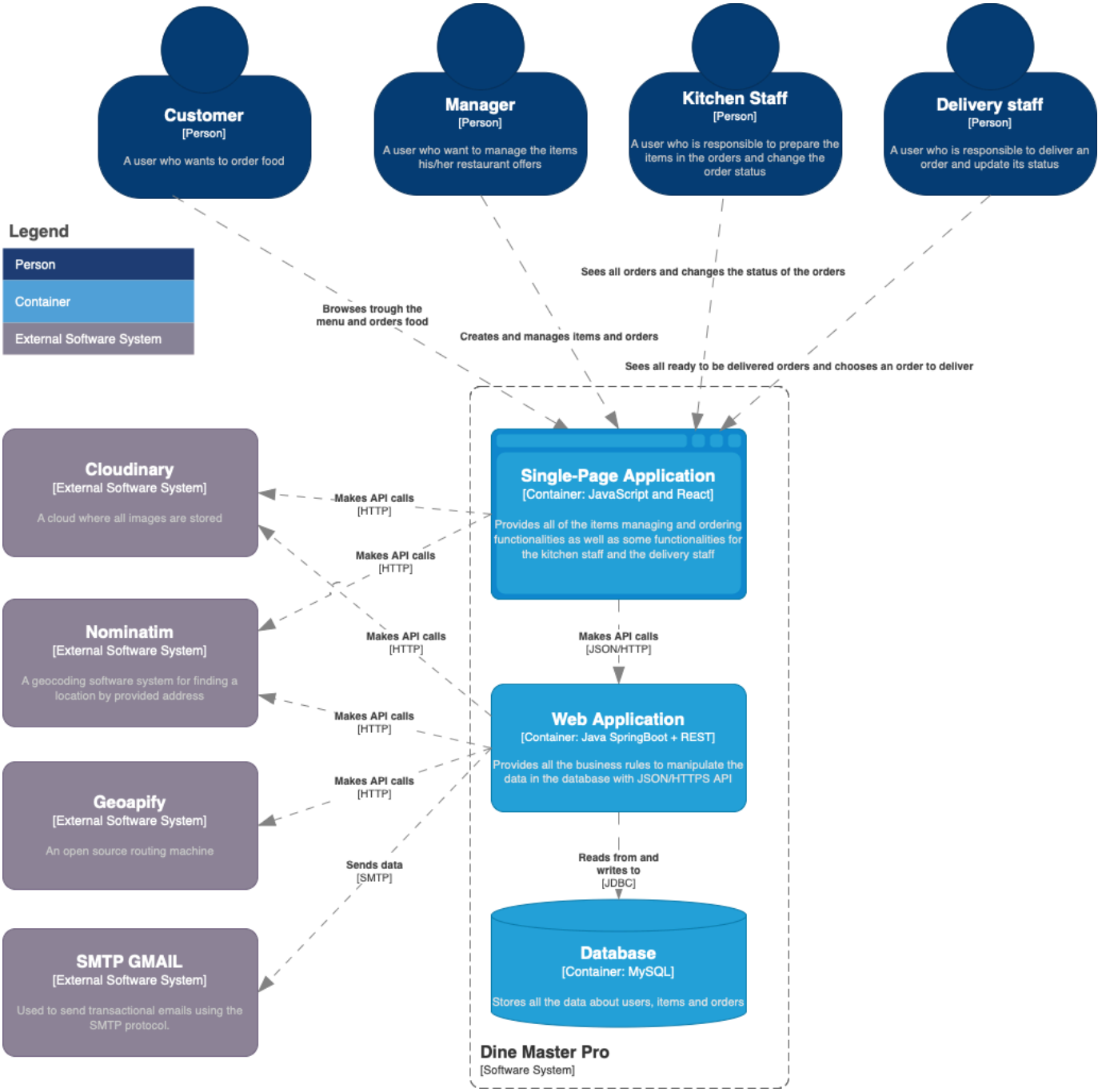
System Context Diagram (Level 1)

This layer shows the software as a whole and the people who use it. Dine Master Pro is accessible by four roles: Customer, Manager, Kitchen Staff and Delivery Staff. Each role has different tasks and responsibilities, hence has different request communication with the software system. The software system references three external systems.



Container Diagram (Level 2)

This layer visualizes the overall shape of the architecture and technology choices. The four different roles of Customer, Manager, Kitchen Staff, Delivery Staff have access to the software system of Dine Master Pro, containing three main containers. On top is the Single-Page Application (SPA) build with JavaScript and React library responsible for the user interactions. Through API calls the SPA communicated with the below located Web Application built with Java Spring Boot and REST APIs, which handles all the business logic. At the bottom is the MySQL database which is responsible for storing all data about users, items and orders.



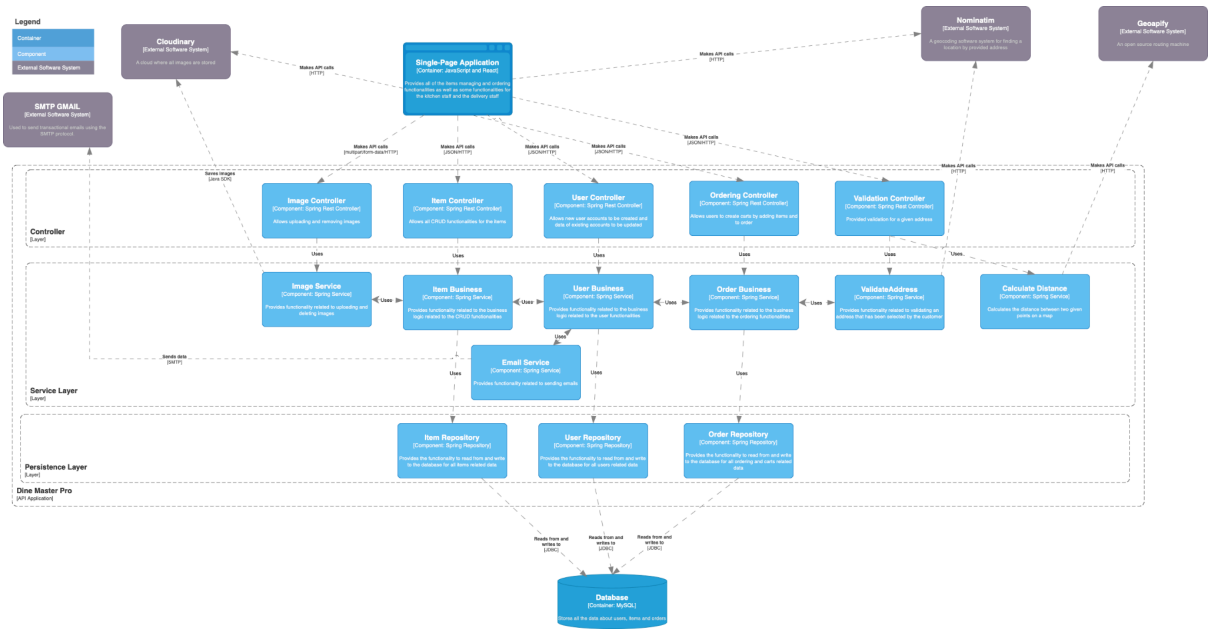
Component Diagram (Level 3)

This diagram visualizes the components and their interactions within the Dine Master Pro API application. The application's architecture is designed into three layers: Controllers, Services and Repositories. The controllers are located at the very top of the application and they are responsible for communicating with the Single-page application. Below them are the services responsible for all business logic operations. At the very bottom is the persistence layer, holding the repositories whose job is to manage all database operations. This architecture aligns all SOLID principles.

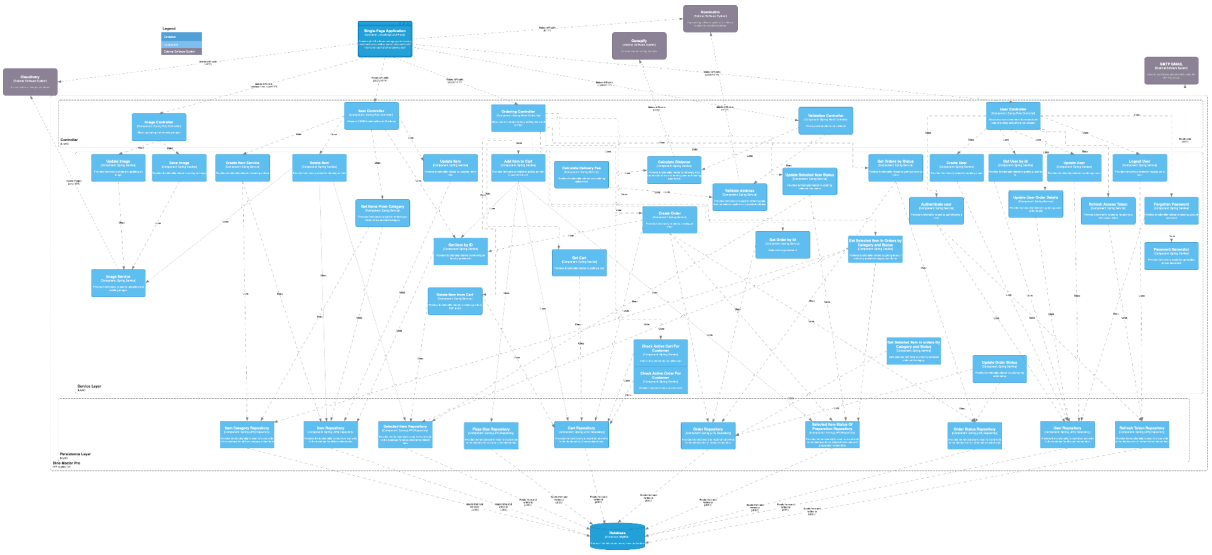
For better quality, please visit the link:

https://drive.google.com/file/d/1Tzr6fhy-WT54D2JdA6oEYzfODy64_tUz/view?usp=sharing

Global Diagram:

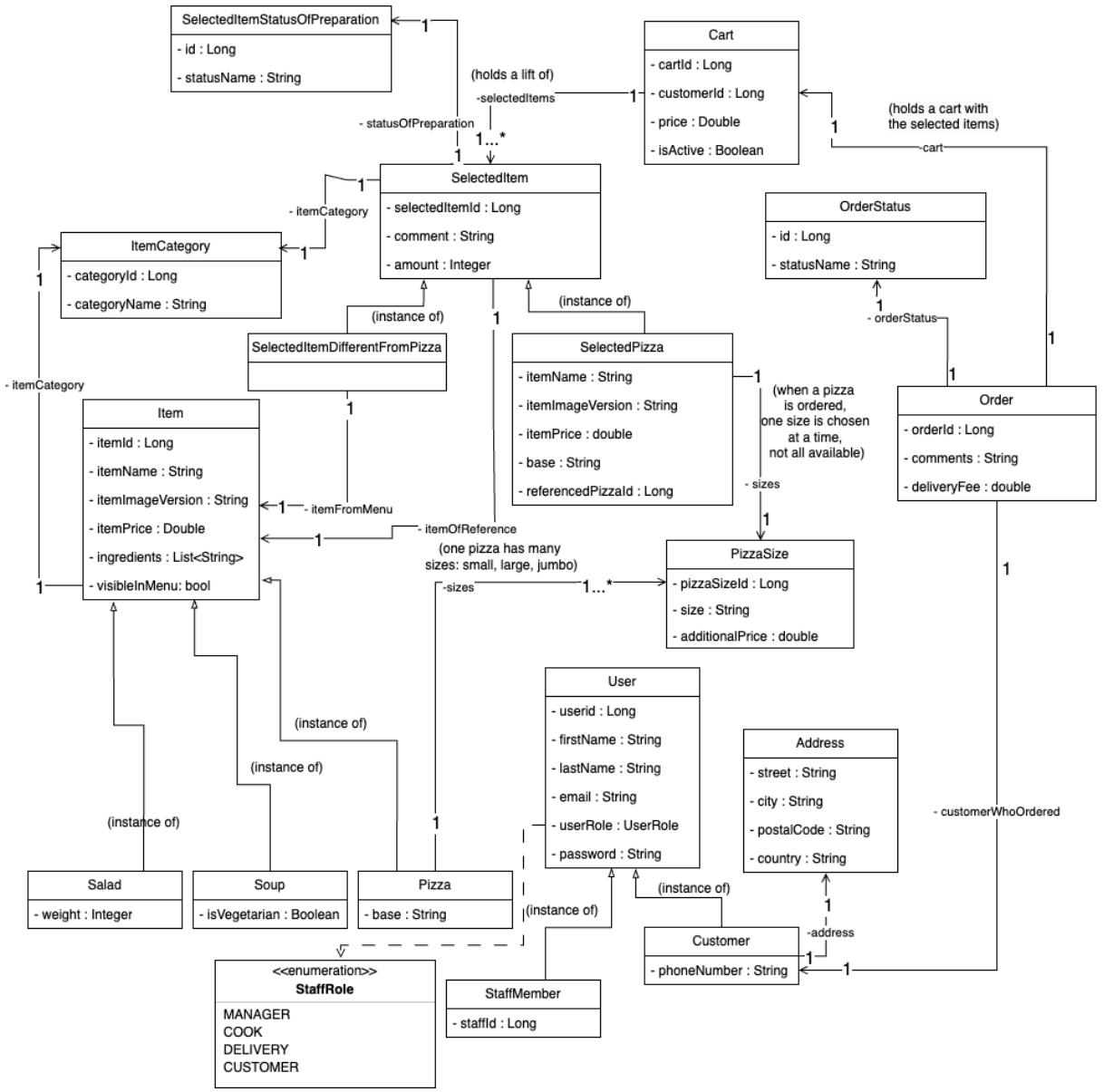


Detailed Diagram:



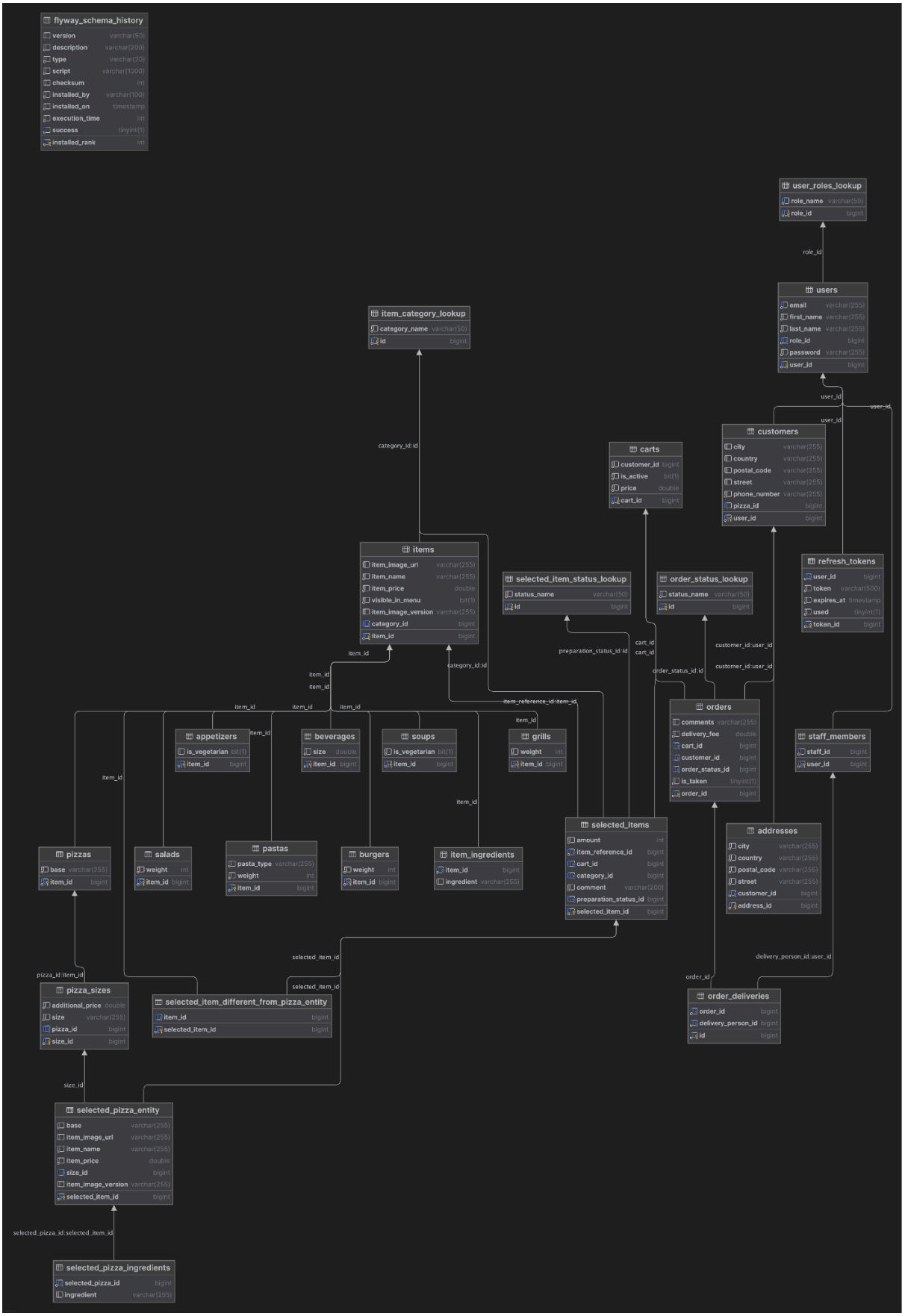
Domain Model Diagram (Level 4)

This diagram visualizes the architecture of the domain in the Dine Master Pro application. It presents the key entities, their fields and the relationships between them. This architecture aligns all OOP (Object-Oriented Programming) principles.



Entity Relationship Diagram

The entity relationship diagram represents the data entities that are used in the application and describes how they are related to each other in the system. This diagram will help the developers to understand how the database is structured and organized and get a better understanding of the system.


flyway_schema_history
version varchar(50)
description varchar(200)
type varchar(20)
script varchar(1000)
checksum int
installed_by varchar(100)
installed_on timestamp
execution_time int
success tinyint(1)
installed_rank int
user_roles_lookup
role_name varchar(50)
role_id bigint
role_id
users
email varchar(255)
first_name varchar(255)
last_name varchar(255)
role_id bigint
password varchar(255)
user_id bigint
user_id
user_id
item_category_lookup
category_name varchar(50)
id bigint
category_id:id
carts
customer_id bigint
is_active bit(1)
price double
cart_id bigint
customers
city varchar(255)
country varchar(255)
postal_code varchar(255)
street varchar(255)
phone_number varchar(255)
pizza_id bigint
user_id bigint
items
item_image_url varchar(255)
item_name varchar(255)
item_price double
visible_in_menu bit(1)
item_image_version varchar(255)
category_id bigint
item_id bigint
selected_item_status_lookup
status_name varchar(50)
id bigint
order_status_lookup
status_name varchar(50)
id bigint
refresh_tokens
user_id bigint
token varchar(500)
expires_at timestamp
used tinyint(1)
token_id bigint
item_id
item_id
item_id
item_id
item_id
item_id
item_id
item_id
item_id
item_id
item_id
item_reference_id:item_id
preparation_status_id:id
cart_id
cart_id
order_status_id:id
customer_id:user_id
customer_id:user_id
appetizers
is_vegetarian bit(1)
item_id bigint
beverages
size double
item_id bigint
soups
is_vegetarian bit(1)
item_id bigint
grills
weight int
item_id bigint
orders
comments varchar(255)
delivery_fee double
cart_id bigint
customer_id bigint
order_status_id bigint
is_taken tinyint(1)
order_id bigint
staff_members
staff_id bigint
user_id bigint
pizzas
base varchar(255)
item_id bigint
salads
weight int
item_id bigint
pastas
pasta_type varchar(255)
weight int
item_id bigint
burgers
weight int
item_id bigint
item_ingredients
item_id bigint
ingredient varchar(255)
selected_items
amount int
item_reference_id bigint
cart_id bigint
category_id bigint
comment varchar(200)
preparation_status_id bigint
selected_item_id bigint
addresses
city varchar(255)
country varchar(255)
postal_code varchar(255)
street varchar(255)
customer_id bigint
address_id bigint
selected_item_id
delivery_person_id:user_id
order_id
pizza_id:item_id
selected_item_id
pizza_sizes
additional_price double
size varchar(255)
pizza_id bigint
size_id bigint
selected_item_different_from_pizza_entity
item_id bigint
selected_item_id bigint
order_deliveries
order_id bigint
delivery_person_id bigint
id bigint
size_id
selected_pizza_entity
base varchar(255)
item_image_url varchar(255)
item_name varchar(255)
item_price double
size_id bigint
item_image_version varchar(255)
selected_item_id bigint
selected_pizza_id:selected_item_id
selected_pizza_ingredients
selected_pizza_id bigint
ingredient varchar(255)

Continuous Integration and Deployment

CI/CD pipeline backend:

Overview

The CI/CD pipeline is defined using a GitLab YAML file, which automates the build, testing, and quality assurance process for the project. This configuration ensures reliable and consistent execution of tasks whenever code is pushed or merged into the repository.

The pipeline includes the following stages:

1. **Build**: Compiles the project.
2. **Test**: Runs unit tests to verify code correctness.
3. **SonarQube**: Performs static code analysis to ensure code quality and adherence to standards.
4. **Docker**: Builds and deploys the Docker image for the application.

Other features:

- **Environment Variables**: Manages sensitive data like the API key securely using `variables`.

Pipeline Structure

1. Variables *(Lines 3-8)*

- `GRADLE_OPTS`: Disables the Gradle daemon to ensure each CI pipeline run uses a fresh, isolated runtime environment. This prioritizes reliability over execution speed.
- `API_KEY_GEOAPIFY`: Dynamically retrieves the API key for Geoapify from environment variables, ensuring sensitive data remains secure.

2. Stages *(Lines 10-12)*

The pipeline is divided into the following stages:

1. **Build**: Compiles the code and ensures all dependencies are resolved.
2. **Test**: Runs automated unit tests to verify code correctness.
3. **SonarQube**: Analyzes the code using SonarQube to ensure quality standards are met.
4. **Docker**: Builds and deploys the Docker image and container for the application.

3. Build Stage *(Lines 14-19)*

- **Execution Node**: GitLab Runner
- **Purpose**:
    1. Compiles the project using Gradle.
    2. Prepares the application for the subsequent stages.
- **Key Steps**:
    1. Grants execution permissions to the Gradle wrapper (`chmod +x ./gradlew`). (moderator is changed because the application is developed on OS machines)
    2. Verifies the `API_KEY_GEOAPIFY` is set (for debugging purposes).
    3. Executes the Gradle `assemble` task, which compiles the project.
- **Interactions**: GitLab Runner interacts with the project repository to retrieve the source code.

4. Test Stage *(Lines 22-27)*

- **Execution Node**: GitLab Runner
- **Purpose**:
    1. Executes all unit tests to ensure the code is functioning as expected.
- **Key Steps**:
    1. Grants execution permissions to the Gradle wrapper. (moderator is changed because the application is developed on OS machines)
    2. Runs the Gradle `test` task, executing all test cases in the project. (Mockito is used)
- **Interactions:** GitLab Runner executes the tests locally on the runner**.**

5. SonarQube Stage *(Lines 30-46)*

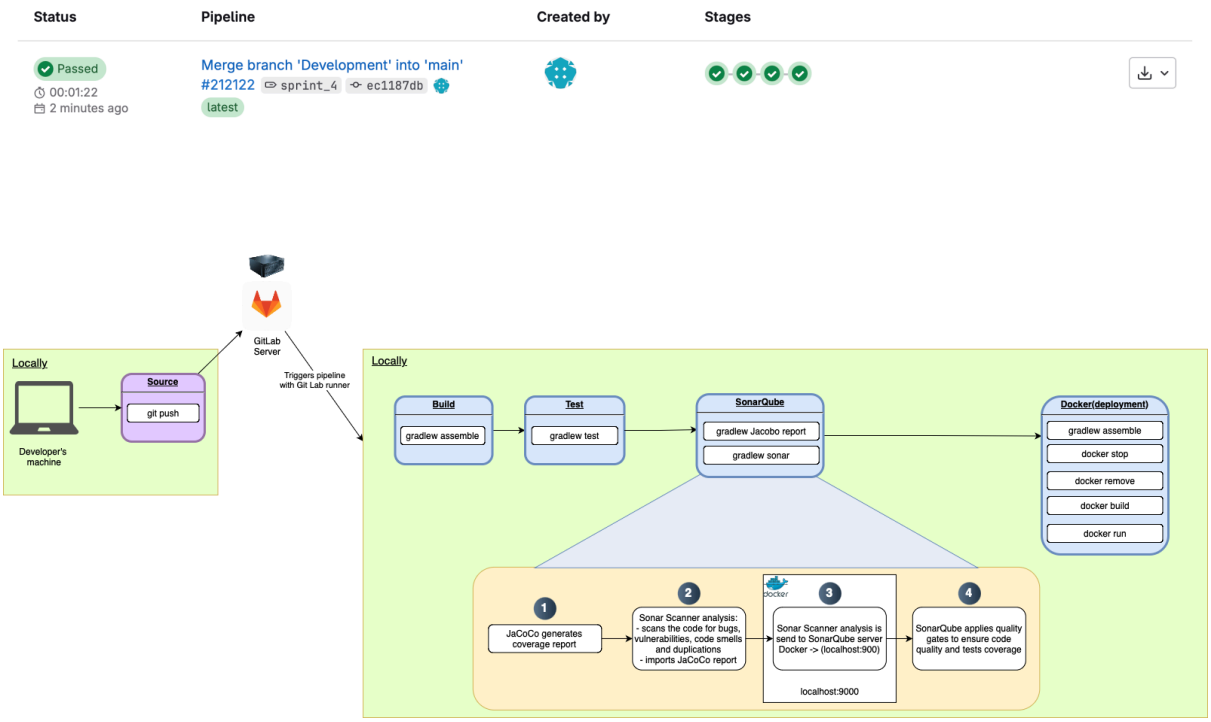
- **Execution Node**:
    - **GitLab Runner**: Executes the Gradle tasks.
    - **SonarQube Server**: Receives code analysis data and performs static code analysis (localhost:9000)

- **Interactions**:
  - Generates code report using JaCoCo
  - GitLab Runner interacts with the SonarQube server (`http://localhost:9000`) to upload the analysis results.
- **Purpose**:
  1. Performs static code analysis using SonarQube.
  2. Ensures the code meets predefined quality standards and satisfies the quality gate.
- **Configuration**:
  1. **Variables**:
     - `SONAR_USER_HOME`: Specifies a local directory to cache SonarQube data for improved performance.
     - `GIT_DEPTH`: Ensures a complete clone of the repository for SonarQube analysis.
- **Key Steps**:
  1. Runs `./gradlew clean test jacocoTestReport`:
     - Cleans the project.
     - Re-runs unit tests.
     - Generates a code coverage report using Jacoco.
  2. Executes `./gradlew sonar` with the following parameters:
     - **`sonar.host.url`**: The SonarQube server URL.
     - **`sonar.login`**: The authentication token for SonarQube.
     - **`sonar.qualitygate.wait=true`**: Ensures the pipeline pauses until the quality gate result is received.
  3. Fails the pipeline if the quality gate conditions are not met.

6. Docker Stage *(Lines 49-59)*

- **Execution Node:** GitLab Runner
- **Interactions**:
  - GitLab Runner communicates with the Docker on the runner machine (locally)
  - The container is attached to the specified Docker network (`flyway_dineMaster_network_staging`).
- **Purpose**:
  Builds and deploys the Docker image for the application. Ensures the application is containerized and running in a production-like environment (`localhost:8090`).
- **Key Steps**:
  1. Grants execution permissions to the Gradle wrapper (`chmod +x ./gradlew`).

2. Executes `./gradlew assemble` to build the application.
3. Stops any running container with the name `dinemasterpro_container` (if exists) to avoid conflicts.
4. Removes the existing container (`docker rm dinemasterpro_container`) if it exists.
5. Removes the existing image (`docker rmi dinemasterpro`) if it exists to ensure a fresh build.
6. Builds a new Docker image using the `docker build` command and tags it as `dinemasterpro`.
7. Runs the Docker container using `docker run` with the following parameters:
    - `-d`: Runs the container in detached mode.
    - `-p 8090:8080`: Maps port `8090` on the host to port `8080` in the container.
    - `--net=flyway_dineMaster_network_staging`: Specifies the Docker network for the container.
    - `--env spring_profiles_active=staging`: Sets the active Spring profile to `staging`.
    - `--name=dinemasterpro_container`: Assigns a name to the running container.



CI/CD Pipeline Frontend

Overview

The CI/CD pipeline is defined using a GitLab YAML file, automating the build, testing, and deployment process for the frontend application. This configuration ensures consistent execution of tasks every time code is pushed or merged into the repository.

The pipeline includes the following stages:

1. **Install**: Installs the project dependencies.
2. **Build**: Builds the application for deployment.
3. **Test**: Executes end-to-end (E2E) tests using Cypress.
4. **Docker**: Builds and deploys the application as a Docker container.

---

1. Stages

The pipeline is divided into four distinct stages:

1. **Install**: Ensures that all necessary dependencies are installed.
2. **Build**: Produces a production-ready build of the application.
3. **Test**: Runs automated tests to verify the application's functionality.
4. **Docker**: Builds a Docker image and runs the application in a container.

Stage Details

Install Stage

Execution Node: GitLab Runner
Purpose:

- Installs all required dependencies for the application.
- Caches dependencies to improve pipeline efficiency in subsequent runs.

Key Steps:

1. Navigate to the frontend directory: `cd dinemaster_fe`.
2. Install dependencies using `npm install`.
3. Cache the `node_modules` directory for future pipeline runs.
4. Save `node_modules` as an artifact for use in the build stage.

Interactions:

- GitLab Runner interacts with the repository to retrieve the source code and `package.json`.

Build Stage

Execution Node: GitLab Runner
Purpose:

- Builds the frontend application into a production-ready format.

Key Steps:

1. Navigate to the frontend directory: `cd dinemaster_fe`.
2. Execute `npm run build` to compile and optimize the application.
3. Save the `dist/` directory (build output) as an artifact for the test and Docker stages.

Interactions:

- GitLab Runner uses the artifacts and cached `node_modules` from the install stage.

Test Stage

Execution Node: GitLab Runner
Purpose:

- Runs end-to-end (E2E) tests using Cypress to ensure the application's functionality.
- Simulates user interactions in a browser-like environment.

Key Steps:

1. Navigate to the frontend directory: `cd dinemaster_fe`.
2. Start a preview server in the background using `npm run preview`.
3. Capture the server process ID to terminate it after tests are done.
4. Run Cypress tests with `npx cypress run`, dynamically setting the base URL based on the environment.
5. Stop the preview server to avoid blocking subsequent stages and close the current steage.

Interactions:

- GitLab Runner runs tests on the built application using the Cypress Docker image.

Docker Stage

Execution Node: GitLab Runner
Purpose:

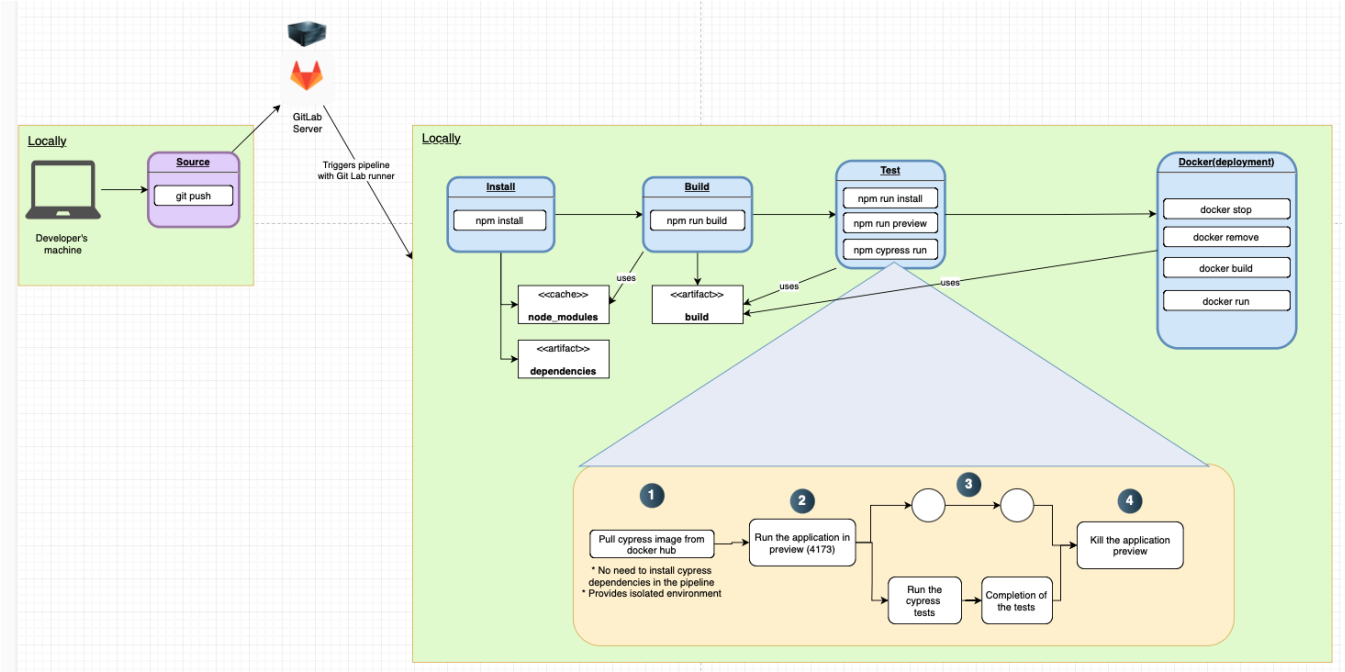
- Builds a Docker image for the application.
- Deploys the application in a containerized environment.

Key Steps:

1. Navigate to the frontend directory: `cd dinemaster_fe`.
2. Stop and remove any existing container named `dinemaster_fe_container` (if it exists).
3. Remove any existing Docker image named `dinemaster_fe` to ensure a fresh build.
4. Build a new Docker image using `docker build -t dinemaster_fe .`.
5. Run the Docker container with the following parameters:
    - Detached mode (`-d`). (this is used in order the container to work in the background which is a good practice for web servers)
    - Maps port 80 on the host to port 80 in the container (`-p 80:80`).
    - Assigns the name `dinemaster_fe_container` to the running container.

Interactions:

- GitLab Runner communicates with Docker to build and run the container.



System Security

Authentication and Authorization Overview

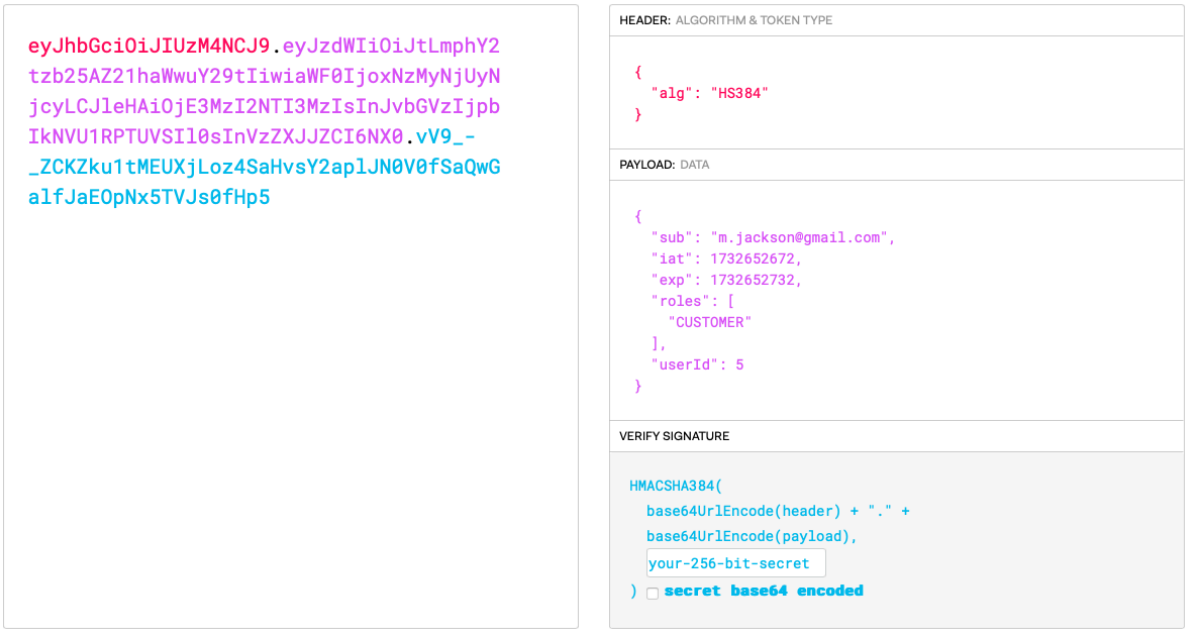
In our application, **authentication** and **authorization** are handled using **JWT-based tokens** - specifically **Access Tokens** and **Refresh Tokens**. These tokens ensure secure communication between the client and the backend while enforcing proper access control.

Access Token

The **Access Token** is used to authenticate the user for each API request. It contains information about the user and the roles assigned to them, and it has an expiration time to limit the validity of the token.

- **Structure:**
  - **Subject**: The subject of the token, typically the user's ID or username.
  - **Roles**: A set of roles assigned to the user, which determines what actions they are authorized to perform.
  - **UserId**: A unique identifier for the user.
  - **Expiration**: The access token has a short expiration time of **20 minutes**. After this time, the token becomes invalid and requires renewal.
- **Usage**:

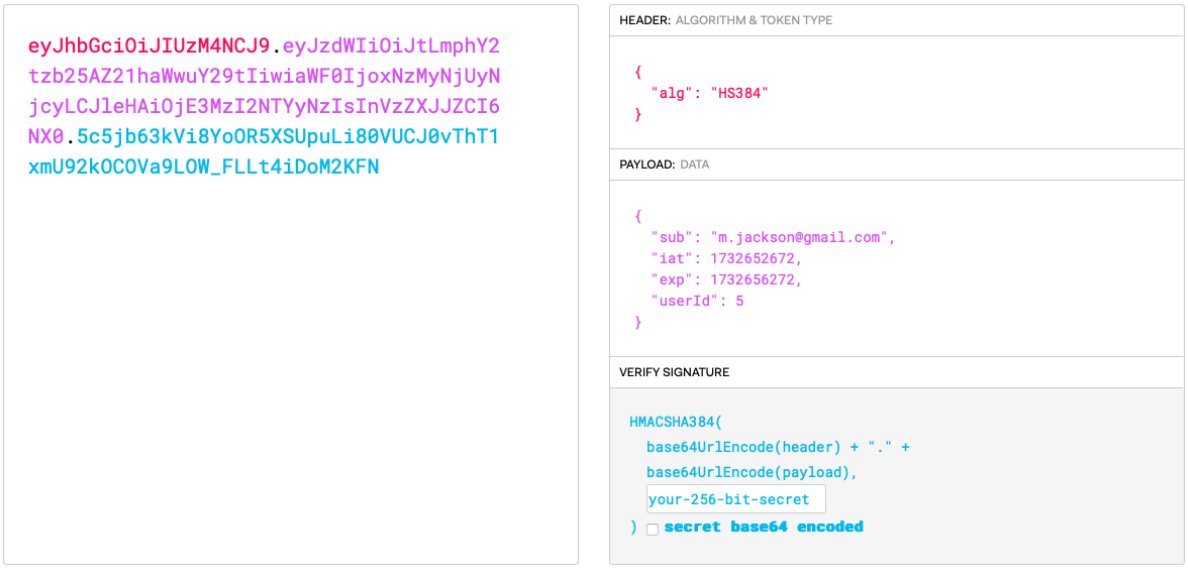
- The access token is sent in the **Authorization header** of API requests, prefixed with the word **"Bearer"**(e.g., `Authorization: Bearer <access_token>`).
- If the token is expired or invalid, the request will be rejected, and a new token can be requested using the refresh token.



Refresh Token

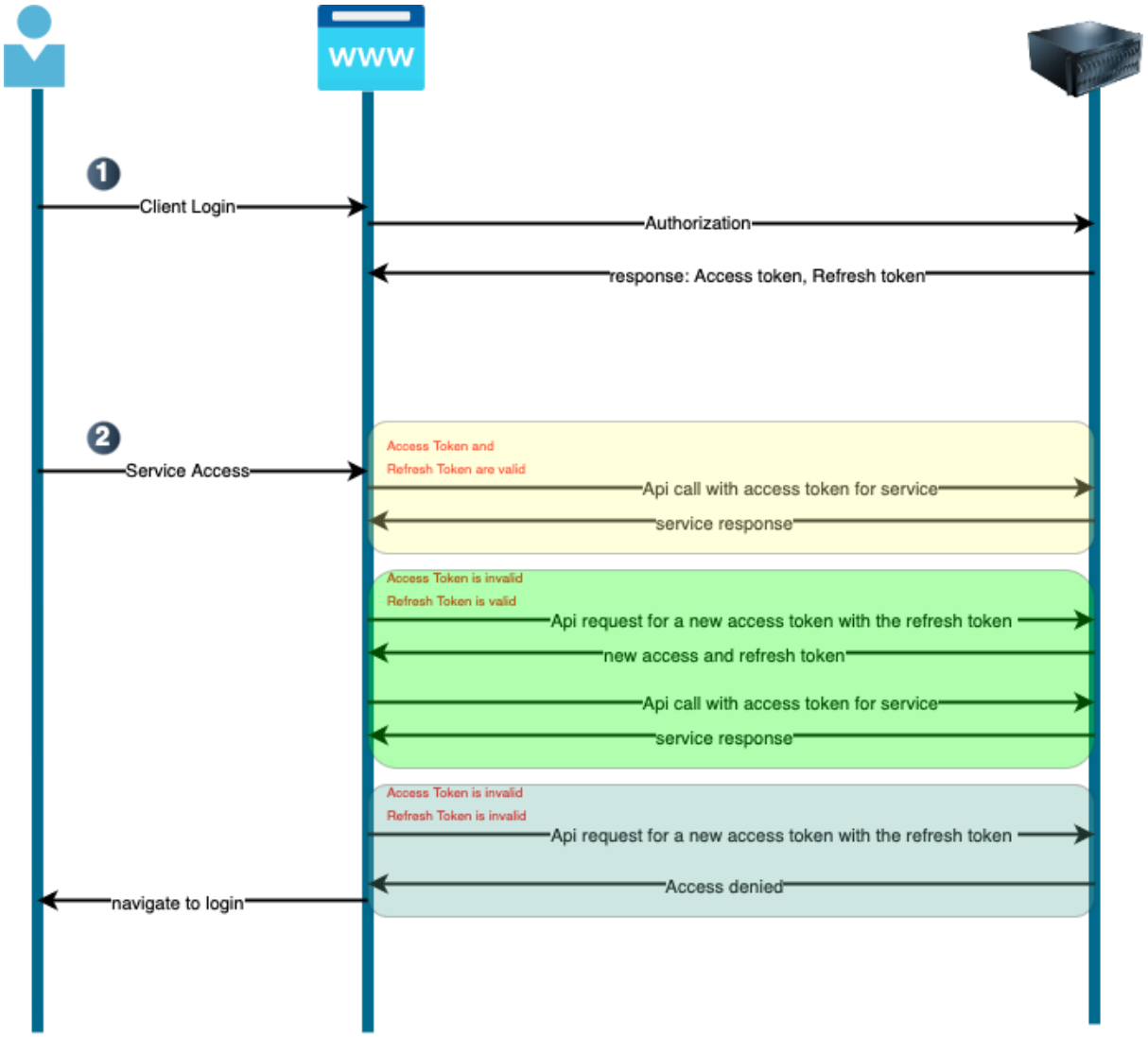
The **Refresh Token** is used to obtain new access tokens after the old one expires. It has a longer expiration time than the access token and is stored securely in the backend.

- **Structure**:
  - **Subject**: Similar to the access token, it contains the subject information, usually the user's unique identifier.
  - **UserId**: A unique identifier for the user.
  - **Expiration**: The refresh token has a validity period of **60 minutes**.
- **Usage**:
  - When the access token expires (e.g., after 20 minutes), the client sends the refresh token to the backend to request a new pair of access and refresh tokens.
  - The refresh token is validated and checked if it has already been used (or marked as expired).
  - If the refresh token is valid, the backend issues a new access token and refresh token pair.
  - If the refresh token has been used or is invalid, the user is forced to log in again.



Refresh Token Automatic Reuse Detection is implemented with the following strategy and working flow: https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/

Token Flow Overview



Git Branch Strategy

The branching strategy of DineMaster Pro follows the conventions from this article:

https://medium.com/@abhay.pixolo/naming-conventions-for-git-branches-a-cheatsheet-8549feca2534