SPE MAJOR PROJECT

Topic: Courier Delivery Management System

Team Members:

1. Manish Reddy Koppula (IMT2020019)
2. Akhil Tavva (IMT2020124)

Project Description:

The Courier Delivery Management Application is a platform where users can keep the list of items they deliver from one place to another place and can see the lists of couriers so they can manage. The main goal is to create an application for keeping track of couriers in digital instead of manual and placing orders from one place to another without going anywhere.

GitHub repo: https://github.com/ManishReddyK/SPE_Final_Proj.git
Docker Hub repositories:
https://hub.docker.com/repository/docker/manish3693/backend/general
https://hub.docker.com/repository/docker/manish3693/frontend/general

Steps to run the application:

1. After cloning/downloading the repo, open the terminal in the front end.
2. Then use " npm install " and then " npm start" to start the frontend.
3. Then open the backend directory in the backend and run the main().

DevOps Tools Used

❖ Version Control System(Git, GitHub):

Version control systems like Git and platforms like GitHub offer numerous benefits to developers and teams. They enable seamless collaboration by allowing multiple contributors to work on the same project at the same time without conflict. Git tracks code changes, allowing for easy rollback to previous versions, project stability, and experimentation without jeopardising the core codebase.
As a hosting platform for Git repositories, GitHub improves collaboration with features such as pull requests, issue tracking, and code reviews. It acts as a centralised hub for code management, encouraging transparency, documentation,

and effective team coordination, which leads to increased productivity and better software development practices.

❖ CI/CD Pipeline:

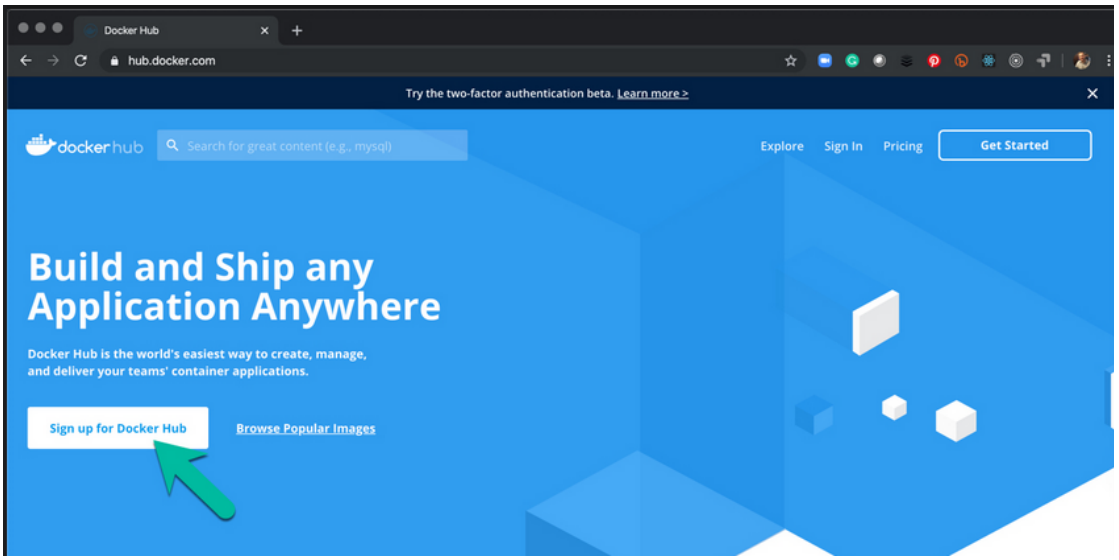
A CI/CD pipeline (Continuous Integration/Continuous Delivery) streamlines the software development process by enabling frequent and efficient code integration, testing, and deployment. This pipeline shortens development cycles, improves software quality, and allows for rapid, dependable, and iterative application delivery, fostering a more agile and responsive development environment.

We use Jenkins as CI/CD pipeline in this application. Jenkins is a popular open-source automation server that facilitates Continuous Integration and Continuous Delivery (CI/CD) in software development. It allows for the automation of building, testing, and deploying code, enabling teams to streamline the development process, automate repetitive tasks, and achieve faster software delivery with increased reliability.

❖ Containerization(Docker):

Containerization is a technology that enables applications and their dependencies to be packaged as lightweight, portable containers, ensuring consistency across multiple environments.

Docker, a popular containerization platform, allows you to create, deploy, and manage containers. It employs containerization technology to encapsulate applications and their dependencies into containers that can run on any system, allowing for easier deployment, scalability, and consistent operation of applications across a variety of environments, including on-premises, in the cloud, and hybrid configurations. Docker streamlines the development and deployment process by isolating applications within containers, allowing for more efficient resource utilisation, and ensuring that software runs consistently across multiple environments.



❖ Ansible:

Ansible is a configuration management, application deployment, and task automation open-source automation tool. It streamlines IT operations by allowing users to automate tasks like software provisioning, configuration management, and application deployment. Ansible, which is based on the declarative language YAML, uses "playbooks" to define sets of procedures and configurations that can be executed across multiple systems at the same time. Ansible communicates with remote nodes via SSH or other protocols due to its agentless architecture, making it simple to set up and use. It enables the management and orchestration of complex infrastructures in a centralised manner, ensuring consistency, scalability, and efficiency in IT environments.

❖ Docker Compose:

Docker Compose is a DevOps tool that allows you to define and manage multi-container Docker applications. It configures the services, networks, and volumes required for an application to run using a YAML file. Developers can use Docker Compose to define an application's services (such as databases, web servers, and APIs) in a single file, specify how they interact, and easily spin up the entire application environment with a single command. This simplifies the setup and orchestration of complex multi-container applications, streamlining the development, testing, and deployment process within the Docker ecosystem in a consistent and reproducible manner.

In our project, we use docker-compose to configure and manage frontend, backend and mysql containers in a single docker-compose.yml file. Because of the docker-compose the orchestration of the application will be simple by allowing us to start, stop, and manage multiple containers with a single command.

❖ ELK Stack:

The ELK stack is a powerful combination of three open-source tools used for log management and analysis: Elasticsearch, Logstash, and Kibana.

- ### Elasticsearch:

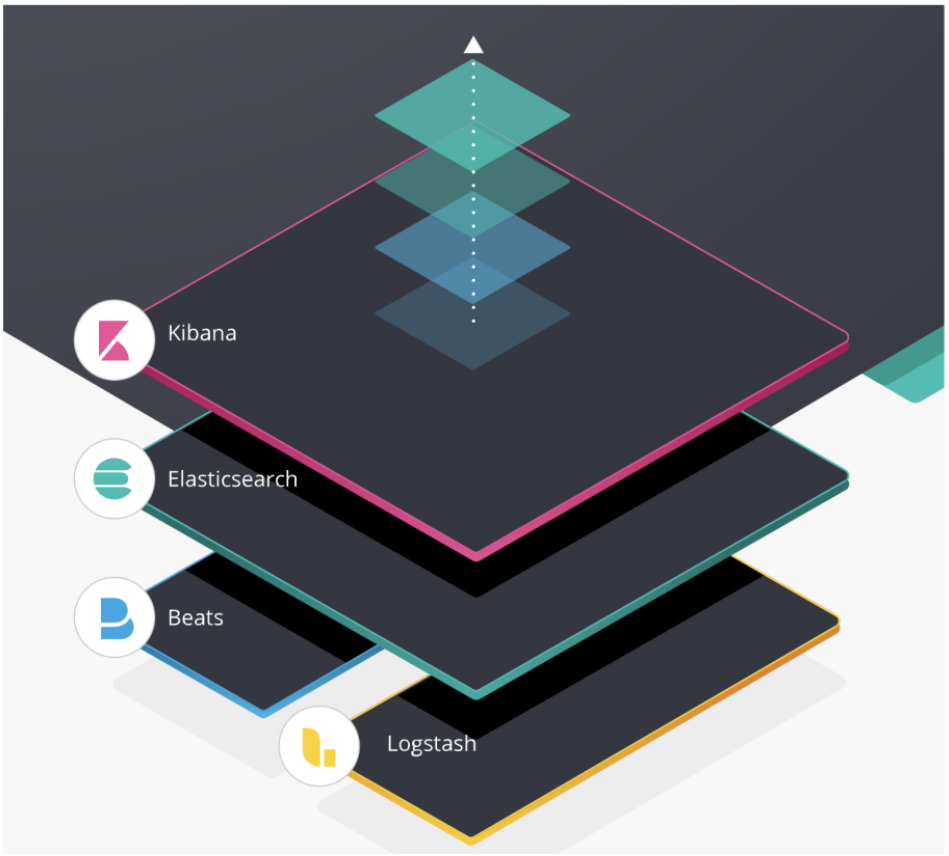
Elasticsearch is a distributed, RESTful search and analytics engine that can store, search, and analyse large amounts of data in near real time.

- ### Logstash:

Logstash is a data pipeline tool that collects, processes, and enriches log data from a variety of sources before sending it to a centralised data store such as Elasticsearch.

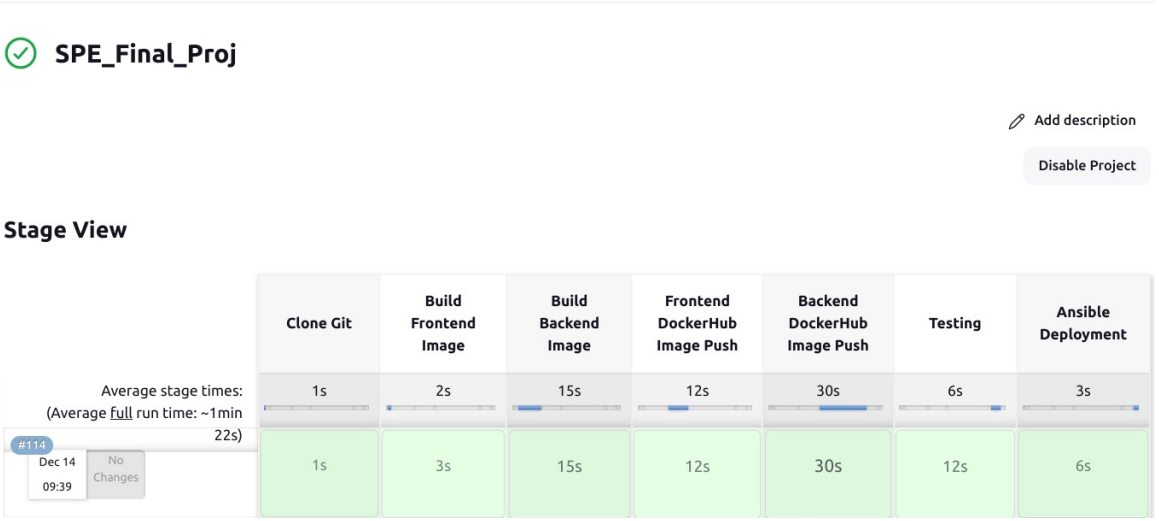
- **Kibana:**

  Kibana is a data visualisation and exploration tool that provides an interface for visualising and analysing Elasticsearch data, with powerful dashboards, graphs, and visualisations to gain insights from log data.



In this project, Index logs are received by Elasticsearch from the containers. It enables us to run complex queries on the logs for analysis. Logstash collects and parses logs from containers before sending them to Elasticsearch. It ensures that logs are properly structured and standardised for analysis. The visualisation tool Kibana then allows us to explore, visualise, and interact with the logs stored in Elasticsearch.

Jenkins Pipeline Stages



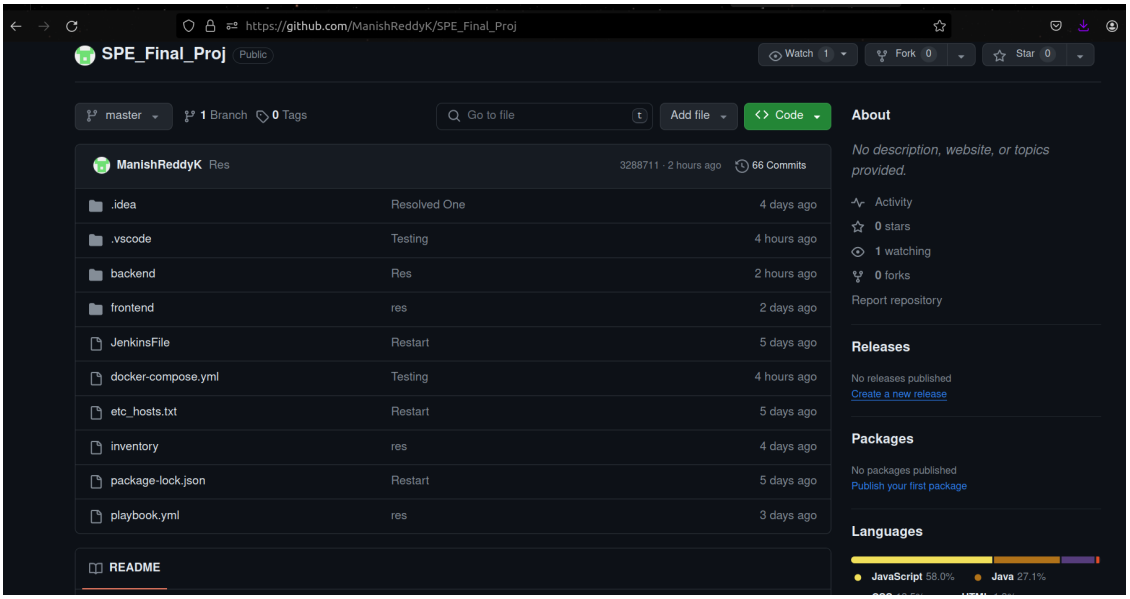
There are total of 7 stages in Jenkins pipeline for this project:

1. Git clone:

Jenkins clones the GitHub repository locally with the help of repo url.

```
stage('Clone Git'){
    steps{
        git branch: 'master',
        url:'https://github.com/ManishReddyK/SPE_Final_Proj.git'
    }
}
```

GitHub Repository:



2. Build Frontend Image:

Jenkins builds the frontend image from the Dockerfile written in the frontend directory.

```
stage('Build Frontend Image'){
    steps
    {
        dir("frontend/")
        {
            sh "docker build -t manish3693/frontend:latest ."
        }
    }
}
```

Dockerfile for frontend is:

```
FROM node:18.0.0-alpine
WORKDIR /
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

3. Build Backend Image:

Jenkins builds the backend image from the Dockerfile written in the backend directory.

```
stage('Build Backend Image'){
    steps
    {
        dir("backend/")
        {
            sh "mvn clean install"
            sh "docker build -t manish3693/backend:latest ."
        }
    }
}
```

Dockerfile for backend is:

backend > Dockerfile > ...

You, 3 days ago | 1 author (You)

```
1  FROM openjdk:17
2  EXPOSE 9090
3  ADD target/delivery-0.0.1-SNAPSHOT.jar delivery-0.0.1-SNAPSHOT.jar
4  ENTRYPOINT [ "java", "-jar", "/delivery-0.0.1-SNAPSHOT.jar" ]
```

4. Pushing Frontend Image to DockerHub:

After building the frontend image in Jenkins, it pushes the image into Docker Hub. To push into DockerHub we store the credentials of docker hub in jenkins to use in the pipeline.

```
stage('Frontend DockerHub Image Push')
{
    steps
    {
        script
        {
            docker.withRegistry('', registryCredential)
            {
                sh "docker push manish3693/frontend:latest"
            }
        }
    }
}
```

5. Pushing Backend Image to DockerHub:

After building the backend image in Jenkins, it pushes the image into Docker Hub. To push into DockerHub we store the credentials of docker hub in jenkins to use in the pipeline.

```
stage('Backend DockerHub Image Push')
{
    steps
    {
        script
        {
            docker.withRegistry('', registryCredential)
            {
                sh "docker push manish3693/backend:latest"
            }
        }
    }
}
```

6. Ansible deployment:

In this stage, Jenkins runs ansible-playbook which deploys the application based on the host mentioned in the inventory file.

```
stage('Ansible Deployment'){
    steps{
        ansiblePlaybook becomeUser: null,
        colorized: true,
        credentialsId: 'localhost',
        disableHostKeyChecking: true,
        installation: 'Ansible',
        inventory: 'inventory',
        playbook: 'playbook.yml',
        sudoUser: null
    }
}
```

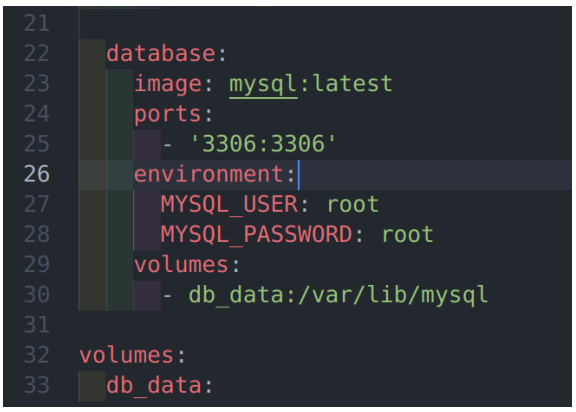
Ansible playbook.yml file is given in below figure:

```
---
- name: Deploy Delivery Application
  hosts: all
  vars:
    ansible_python_interpreter: /usr/bin/python3

  tasks:
    - name: Copy Docker Compose file
      copy:
        src: docker-compose.yml
        dest: "docker-compose.yml"

    - name: Run Docker Compose
      command: docker-compose up -d
```

In the ansible file The use of ansible_python_interpreter in the Ansible playbook specifies the Python interpreter path for the target hosts where the playbook tasks will be executed. In this case, this variable is set to /usr/bin/python3. This variable is often used when the Python interpreter path differs or needs to be explicitly defined on the target machines.

Docker-compose yml file:

```
version: '3'
services:
  frontend:
    image: manish3693/frontend:latest
    ports:
      - '3000:3000'
    depends_on:
      - backend

  backend:
    image: manish3693/backend:latest
    ports:
      - '9090:9090'
    depends_on:
      - database
    environment:
      - MYSQL_HOST=database
      - MYSQL_PORT=3306
      - MYSQL_USER=root
      - MYSQL_PASSWORD=root
```



“Services” defines the services that compose your application. In this file, there are three services: frontend, backend and database.

Frontend:

- In the frontend service we specify the latest version of docker image to use for the frontend.
- For the frontend service, map port 3000 on the host to port 3000 on the container. This allows you to connect to the frontend application via port 3000.
- In the docker compose file, it also shows that the frontend service is dependent on the backend service. This ensures that the backend service is launched before the frontend service.

Backend:

- In the backend service we specify the latest version of docker image to use for the backend.
- For the backend service, map port 9090 on the host to port 9090 on the container. This allows you to connect to the backend application via port 9090.
- In the docker compose file, it also shows that the backend service is dependent on the database service. This ensures that the database service is launched before the backend service.
- Sets environment variables for backend service. The variables we put are mysql user, password, port and host.

Database:

- In the database service we specify the latest version of docker image which was pulled from mysql to use for the database.
- For the database service, map port 3306 on the host to port 3306 on the container. This allows you to connect to the database application via port 3306.
- Sets environment variables for backend service. The variables we use here are mysql user, password.

7. Testing:

In this stage, we have done unit testing by using junit and mockito.

```
stage('Testing') {
    steps {
        dir("backend") {
            // Run AdminControllerTest.java
            sh "mvn test -Dtest=AdminControllerTest"

            // Run CourierDeliveryApplicationTests.java
            sh "mvn test -Dtest=CourierDeliveryApplicationTests"
        }
    }
}
```

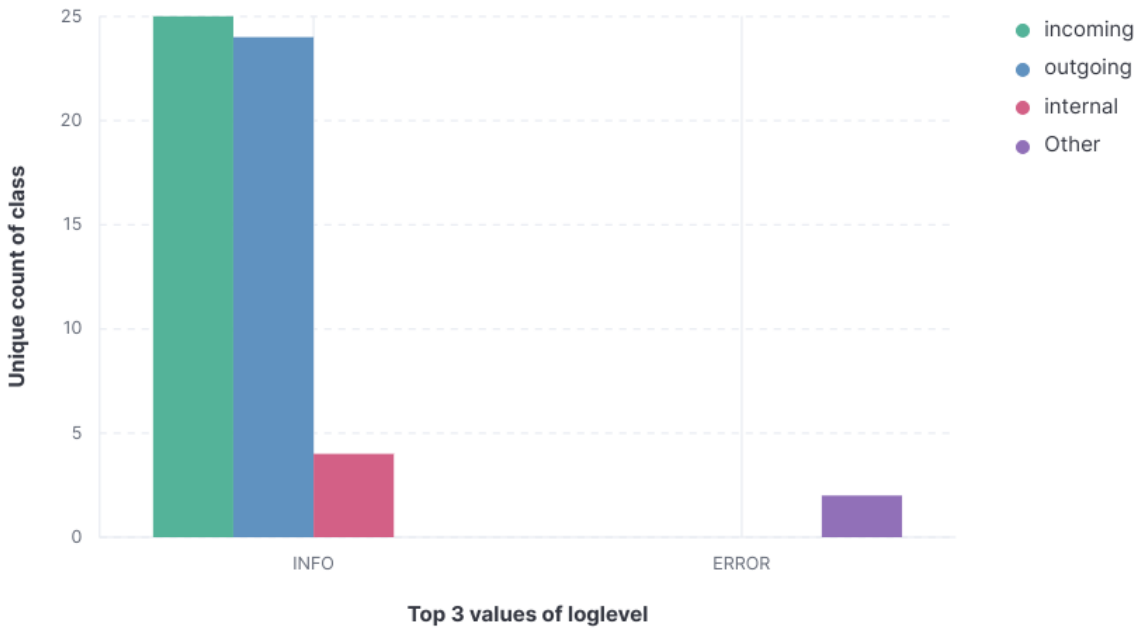
JUnit is a popular open-source Java testing framework that includes annotations and assertions for writing and running unit tests. It enables developers to define test cases, make assertions about expected outcomes, and run these tests to ensure the correctness and reliability of Java code.

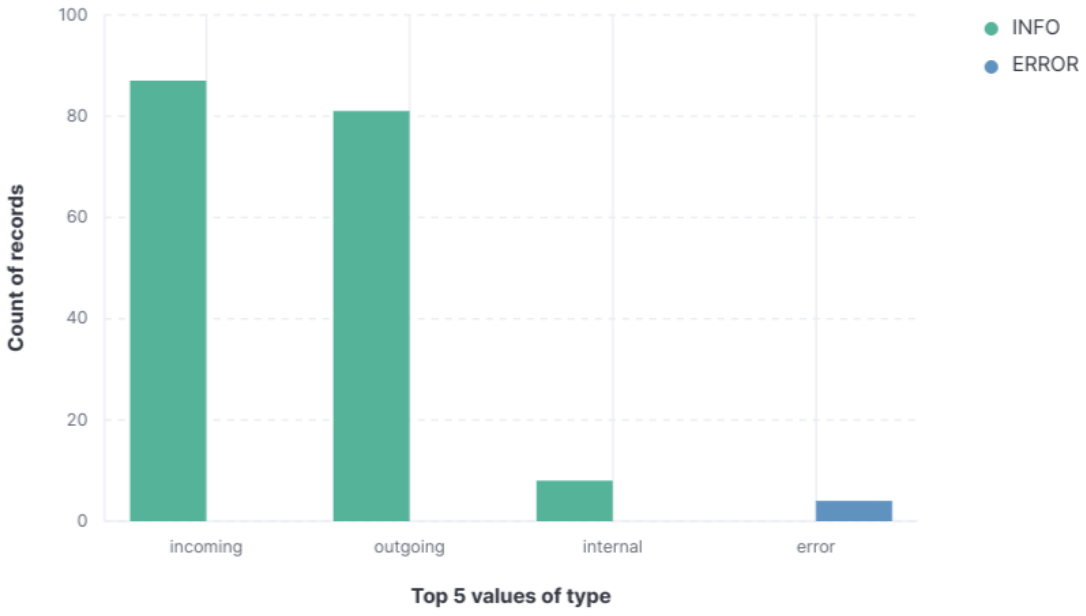
Mockito is a Java framework used in unit tests to create and configure mock objects. It allows developers to simulate the behaviour of dependencies or external systems with which a class is tested, allowing for isolated and controlled testing environments. Mockito provides methods for mocking behaviours, setting expectations, and verifying interactions with mock objects, allowing for more comprehensive and effective unit testing in Java applications.

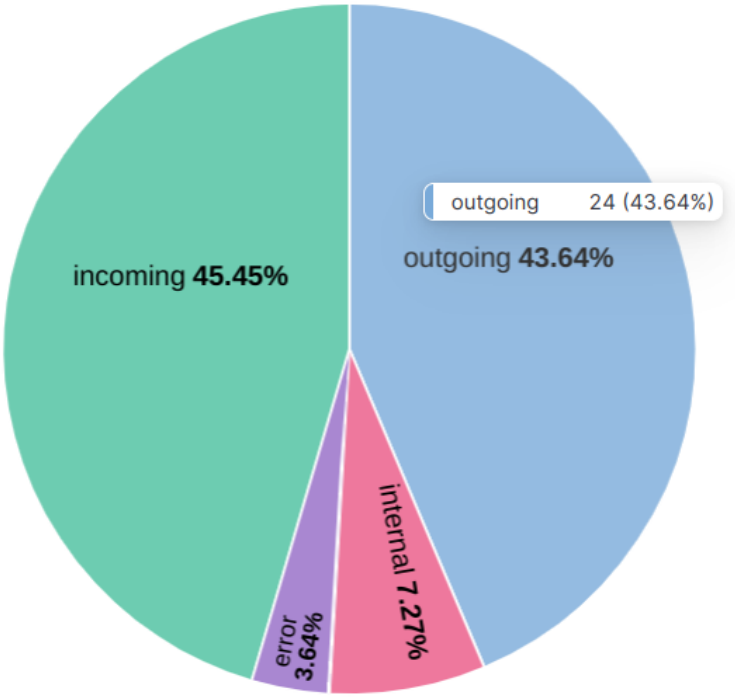
We use these two in admin controller test and user controller test java files.

Logs and Kibana Visualisations:

The following visualisations listed below were created using Kibana by uploading the container log files we generated from containers. In the files where we wrote logger commands we wrote incoming, outgoing, internal. (CAN WRITE MORE)


25
20
15
10
5
0
Unique count of class
INFO
ERROR
Top 3 values of loglevel
incoming
outgoing
internal
Other

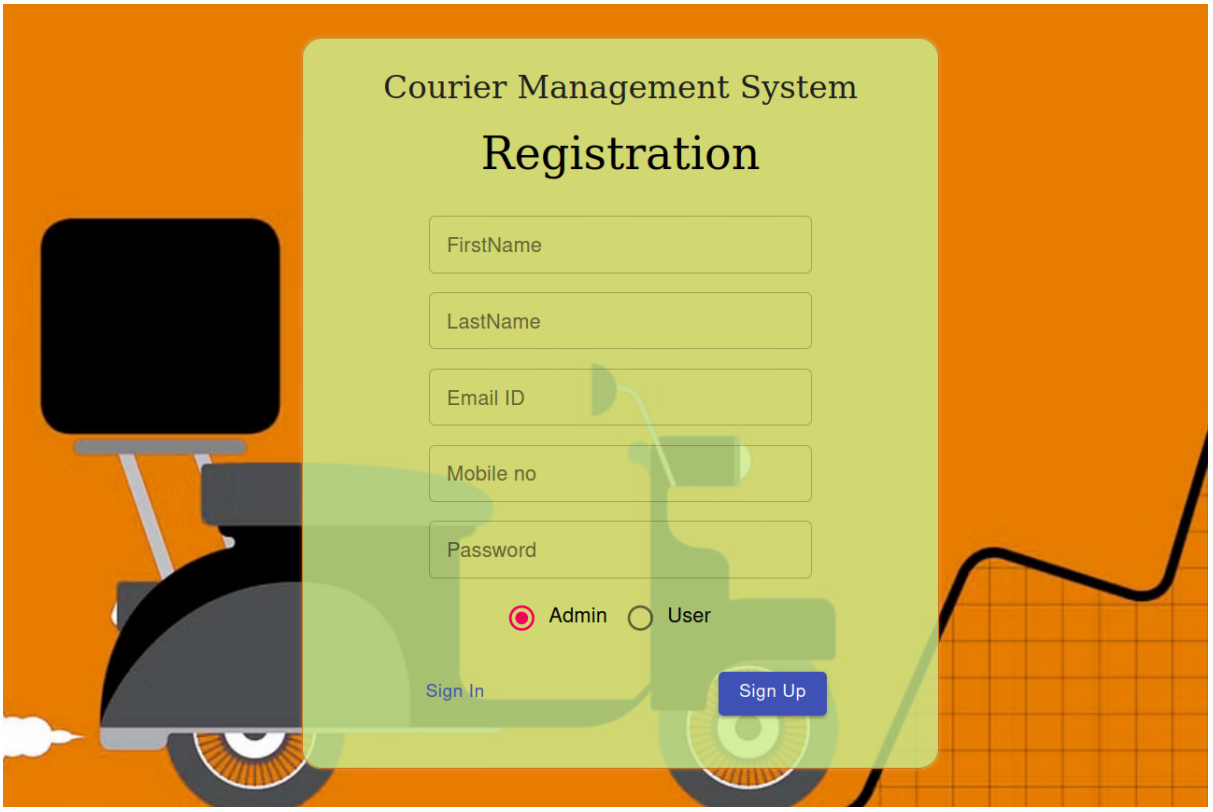

100
80
60
40
20
0
Count of records
incoming
outgoing
internal
error
Top 5 values of type
INFO
ERROR



Frontend

1. SignUp and SignIn page:

The signup page allows you to create a new user in the application with appropriate details and can either sign up as Admin or User. After creating the user, they can enter the application through the Sign in/Login page.


Courier Management System
Registration
FirstName
LastName
Email ID
Mobile no
Password
Admin
User
Sign In
Sign Up

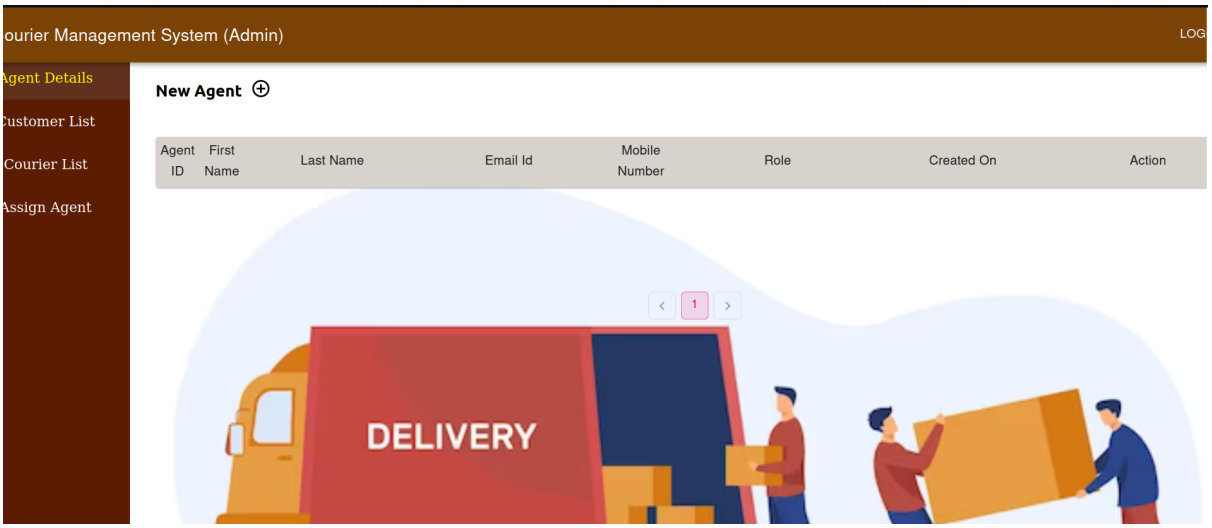

Courier Management System
Log In
Email ID
Password
Create New Account
Sign In

2. Admin Dashboard:

The admin dashboard contains the following:

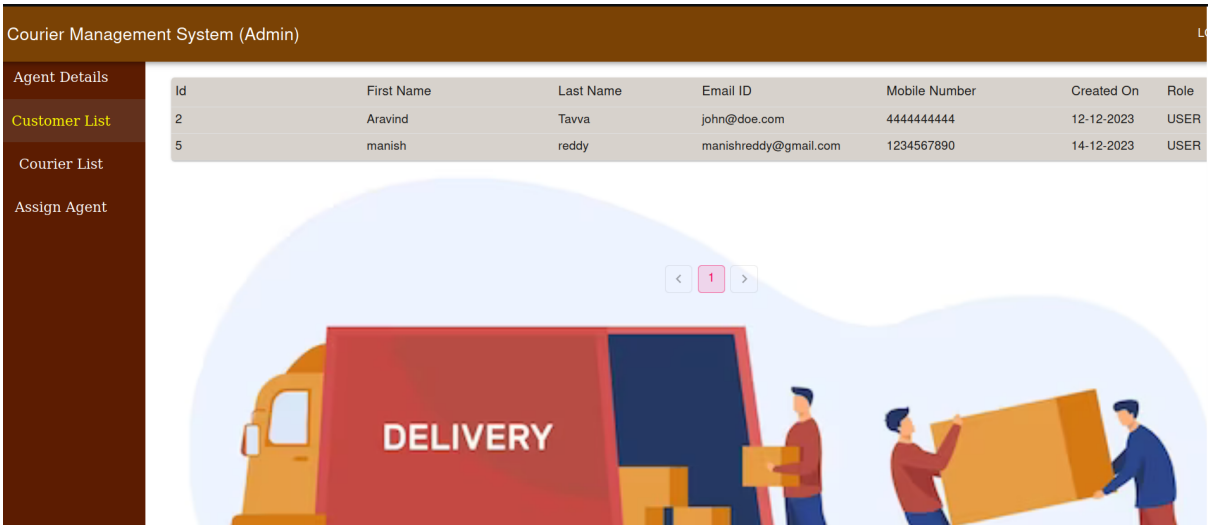
a. Agent Details:

Information of all the agents can be seen by the admin and the admin can add a new agent to the list of existing agents.



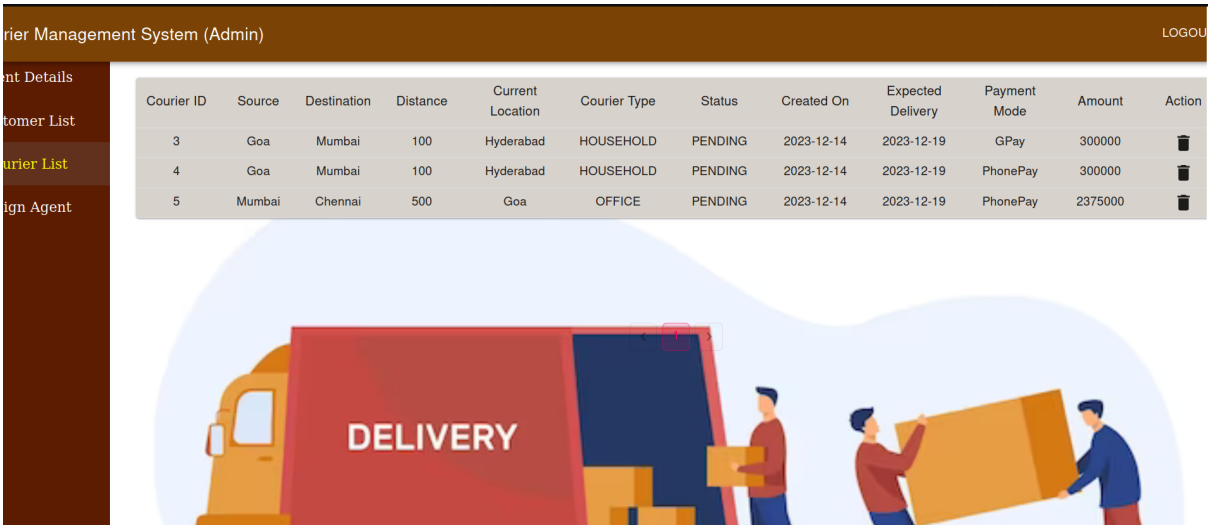
b.Customer List:

It contains all the information of the users like first,last name,email, when it was created etc.



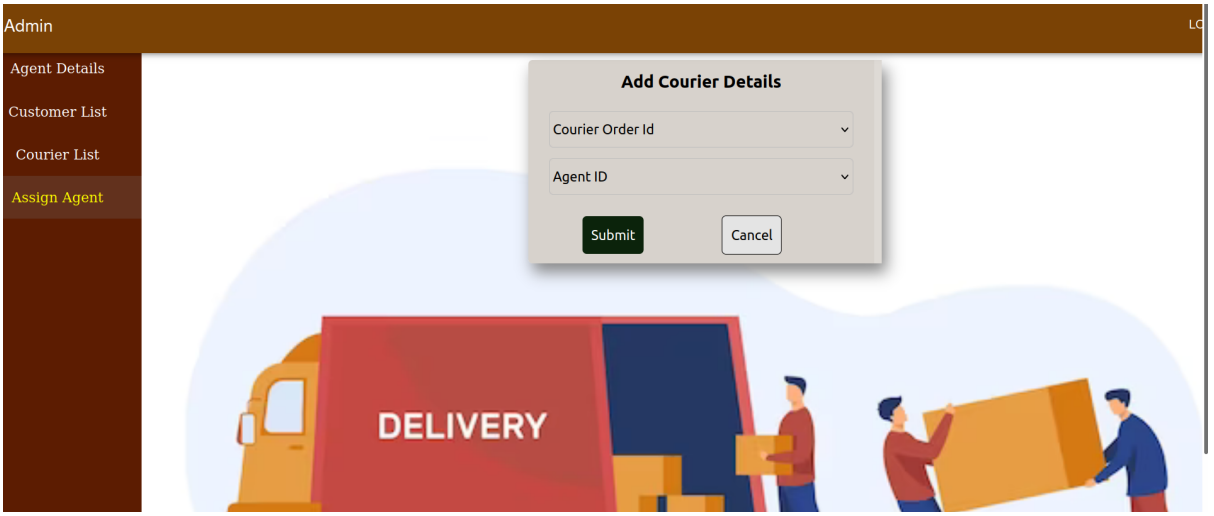
c. Courier List:

It gives all the couriers that are placed by all the users and admin can see and assign an agent to the courier. After a courier is reached then admin will delete the courier from the list.
It also shows the placed order date and delivery date of the courier.



d. Assign Agent:

Admin does this task to assign agents from their ids to the courier.

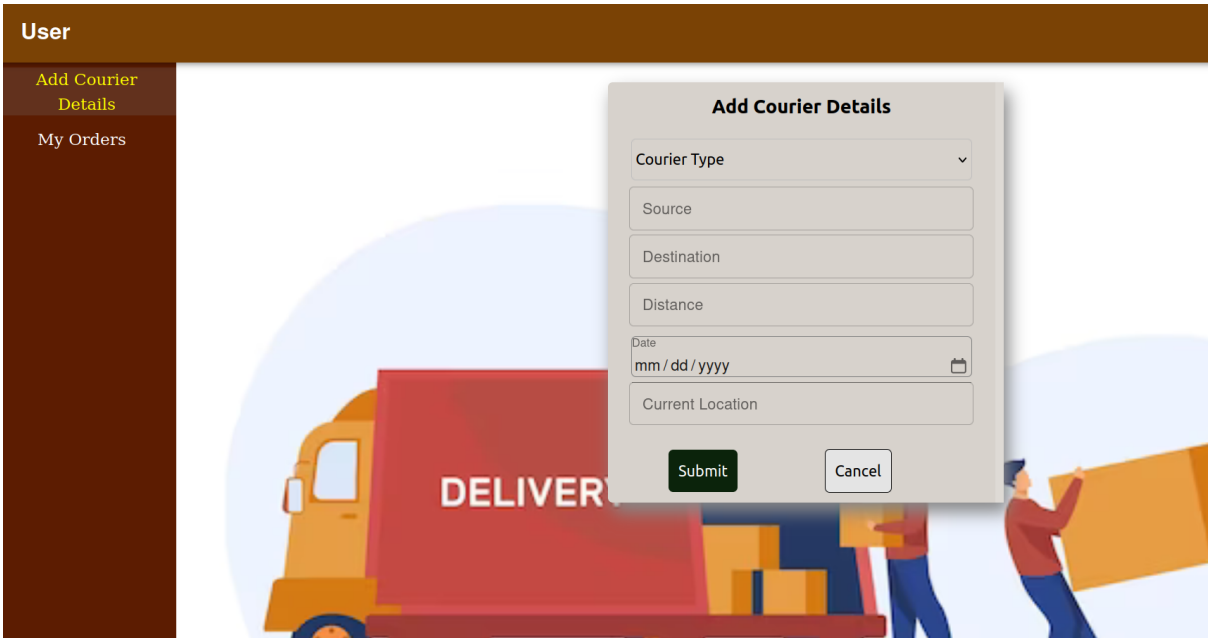


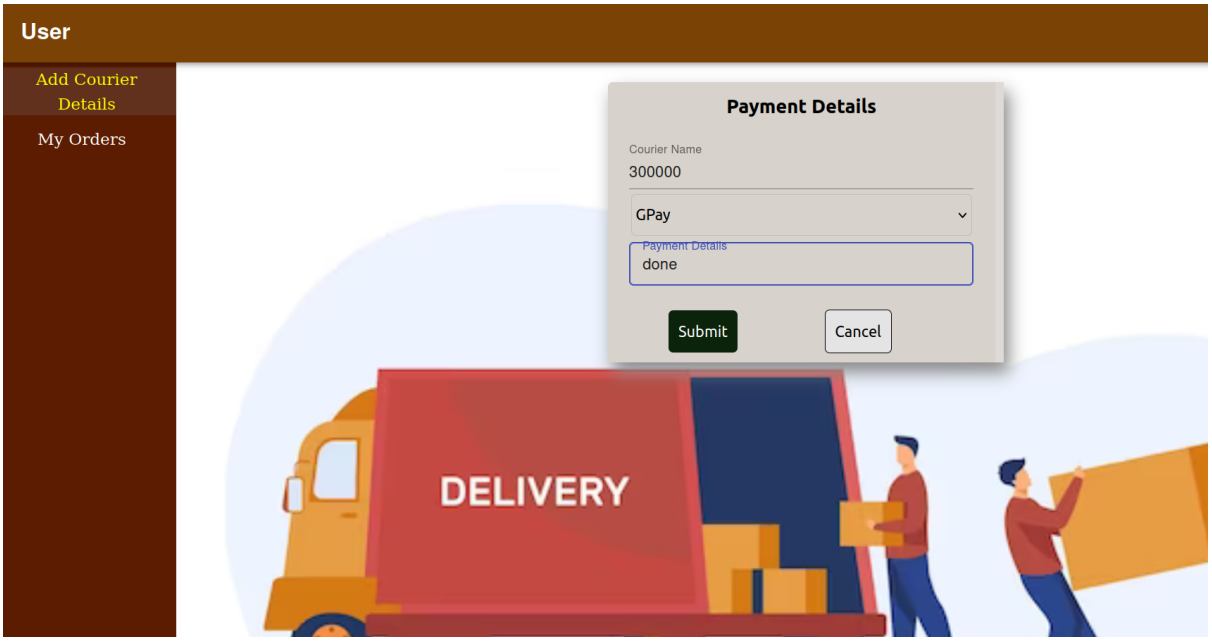
3.User Dashboard:

The user dashboard contains the following:

1. Add Courier Details:

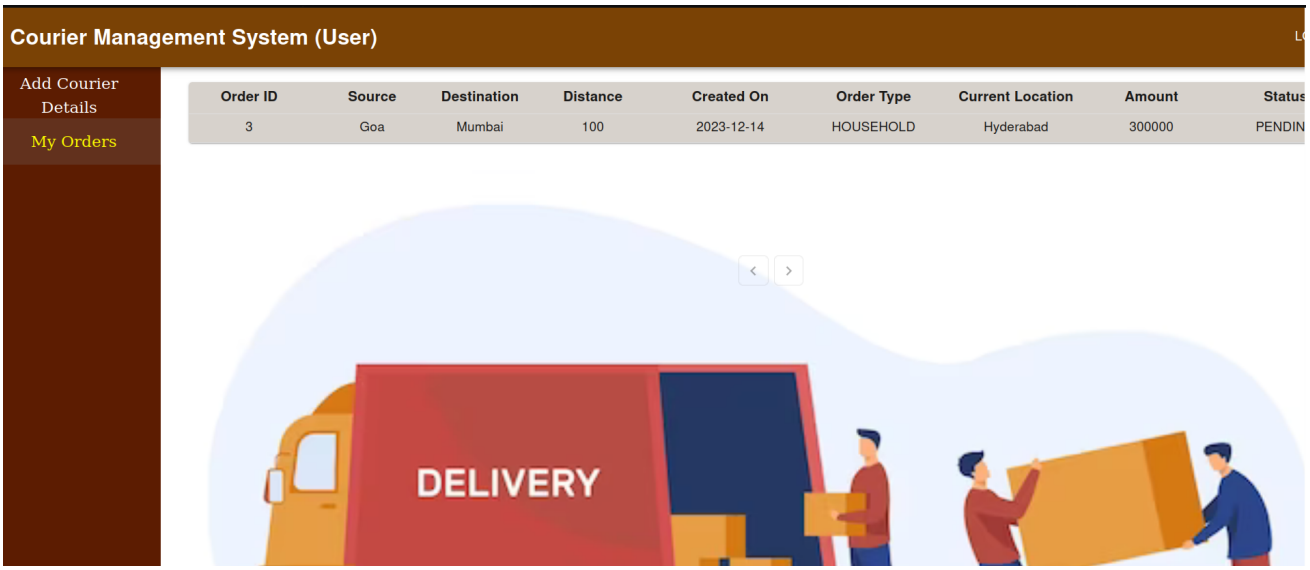
After logging in, the user can add a courier by going to add courier details and entering details such as courier type, source, destination, distance and payment via Google pay or Phone pay.





2. My Orders:

My orders shows the list of all orders present to be delivered to the destination which was added by the user by using add courier details.



Backend

MySQL:

MySQL is a free and open-source relational database management system that manages and manipulates data using SQL (Structured Query Language). It is widely used as a backend database in web applications and can handle large amounts of data efficiently. MySQL is well-known for its speed, dependability, and simplicity of use, making it a popular choice for both developers and businesses.

We have to create a new database called courier_delivery in mysql.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| courier_delivery   |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| test               |
+--------------------+
6 rows in set (0.00 sec)
```

Dependencies Used:

We used a lot of dependencies in the backend and we will explain now what they are:

- ***Spring Boot Starter Web:***

  This dependency is used when creating web applications using Spring Boot. It includes the Spring MVC framework, which provides assistance in creating online applications using the Model-View-Controller pattern. The embedded web server provided by the Spring Boot Starter Web dependency, which contains Tomcat, makes it simple to run your application without the need for an external web server. For our project, we use Tomcat.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- ***Spring Boot Starter Data JPA:***

  It is a dependency that provides you with a set of pre-configured libraries for creating JPA-based data access layers in Spring Boot applications. It includes Hibernate ORM technology, which allows you to map Java objects to tables in a relational database. The Spring Boot Starter Data JPA dependency also aids in database migrations, transaction management, and other standard data access tasks. Making this a project dependency allows you to quickly set up a JPA-based data access layer without having to do a lot of setup or write a lot of boilerplate code.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- ***Mysql Connector:***

  It is a JDBC driver that allows Java applications to connect to MySQL databases. Java developers can use JDBC API to connect to MySQL servers and perform database operations.

```xml
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.0.33</version>
    <scope>runtime</scope>
</dependency>
```

- ***Spring Boot Starter Security:***

  It is a dependency that provides a set of pre-configured libraries for using Spring Boot to build secure web applications. It supports authentication, authorization, and other security-related functions.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

JWT (Java JSON Web Tokens) is a dependency that allows you to generate, sign, and validate JSON web tokens. To implement secure authentication and authorization, it is frequently used in conjunction with Spring Boot Starter Security. You can quickly set up a secure web application with minimal configuration by including these dependencies in your project.

```xml
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

- ***Spring Boot Starter Test:***

  It contains all of the necessary libraries, utilities, and configurations for writing unit, integration, and end-to-end tests within a Spring Boot application. This starter enables developers to use popular testing frameworks such as JUnit, TestNG, Mockito, and Spring Test, as well as auto-configurations, to make it easier to set up

and execute tests while ensuring a consistent and effective testing environment for Spring Boot applications.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

- ***For ELK:***

  These dependencies we installed are to initiate logger in controller files so that we can get logs and perform monitoring and visualisations.

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.20.0</version>
</dependency>
```

Functionalities:

The functionalities we have done in our project are listed as follows:

- **Add Courier:**

  Users can add the courier by filling out the respective details it needs to create a courier and store it in the database.

- **Add Agent:**

  Admin is the one that can create an agent and these agents are the ones that take care of couriers and store it in the database.

- **Delete Courier:**

  Admin can delete the courier by clicking the delete icon in the courier of list of couriers and update info in the database.

- **Delete Agent:**

  Admin can delete the agents by clicking the delete icon in the Agent details section and and update info in the database.

- **Update:**

  Admin can update the details of an existing courier in the database.

- **Login and SignUp:**

  A User can login using username and password. Users can register by filling out respective details.

- **Authorization:**

  In our project, two roles are defined: "admin" and "user." Each functionality has an authorization check, which means that only people with specific roles can use it.

Testing using Junit and Mockito:

In the admin controller test file, we tested if the agent is registered successfully or not and agent information is deleting or not. We use registerAgent() and agentDelete() functions from the admin controller. We also tested the courierDelete() function in this same file.

```java
@Test
public void testAgentDeleteSuccess() {
    // Mocking data
    Long agentId = 1L;

    // Mocking behavior
    List<CourierDetails> courierDetailsList = new ArrayList<>();
    when(courierDetailsDAO.findByAgent_Id(agentId)).thenReturn(courierDetailsList);

    // Call the method
    ResponseEntity<BasicDTO<CourierDetails>> responseEntity = adminController.agentDelete(agentId);

    // Assert the result
    assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
    verify(courierDetailsDAO).findByAgent_Id(agentId);
    verify(courierDetailsDAO).saveAll(courierDetailsList);
    verify(userDAO).deleteById(agentId);
}

tabnine: test | explain | document | ask
@Test
public void testAgentDeleteFailureCourierNotFoundException() {
    // Mocking data
    Long agentId = 1L;

    // Mocking behavior
    when(courierDetailsDAO.findByAgent_Id(agentId)).thenReturn(new ArrayList<>());

    // Call the method and assert the expected exception
    try {
        adminController.agentDelete(agentId);
    } catch (CourierNotFoundException e) {
        assertEquals("Courier not found for agent with ID: " + agentId, e.getMessage());
    }
}
```

```java
@Test
public void testCourierDeleteSuccess() {
    // Mocking data
    Long orderId = 1L;

    // Mocking behavior
    when(courierDetailsDAO.findById(orderId)).thenReturn(Optional.of(new CourierDetails()));

    // Call the method
    ResponseEntity<BasicDTO<CourierDetails>> responseEntity = adminController.courierDelete(orderId);

    // Assert the result
    assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
    verify(courierDetailsDAO).deleteById(orderId);
}

tabnine: test | explain | document | ask
@Test
public void testCourierDeleteFailureCourierNotFoundException() {
    // Mocking data
    Long orderId = 1L;

    // Mocking behavior
    when(courierDetailsDAO.findById(orderId)).thenReturn(Optional.empty());

    // Call the method and assert the expected exception
    try {
        adminController.courierDelete(orderId);
    } catch (CourierNotFoundException e) {
        assertEquals("Courier not found", e.getMessage());
    }
}
```

```java
@Test
public void testRegisterAgentSuccess() {
    // Mocking data
    RegisterRequestDTO registerRequestDTO = new RegisterRequestDTO();
    registerRequestDTO.setEmail(email:"test@example.com");
    registerRequestDTO.setPassword(password:"password");
    registerRequestDTO.setConfirmPassword(confirmPassword:"password");

    // Mocking behavior
    when(userDAO.existsByEmail(registerRequestDTO.getEmail())).thenReturn(false);
    when(passwordEncoder.encode(registerRequestDTO.getPassword())).thenReturn("encodedPassword");
    when(userDetailsService.loadUserByUsername(registerRequestDTO.getEmail())).thenReturn(
        Mockito.mock(org.springframework.security.core.userdetails.UserDetails.class));
    when(jwtUtil.generateToken(any())).thenReturn("token");

    // Call the method
    ResponseEntity<?> responseEntity = adminController.registerAgent(registerRequestDTO);

    // Assert the result
    assertEquals(HttpStatus.CREATED, responseEntity.getStatusCode());
}
```

In the user controller test file, we tested if the courier is created or not when a user is adding a courier. We test the createCourier() function from the user controller and test to see if it works or not.

```java
Akhil-Tavva
@BeforeEach
public void setUp() { MockitoAnnotations.openMocks( testClass: this); }

Akhil-Tavva
@Test
public void testCreateCourierSuccess() {
    // Mocking data
    String token = "Bearer mockToken";
    CourierDetails courierDetails = new CourierDetails();
    courierDetails.setId(1L);
    courierDetails.setStatus(CourierStatusEnum.PENDING);

    // Mocking behavior
    when(jwtUtil.getUsernameFromToken(token.replace( target: "Bearer ", replacement: ""))).thenReturn( t: "test@example.com");
    when(userDAO.findUserByEmail("test@example.com")).thenReturn(Optional.of(new User()));
    when(courierDetailsDAO.save(any())).thenReturn(courierDetails);

    // Call the method
    ResponseEntity<BasicDTO<CourierDetails>> responseEntity = userController.createCourier(courierDetails, token);

    // Assert the result
    assertEquals(HttpStatus.CREATED, responseEntity.getStatusCode());
    assertEquals( expected: "Successfully create", responseEntity.getBody().getMessage());
    assertEquals(courierDetails, responseEntity.getBody().getData());
    verify(courierDetailsDAO).save(courierDetails);
}
```

API Documentation

➢ ***Get all couriers:*** http://localhost:9090/admin/agent/register

- HTTP Method: GET
- Request body: None

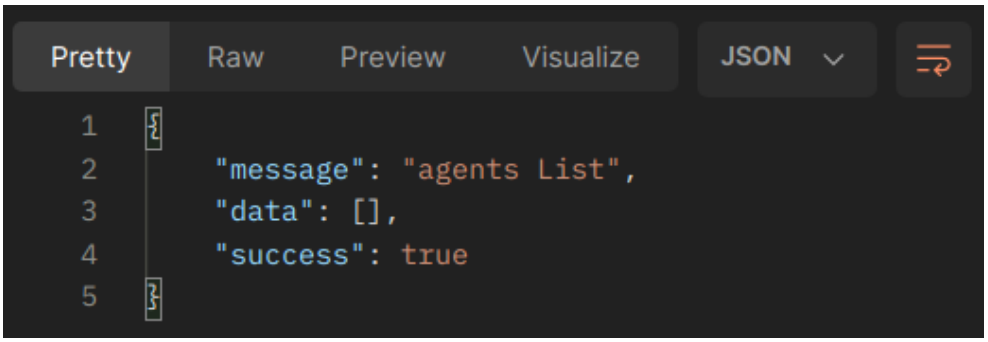
- Response body:


```
        "id": 1,
        "source": "Bangalore",
        "destination": "Delhi",
        "weight": 50.0,
        "distance": 60.0,
        "amount": 285000.0,
        "createdOn": "2023-12-12",
        "expectedDeliveryDate": "2023-12-17",
        "paymentMode": "PhonePay",
        "paymentDetails": "paid",
        "currentLocation": "Hyderabad",
        "status": "PENDING",
        "orderType": "OFFICE"
    },
    {
        "id": 2,
        "source": "Delhi",
        "destination": "Chennai",
        "weight": 50.0,
        "distance": 700.0,
        "amount": 2100000.0,
        "createdOn": "2023-12-12",
        "expectedDeliveryDate": "2023-12-17",
        "paymentMode": "GPay",
        "paymentDetails": "paid",
```


➢ Get all agents:

http://localhost:9090/admin/agent/register

- HTTP Method: GET
- Request body: None
- Response body:



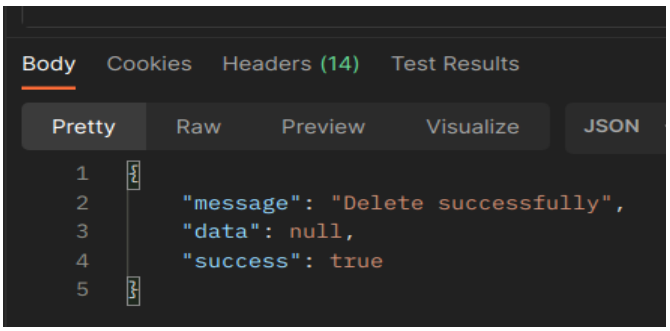
➢ Get all users:

- HTTP Method: GET
- Request body: None
- Response body:

```
{
    "message": "users List",
    "data": [
        {
            "id": 2,
            "firstName": "Aravind",
            "lastName": "Tavva",
            "email": "john@doe.com",
            "mobileNo": "4444444444",
            "role": "USER",
            "createdOn": "2023-12-12T10:27:40.000+00:00",
            "active": true
        }
    ],
    "success": true
}
```


➢ Delete courier:

- HTTP Method: DELETE
- Request body: None
- Response body:



➢ Register New Agent:

http://localhost:9090/admin/agent/register

- HTTP Method: POST
- Request body:


```
{
  "mobileNo": "9876543210",
  "firstName": "Agent",
  "lastName": "Smith",
  "email": "agent.smith@example.com",
  "role": "AGENT",
  "password": "M@nish19",
  "confirmPassword": "M@nish19"
}
```

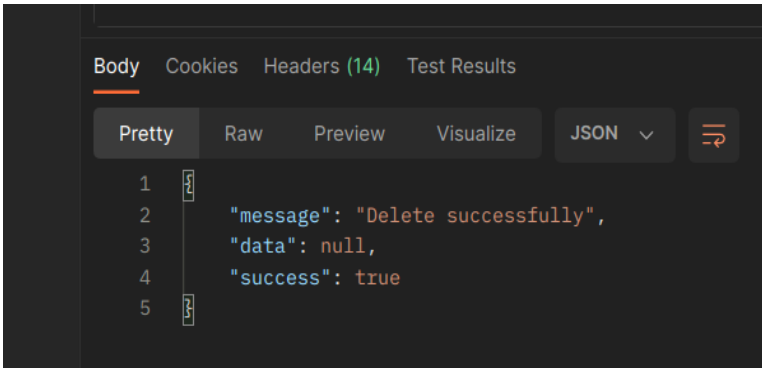
- Response body:


```
{
    "message": null,
    "data": {
        "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZ2VudC5zbW
            9JdRZnCHHH13D260-nz49r2xX7mDyD2132WrUo8EUoTB2Lhqz
        "email": "agent.smith@example.com",
        "name": "Agent"
    },
    "success": true
}
```


➢ Delete Agent:

http://localhost:9090/agent/delete/{agentId}

- HTTP Method: DELETE
- Request body:
- Response body:



➢ Assign Agent:

http://localhost:9090/courier/assignAgent/{agentId}/{orderId}

- HTTP Method: POST
- Request body:
- Response body: