Travel Planning App

CS 816 - Software Production Engineering

International Institute of Information Technology, Bangalore

Under the Guidance of
Prof. B Thangaraju

Team - 77
(MT2023091) Abhishek Sharma
(MT2023172) Somesh Awasthi

Index

Problem Statement

In today's fast-paced world, efficient travel planning is paramount for both leisure and business travelers. Our project focuses on the development of a sophisticated travel planning application tailored to meet the diverse needs of modern travelers. The objective is to create a user-centric platform that simplifies the entire travel planning process, from initial booking of transportation and accommodation to managing tour packages.

To achieve this, we leverage cutting-edge DevOps tools to automate development processes and implement robust monitoring tools for continuous performance optimization. By embracing these methodologies, we ensure rapid and agile development within our designated timeframe, without compromising on quality or reliability.

We aim to build our application into a comprehensive travel companion, integrating features such as creating tour packages, booking packages, and adding agents who will receive user package requests and handle all necessary bookings. By consolidating these diverse services into a single, cohesive platform, we aspire to provide users with a seamless and hassle-free experience, eliminating the need to juggle multiple apps or websites for their travel needs.

Introduction

Travel planning involves multiple components, including transportation, accommodation, and activities, which often require users to navigate several platforms. This fragmentation can lead to increased stress and time consumption. Our project addresses this by developing an integrated application that provides end-to-end travel planning solutions.

Objectives:

1. Destination Selection: Users kick start their journey by specifying their desired travel destination.
2. Attraction Selection: Present users with a curated selection of must-visit attractions, customized based on their trip duration and preferences.
3. Itinerary Planning: Arrange selected attractions into a cohesive itinerary, meticulously organized by day for optimal exploration.

Project Technology Stack:

1. Front End - ReactJS
2. Back End - Spring Boot
3. Database - MySql
4. Version Control System - Git
5. CI/CD – Jenkins
6. Containerisation – Docker
7. Container Orchestration - Docker Compose
8. Configuration Management tool - Ansible
9. Monitoring and Alerting tool - ELK Stack

Important Links:

1. GitHub repository - click here
2. DockerHub Registry Link:

- Frontend Image Repository Link: click here
- Backend Image Repository Link: click here

DevOps tool chain

You can use any set of DevOps tool chains you want, but the pipeline would be the same. The pipeline includes,

1. Source Control Management:
    - Tools: GitHub, GitLab, BitBucket, etc.
    - Purpose: To manage and track changes in the source code, facilitate collaboration, and ensure version control.
2. Testing:
    - Tools: JUnit, Selenium, PyUnit, etc.
    - Purpose: To validate the functionality of individual components and ensure code quality through automated testing.
3. Build:
    - Tools: Maven, Gradle, Ant, etc.
    - Purpose: To automate the process of compiling the code, packaging it into executable formats, and managing dependencies.
4. Continuous Integration:
    - Tools: Jenkins, GitLab CI, Travis CI, etc.
    - Purpose: To automate the integration of code changes from multiple contributors, run automated tests, and build the application continuously.
5. Containerization:
    - Tool: Docker

- Purpose: To create lightweight, portable containers for the application, ensuring consistency across different environments. Push the created Docker image to Docker Hub.

6. Deployment and Configuration Management:
    - Tools: Chef, Puppet, Ansible, Rundeck
    - Purpose: To automate the configuration of environments, deploy Docker containers, and manage application settings. These tools are used to pull Docker images and run them on managed hosts.
7. Container Orchestration:
    - Tools: Docker Compose, Kubernetes, OpenStack
    - Purpose: To manage container deployment, scaling, and operations. Deployment can be done on a local machine or using cloud services such as Amazon AWS, Google Cloud, or other third-party clouds.
8. Monitoring:
    - Tool: ELK Stack (Elasticsearch, Logstash, Kibana)
    - Purpose: To monitor application performance, collect and analyze logs, and visualize data for proactive maintenance and troubleshooting.

DevOps Introduction:

What's DevOps?

DevOps derives its name from the fusion of development (Dev) and operations (Ops).

DevOps integrates development and operations to enhance software delivery efficiency, speed, and security, surpassing traditional methods. It provides a competitive advantage through a swifter and more iterative software development process rooted in Agile principles.

Additionally, DevOps promotes collaboration across the development cycle, leading to improved application flow and value delivery. It symbolizes a cultural transformation within IT, prioritizing incremental development, rapid deployment, and collective

responsibility for business results, drawing from Agile, lean methodologies, and systems theory.

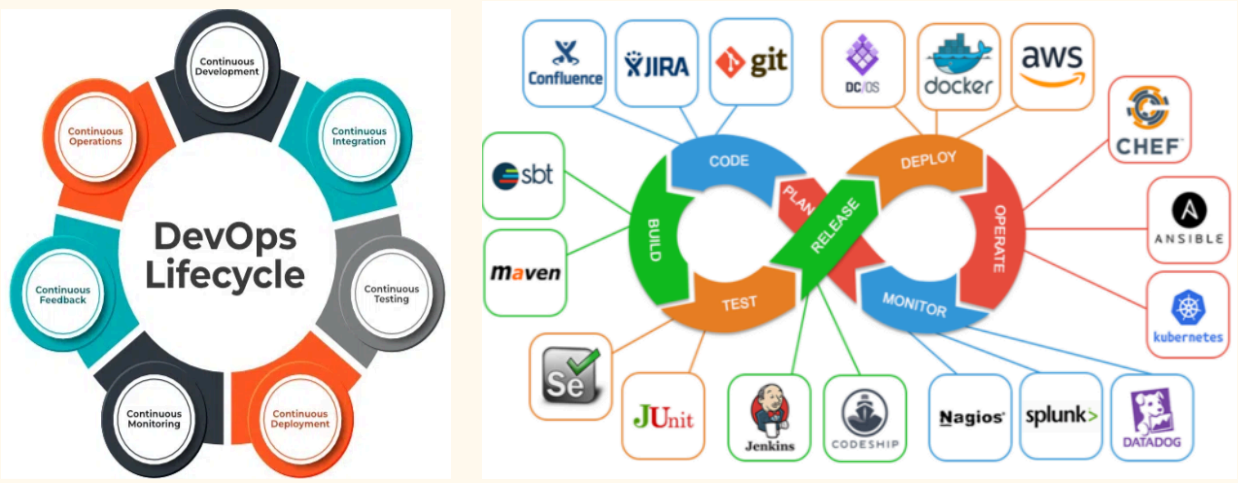


Fig. DevOps Lifecycle Phases

Key Goals and Benefits of DevOps :

Goals of DevOps

1. Ensures effective collaboration between teams
2. Creates scalable infrastructure platforms
3. Builds on-demand release capabilities
4. Provides faster feedback

Benefits of DevOps

1. Smarter work and faster release
2. Quick resolution of issues
3. Better collaboration between teams
4. Fostering innovative mindsets
5. Faster threat detection
6. Increased customer satisfaction

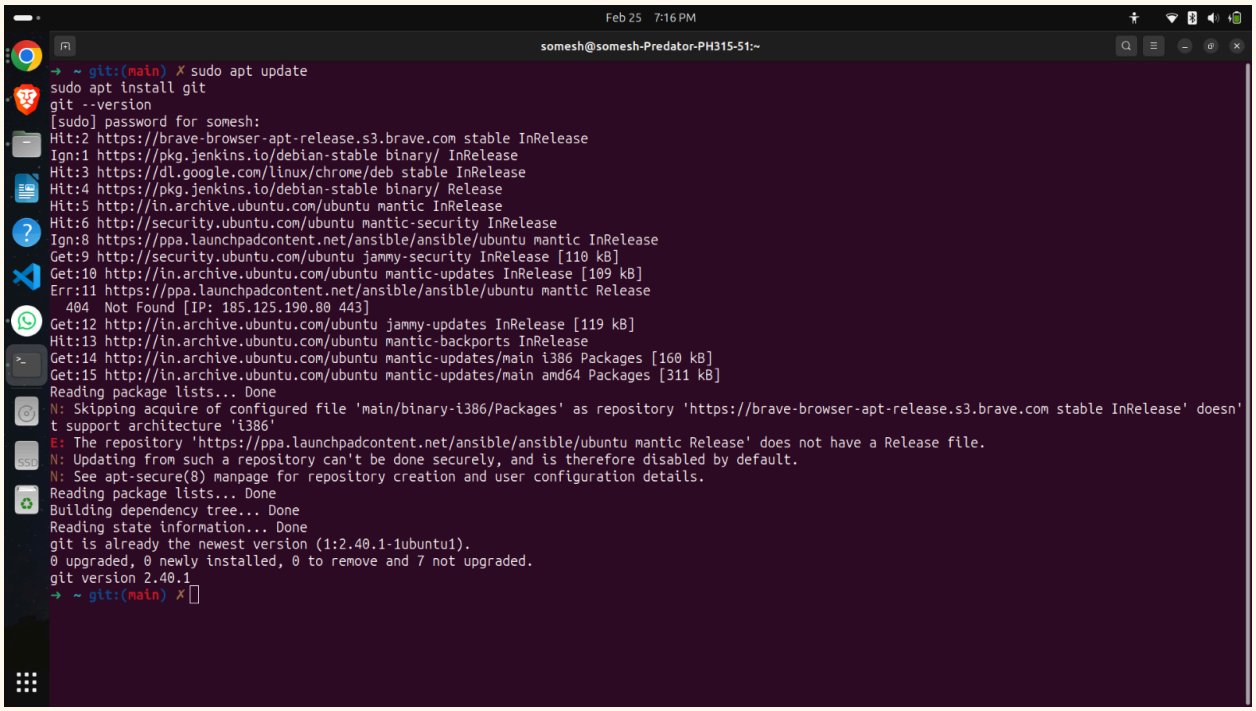
Steps Followed

1. **Pipeline Declaration**
    - Git Installation:
        - Open a terminal window.
        - Update the package index to ensure you have the latest information about available packages by running the following command:
          `sudo apt update`
        - Once the update is complete, install Git by running the following command:

          `sudo apt install git`

        - Once the installation is complete, verify that Git was installed successfully by checking its version. You can do this by running the following command:

          `git –version`



- **Jenkins Installation:**
  - Install Java Development Kit (JDK):

    sudo apt install default-jdk

  - Install Jenkins

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

  - Install ca-certificates

    sudo apt install ca-certificates

  - Install Jenkins

    sudo apt-get update

    sudo apt-get install jenkins

  - To check Jenkins version

    vim /var/lib/jenkins/config.xml

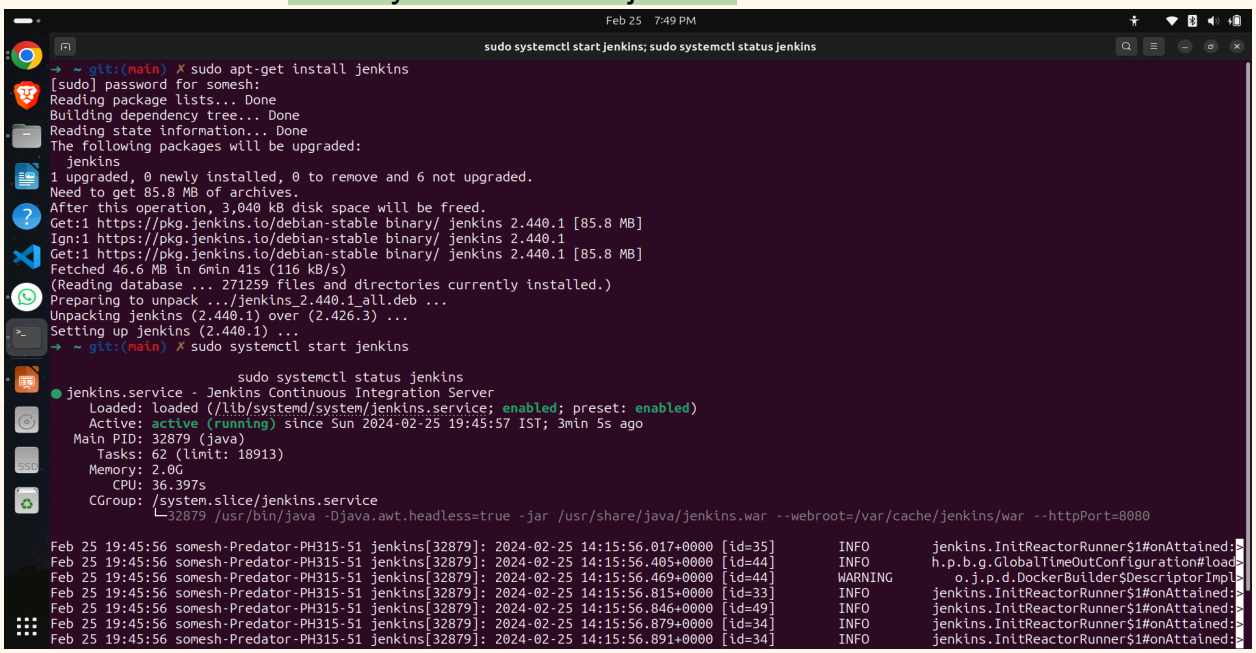
- To copy admin password

  sudo cat /var/lib/jenkins/secrets/initialAdminPassword

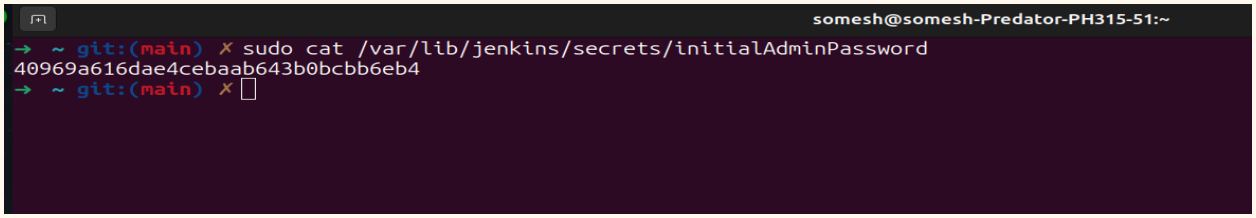
- Once you have installed it, we can start it and check its status by typing the following commands:

  sudo systemctl start jenkins

  sudo systemctl status jenkins

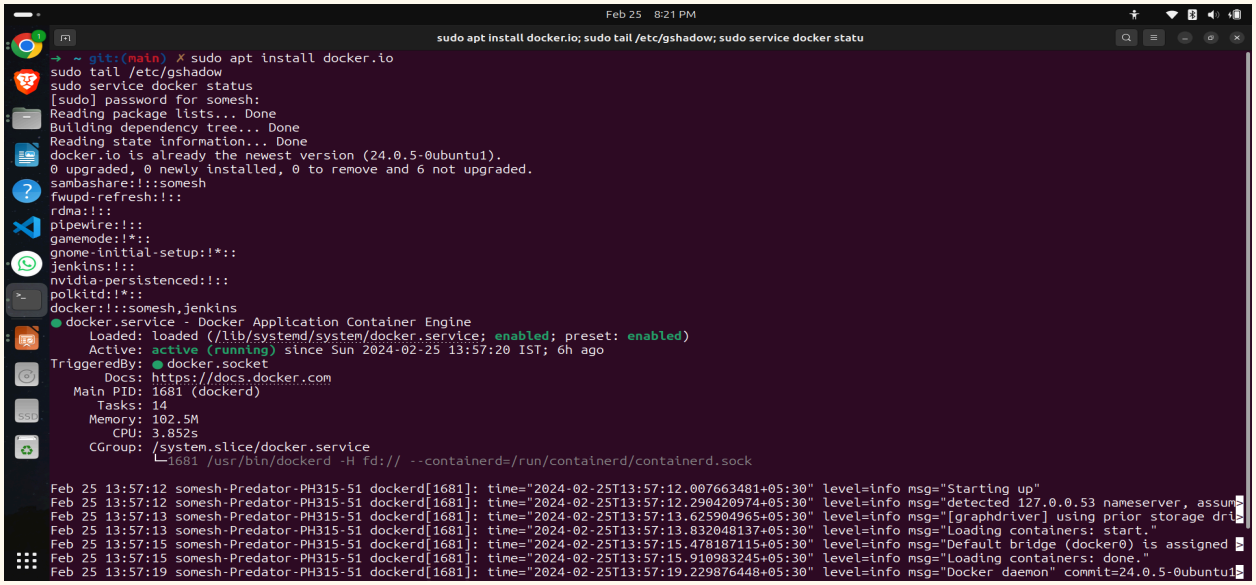


After Jenkins is installed and operational, configuration is necessary. Begin by accessing port 8080 on the localhost. The URL format is: https://localhost:8080. Upon visiting this URL for the first time, Jenkins will prompt for the password, which can be found in the log file.



- ➢ Install Recommended Plugins: After unlocking Jenkins, you'll be prompted to install the recommended plugins or select specific plugins to install.
- ➢ Create Admin User: Once the plugin installation is complete, you'll need to set up an admin user account.
- ➢ Configure Jenkins: After creating the admin user, Jenkins will ask you to set up the URL for accessing Jenkins and optionally install additional plugins.

- **Docker Installation:**
  - Install Docker
    sudo apt install docker.io
  - After Docker has been installed, it's important to ensure that Docker is within the same user group as Jenkins. This can be verified by executing the following command.
  - Check Docker Service Status
    sudo service docker status
  - Pull Docker Images
    docker pull ubuntu
  - List Docker Images
    docker images
  - Explore Docker Hub: Visit Docker Hub (https://hub.docker.com/) to discover and explore available Docker images and repositories



- **Ansible Installation:**
  - Install Ansible
    sudo apt install ansible
  - Check Ansible Version
    ansible --version
  - Verify Installation
    ansible localhost -m ping

- However, a common error that pops up is the following

```
# /opt/freeware/bin/ansible --version
ERROR: Ansible requires the locale encoding to be UTF-8; Detected None.
```

- To address this, we are going to change the locale encoding using the following command.

Type this command

sudo nano /etc/default/locale

It will show

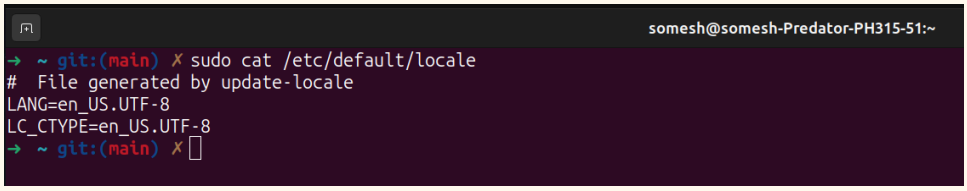
LANG=en_IN

LC_CTYPE=en_IN:en

change it to

LANG=en_US.UTF-8

LC_CTYPE=en_US.UTF-8



DevOps Pipeline Execution Flow

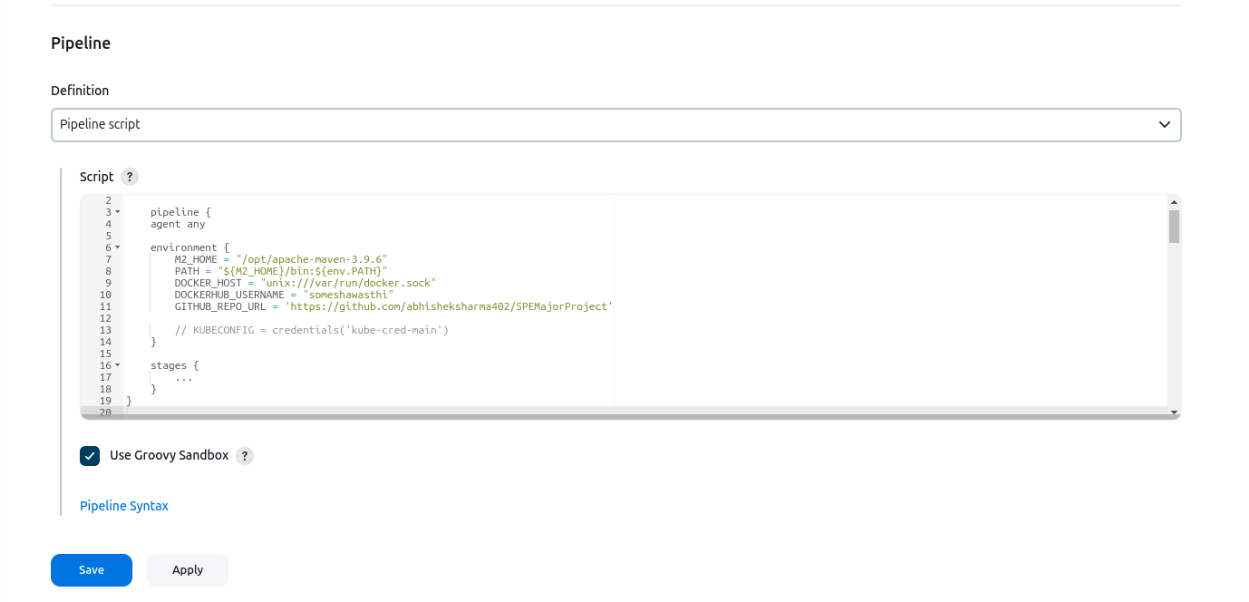
Execution Flow

1. CI/CD Pipeline

Our Jenkins pipeline is designed to automate the entire build, test, and deployment process of our travel planning application. Below is an explanation of each stage in the pipeline, along with the execution flow:

I. Pipeline Declaration

The pipeline is declared using the `pipeline` block, which defines the entire CI/CD process. We use the `agent any` directive to run the pipeline on any available Jenkins agent.



II. Environment Variables

Environment variables are defined to store values that will be used across the pipeline stages, such as Docker Hub username and GitHub repository URL.

```
environment {
    M2_HOME = "/opt/apache-maven-3.9.6"
    PATH = "${M2_HOME}/bin:${env.PATH}"
    DOCKER_HOST = "unix:///var/run/docker.sock"
    DOCKERHUB_USERNAME = "someshawasthi"
    GITHUB_REPO_URL = 'https://github.com/abhisheksharma402/SPEMajorProject'

    // KUBECONFIG = credentials('kube-cred-main')
}
```

III. Stages

The pipeline consists of multiple stages, each representing a step in the CI/CD process.

A. Checkout

In this stage, we clean the workspace and checkout the code from the GitHub repository.

```
19
20
21      stage('Checkout') {
22          steps {
23              script {
24                  cleanWs()
25                  git branch: 'main', url: "${GITHUB_REPO_URL}"
26              }
27          }
28      }
29
```

B. Making Port Available

We stop all running Docker containers, remove them, and delete all Docker images to free up ports and resources.

```
29
30
31
32      stage('Making Port Available') {
33          steps {
34              script {
35                  // Stop all containers
36                  sh 'docker ps -aq | xargs -r docker stop'
37                  sh 'docker ps -aq | xargs -r docker rm'
38                  sh 'docker images -q | xargs -r docker rmi -f'
39                  sh 'docker images'
40              }
41          }
42      }
43
```

C. Maven Build

We build the backend project using Maven, skipping tests for now

```
39
40      stage('Maven Build') {
41          steps {
42              dir('SPE-Project-Backend/main') {
43                  script {
44                      sh 'mvn clean install -DskipTests'
45                  }
46              }
47          }
48      }
49
```

D. Build Docker Images

We build Docker images for the backend and frontend services using the respective Dockerfiles.

```
52
53      stage('Build Docker Images') {
54          steps {
55              script {
56                  sh "docker build -t ${DOCKERHUB_USERNAME}/travelguide-frontend -f Dockerfiles/FrontendDockerfile ."
57                  sh "docker build -t ${DOCKERHUB_USERNAME}/travelguide-backend -f Dockerfiles/BackendDockerfile ."
58                  sh 'docker images'
59              }
60          }
61      }
62
```

E. List Docker Images

We list all Docker images to verify that the images have been built successfully.

```
65
66      stage('List Docker Images') {
67          steps {
68              script {
69                  // Use Docker CLI to list images
70                  sh 'docker images'
71              }
72          }
73      }
74
75
```

F. Push Docker Images to Registry Stage

Docker images are tagged and pushed to Docker Hub in this stage.

```
78
79
80      stage('Push Docker Images to Registry') {
81          steps {
82              script {
83                  // Tag and push backend image
84                  sh 'docker image tag someshawasthi/travelguide-backend:latest ${DOCKERHUB_USERNAME}/travelguide-backend:version1.0'
85                  docker.withRegistry('', 'someshawasthi-dockerhub') {
86                      sh "docker push ${DOCKERHUB_USERNAME}/travelguide-backend:version1.0"
87                  }
88
89                  // Tag and push frontend image
90                  sh 'docker image tag someshawasthi/travelguide-frontend:latest ${DOCKERHUB_USERNAME}/travelguide-frontend:version1.0'
91                  docker.withRegistry('', 'someshawasthi-dockerhub') {
92                      sh "docker push ${DOCKERHUB_USERNAME}/travelguide-frontend:version1.0"
93                  }
94              }
95          }
96      }
97
```

G. Run Ansible Inventory and Playbook Stage

This stage runs the Ansible playbook to deploy the application using the inventory file provided.

```
100
101     stage('Run Ansible Inventory and Playbook') {
102         steps {
103             script {
104                 ansiblePlaybook (
105                     playbook: 'playbook.yml',
106                     inventory: 'inventory.txt',
107                     extras: '-K'
108                 )
109             }
110         }
111     }
112
```

H. Testing Frontend Stage

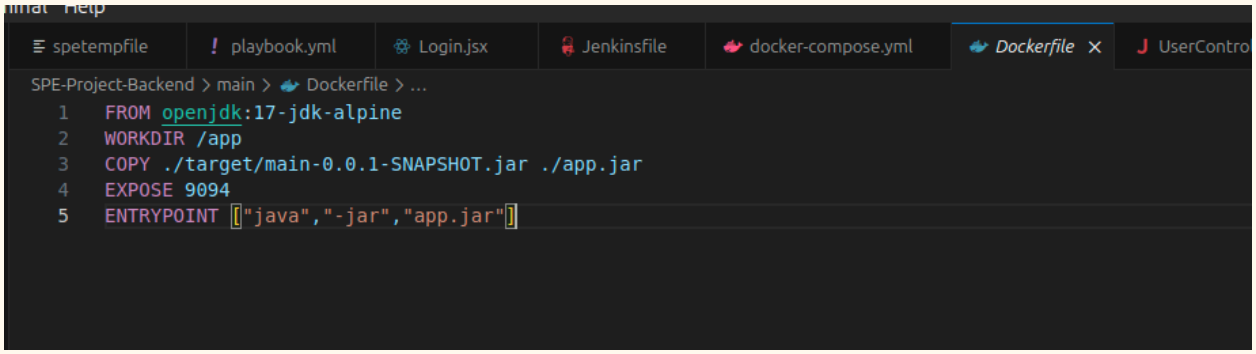
In the final stage, the frontend application is tested using npm and Jest.

```
112
113     stage("Testing Frontend"){
114         steps {
115             dir('SPE-Project-Frontend/SPE-Major-Project/SPEMajorProject'){
116                 script{
117                     sh 'npm install --save-dev jest-environment-jsdom --force'
118                     sh 'npm install --save-dev jest@latest ts-jest@latest --force'
119                     // sh 'npm install -g ts-jest --force'
120                     sh 'npm run test:login'
121                 }
122             }
123         }
124     }
```

2. Containerization

In our travel planning application, we employ containerization to ensure consistent and reliable deployment across various environments. Containerization encapsulates the application and its dependencies into a container, enabling smooth transitions from development to production without compatibility issues. We utilize Docker to containerize both the backend and frontend and database components of our application. Below is a detailed explanation of the Dockerfiles used for each component.

a. SPE-Project-Backend
   i. For the backend component, we use a Dockerfile that sets up an environment for running a Java Spring Boot application. Here is the Dockerfile for the backend:



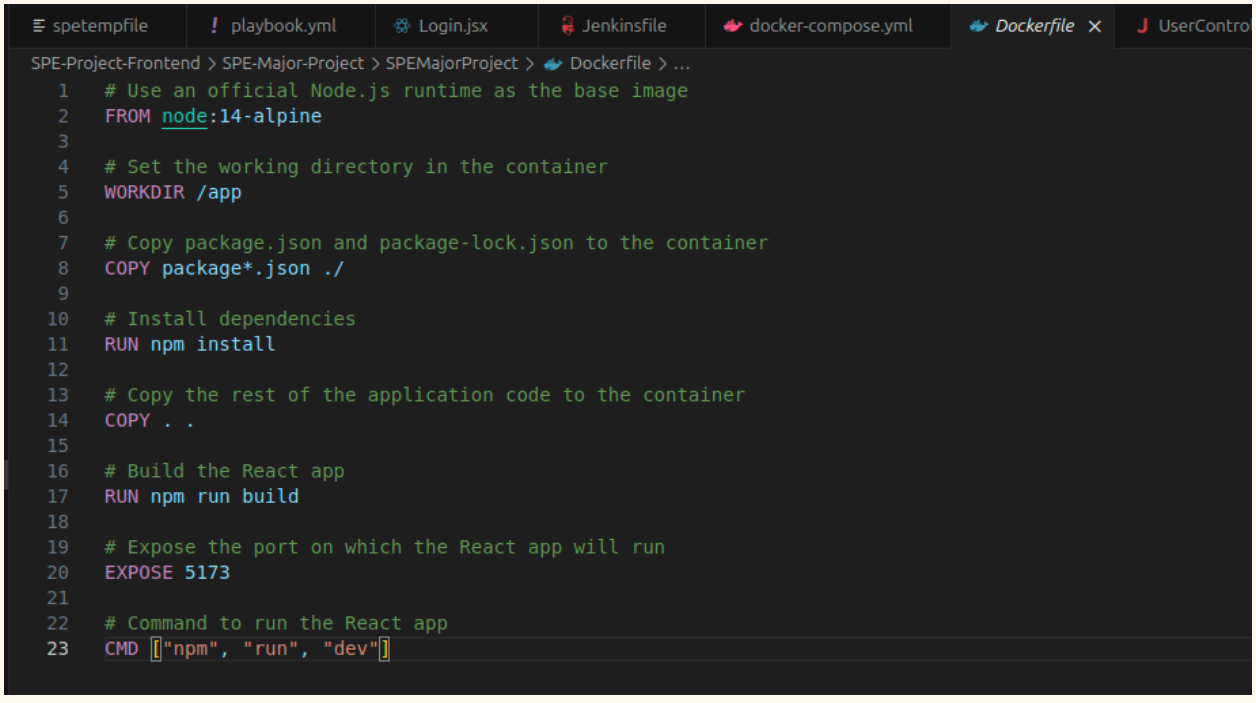
Explanation:

- FROM openjdk:17-jdk-alpine: This line specifies the base image, which is a lightweight Alpine Linux distribution with OpenJDK 17. This choice ensures a minimal footprint and a secure environment for running Java applications.
- WORKDIR /app: Sets the working directory inside the container to /app. All subsequent instructions will be executed in this directory.

- COPY ./target/main-0.0.1-SNAPSHOT.jar ./app.jar: Copies the built JAR file from the host machine into the container. This JAR file contains the compiled Spring Boot application.
- EXPOSE 9094: Exposes port 9094 to the host, allowing external access to the application running inside the container. This is the port on which the backend service will be accessible.
- ENTRYPOINT ["java","-jar","app.jar"]: Defines the command to run when the container starts. It uses the java -jar command to execute the JAR file.

b. SPE-Project-Frontend

i. For the frontend component, we use a Dockerfile that sets up an environment for running a React application. Here is the Dockerfile for the frontend:



Explanation:

- FROM node:14-alpine: Specifies the base image, which is a lightweight Alpine Linux distribution with Node.js 14. This choice ensures a minimal and efficient environment for running Node.js applications.

- WORKDIR /app: Sets the working directory inside the container to /app. This is where the application files will be stored and executed.
- “COPY package.json .*/”: Copies package.json and package-lock.json files to the working directory. These files contain the metadata and dependencies for the Node.js application.
- RUN npm install: Installs the Node.js dependencies defined in the package.json file. This ensures that all necessary libraries and modules are available in the container.
- COPY . .: Copies the entire application code from the host machine to the container. This includes all source files and assets needed to build and run the application.
- RUN npm run build: Builds the React application. This step compiles the React code and optimizes it for production.
- EXPOSE 5173: Exposes port 5173 to the host, allowing external access to the React application running inside the container. This is the port on which the frontend service will be accessible.
- CMD ["npm", "run", "dev"]: Defines the command to run when the container starts. It uses the npm run dev command to start the React development server.

3. Ansible Playbook

The deployment of our travel planning application leverages Ansible to ensure an automated, consistent, and efficient deployment process. Ansible is a powerful automation tool that allows us to manage and configure systems, deploy applications, and orchestrate advanced workflows. By using Ansible, we can streamline our deployment process and minimize manual intervention, thereby reducing the risk of human errors. Below is the Ansible playbook used for deploying our Dockerized application.

Ansible Playbook for Deployment

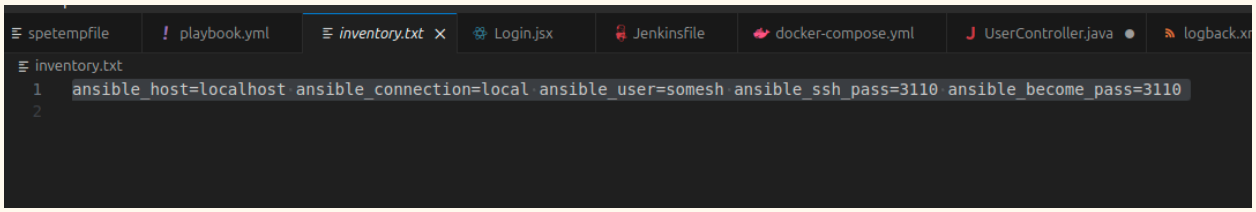
The following Ansible playbook defines the tasks required to deploy the application using Docker Compose.

Explanation:

- name: Run Docker Compose: This is the name of the playbook, providing a brief description of its purpose, which is to run Docker Compose for our application.
- hosts: localhost: Specifies that the playbook will be run on the local machine. This is suitable for our development and initial deployment purposes.
- remote_user: somesh: Defines the user under which the playbook will be executed. In this case, it is the user somesh.
- connection: local: Indicates that the playbook will use a local connection method, meaning it will run on the local machine rather than connecting to a remote host.
- gather_facts: yes: Enables fact gathering, which collects information about the system (e.g., OS details, network interfaces). This can be useful for conditional tasks and configurations within the playbook.

Ansible Inventory

Before diving into the playbook, we need to set up an Ansible inventory file. This file defines the hosts and groups of hosts that Ansible will manage. For our purposes, we are targeting the local machine for deployment.

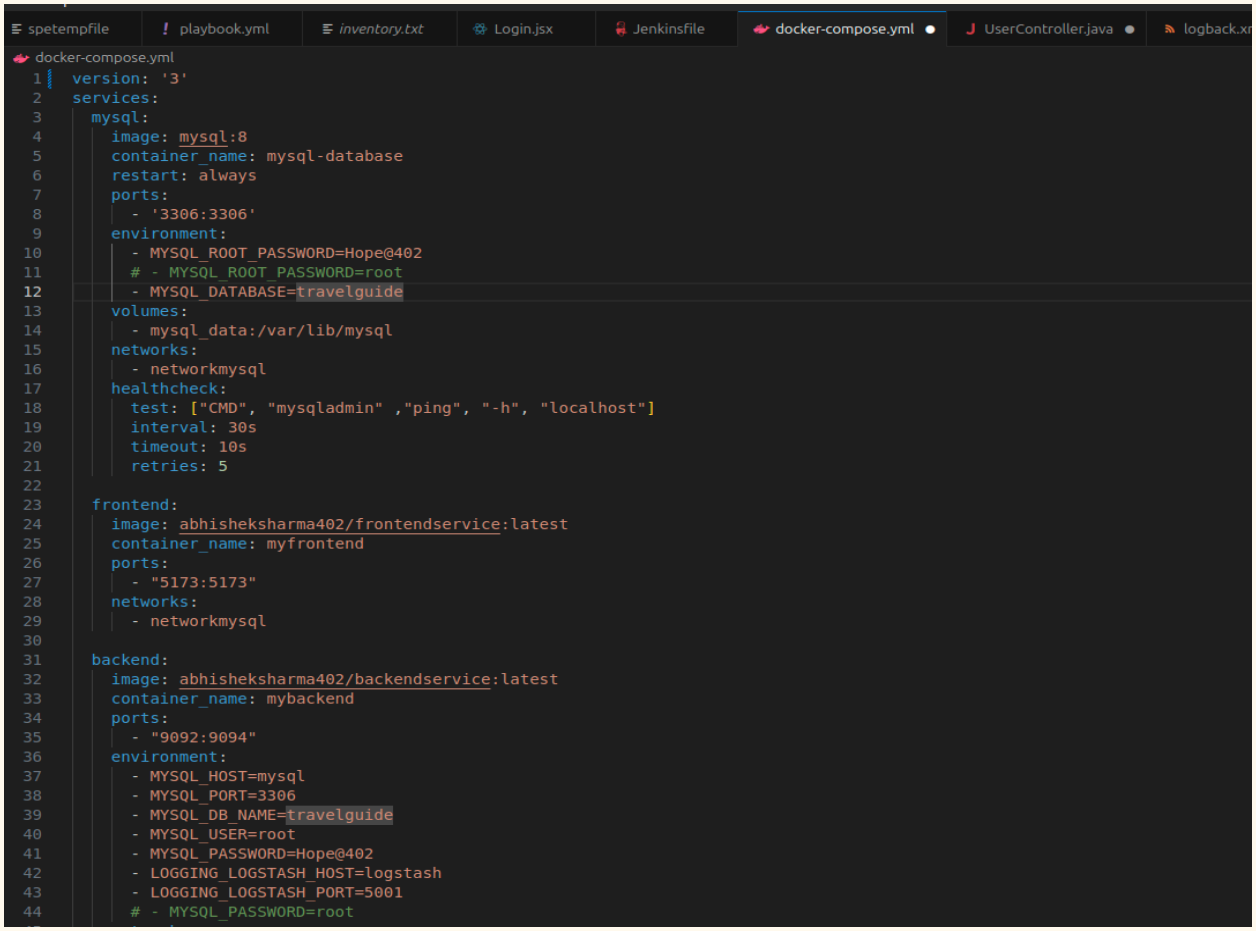


4. Container Orchestration

Container orchestration is a crucial aspect of deploying and managing containerized applications in a production environment. It involves automating the deployment, scaling, and management of containerized applications. In our project, we use Docker Compose for orchestrating the services that make up our travel planning application. This section details the Docker Compose setup used to manage the various containers involved in our project.

Docker Compose File

The docker-compose.yml file is used to define and run multi-container Docker applications. Below is the configuration file used for orchestrating our application's services.


spetempfile
playbook.yml
inventory.txt
Login.jsx
Jenkinsfile
docker-compose.yml
UserController.java
logback.xml
docker-compose.yml
version: '3'
services:
mysql:
image: mysql:8
container_name: mysql-database
restart: always
ports:
- '3306:3306'
environment:
- MYSQL_ROOT_PASSWORD=Hope@402
- MYSQL_ROOT_PASSWORD=root
- MYSQL_DATABASE=travelguide
volumes:
- mysql_data:/var/lib/mysql
networks:
- networkmysql
healthcheck:
test: ["CMD", "mysqladmin" ,"ping", "-h", "localhost"]
interval: 30s
timeout: 10s
retries: 5
frontend:
image: abhisheksharma402/frontendservice:latest
container_name: myfrontend
ports:
- "5173:5173"
networks:
- networkmysql
backend:
image: abhisheksharma402/backendservice:latest
container_name: mybackend
ports:
- "9092:9094"
environment:
- MYSQL_HOST=mysql
- MYSQL_PORT=3306
- MYSQL_DB_NAME=travelguide
- MYSQL_USER=root
- MYSQL_PASSWORD=Hope@402
- LOGGING_LOGSTASH_HOST=logstash
- LOGGING_LOGSTASH_PORT=5001
- MYSQL_PASSWORD=root

```
  backend:
    image: abhisheksharma402/backendservice:latest
    container_name: mybackend
    ports:
      - "9092:9094"
    environment:
      - MYSQL_HOST=mysql
      - MYSQL_PORT=3306
      - MYSQL_DB_NAME=travelguide
      - MYSQL_USER=root
      - MYSQL_PASSWORD=Hope@402
      - LOGGING_LOGSTASH_HOST=logstash
      - LOGGING_LOGSTASH_PORT=5001
      # - MYSQL_PASSWORD=root
    networks:
      - networkmysql
    depends_on:
      - mysql
    restart: on-failure:3
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9092/actuator/health"]
      interval: 30s
      timeout: 10s
      retries: 3

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.3.3
    container_name: elastic
    environment:
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xms512m"
      - "discovery.type=single-node"
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
    # volumes:
    #   - elastic_data:/usr/share/elasticsearch/data
    networks:
      - networkmysql

```

```
  kibana:
    image: docker.elastic.co/kibana/kibana:8.3.3
    container_name: kibana
    ports:
      - "5601:5601"
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    depends_on:
      - elasticsearch
    networks:
      - networkmysql


  logstash:
    image: docker.elastic.co/logstash/logstash:8.3.3
    container_name: logstash
    volumes:
      - /home/somesh/iiitb files/SPE/SPEMajorProject/logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml
      - /home/somesh/iiitb files/SPE/SPEMajorProject/logstash/pipeline:/usr/share/logstash/pipeline:ro
    ports:
      - "5001:5001/tcp"
      - "5001:5001/udp"
      - "9600:9600"
    environment:
      - LS_JAVA_OPTS=-Xmx512m -Xms512m
    networks:
      - networkmysql
    depends_on:
      - elasticsearch

networks:
  networkmysql:
volumes:
  mysql_data:
```

Explanation:

- MySQL Database:
  - Image: Uses the official MySQL 8 image.
  - Container Name: Named mysql-database.
  - Restart Policy: Set to always restart the container if it stops.
  - Ports: Exposes port 3306 for database access.
  - Environment Variables: Sets the root password and initializes the travelguide database.
  - Volumes: Persists data using a named volume mysql_data.
  - Networks: Connects to the networkmysql network.
  - Health Check: Ensures the MySQL service is running and responsive.
- Frontend Service:
  - Image: Uses the latest image of the frontend service.
  - Container Name: Named myfrontend.
  - Ports: Exposes port 5173 for the React application.
  - Networks: Connects to the networkmysql network.
- Backend Service:
  - Image: Uses the latest image of the backend service.
  - Container Name: Named mybackend.
  - Ports: Exposes port 9094 for the backend service.

    - Environment Variables: Configures database connection and logging settings.
    - Networks: Connects to the networkmysql network.
    - Depends On: Ensures the MySQL service is available before starting.
    - Restart Policy: Restarts the container on failure, up to three times.
    - Health Check: Ensures the backend service is running and responsive.
- Elasticsearch:
    - Image: Uses the Elasticsearch 8.3.3 image.
    - Container Name: Named elastic.
    - Environment Variables: Configures memory and node settings.
    - Ports: Exposes port 9200 for Elasticsearch.
    - Networks: Connects to the networkmysql network.
- Kibana:
    - Image: Uses the Kibana 8.3.3 image.
    - Container Name: Named kibana.
    - Ports: Exposes port 5601 for Kibana.
    - Environment Variables: Connects Kibana to Elasticsearch.
    - Depends On: Ensures Elasticsearch is available before starting.
    - Networks: Connects to the networkmysql network.

- Logstash:
    - Image: Uses the Logstash 8.3.3 image.
    - Container Name: Named logstash.
    - Volumes: Mounts local configuration and pipeline files.
    - Ports: Exposes ports 5001 (TCP/UDP) and 9600 for Logstash.
    - Environment Variables: Configures memory settings for Logstash.
    - Networks: Connects to the networkmysql network.
    - Depends On: Ensures Elasticsearch is available before starting.

Networks and Volumes

- networks:
    - networkmysql: Defines a custom bridge network for inter-service communication.
- volumes:
    - mysql_data: Defines a named volume for persisting MySQL data.

5. Unit Testing

Unit testing ensures that individual components of the application function correctly. This section describes the unit testing setup for both the frontend (React) and backend (Spring Boot) components of the travel planning application.

Frontend Unit Testing

The frontend unit tests focus on the Login component using React Testing Library and Jest. The tests verify that the component renders correctly, responds to user interactions, and handles login errors appropriately.

Key Tests:

1. Render Login Component: Ensures that the login component and its initial elements are rendered correctly.

```
it('renders the Login component with initial elements', () => {
    render(
        <BrowserRouter>
            <Login />
        </BrowserRouter>
    );

    // Check if the buttons for roles are rendered
    expect(screen.getByText('Customer')).toBeInTheDocument();
    expect(screen.getByText('Agency')).toBeInTheDocument();
    expect(screen.getByText('Agent')).toBeInTheDocument();

    // Check if the input fields are rendered
    // expect(screen.getByPlaceholderText('Username')).toBeInTheDocument();
    // expect(screen.getByPlaceholderText('Password')).toBeInTheDocument();

    // Check if the Sign in button is rendered
    expect(screen.getByText('Sign in')).toBeInTheDocument();
});
```

2. Role Button Color Change: Tests the role selection button color change upon clicking

```
test('changes role button color on click', () => {
    render(
        <BrowserRouter>
            <Login />
        </BrowserRouter>
    );

    const customerButton = screen.getByText('Customer');
    const agencyButton = screen.getByText('Agency');
    const agentButton = screen.getByText('Agent');

    // Click on the 'Agency' button
    fireEvent.click(agencyButton);

    // Check if the 'Agency' button has the active background color
    expect(agencyButton).toHaveStyle('background-color: #4FA786');

    // Check if the other buttons have the default background color
    expect(customerButton).toHaveStyle('background-color: #efefef');
    expect(agentButton).toHaveStyle('background-color: #efefef');
});
```

3. Login Failure Handling: Mocks a failed login attempt and verifies that the error message is displayed.

```
test('shows error message on login failure', async () => {
    axios.post.mockRejectedValueOnce(new Error('Login failed'));

    render(
        <BrowserRouter>
            <Login />
        </BrowserRouter>
    );

    // Select role
    fireEvent.click(screen.getByText('Agency'));

    // Enter email and password
    fireEvent.change(screen.getByTestId('signin-username'), { target: { name: 'email', value: 'sd@xyz.com' } });
    fireEvent.change(screen.getByTestId('signin-password'), { target: { name: 'password', value: '1234' } });

    // Submit form
    fireEvent.click(screen.getByTestId('signin-btn'));

    await waitFor(() => {
        expect(axios.post).toHaveBeenCalledWith(
            'http://localhost:9094/agency/login',
            { email: 'sd@xyz.com', password: '1234' },
            expect.any(Object)
        );
    });

    // Check if the error message is displayed
    await waitFor(() => {
        expect(screen.getByText('Error while Signing Up')).toBeInTheDocument();
    });
});
```

Backend Unit Testing

The backend unit tests focus on the UserController using JUnit and Spring Boot Test. These tests check the functionality of user registration, authentication, and other user-related operations.

Key Tests:

1. User Registration: Tests user registration by sending a POST request with user details and verifying the response status.

```
@Test  abhisheksharma402
@WithMockUser
void register() throws Exception{
    RegistrationRequest registrationRequest = new RegistrationRequest();
    registrationRequest.setEmail("test@example.com");
    registrationRequest.setFirstName("test");
    registrationRequest.setLastName("example");
    registrationRequest.setPassword("1234");

    UserEntity userEntity = new UserEntity();
    userEntity.setEmail("test@example.com");
    userEntity.setFirstName("test");
    userEntity.setLastName("example");

    when(userService.register(registrationRequest)).thenReturn(userEntity);

    mockMvc.perform(post( urlTemplate: "/user/register")
                    .with(SecurityMockMvcRequestPostProcessors.csrf()) // Add CSRF token
                    .contentType(MediaType.APPLICATION_JSON)
                    .content("{\"firstName\":\"test\", \"lastName\": \"example\", \"password\":\"1234\",\"email\":\"test@example.com\"}"))
            .andExpect(status().isAccepted());
}
```

2. User Registration: Tests user registration by sending a POST request with user details and verifying the response status.

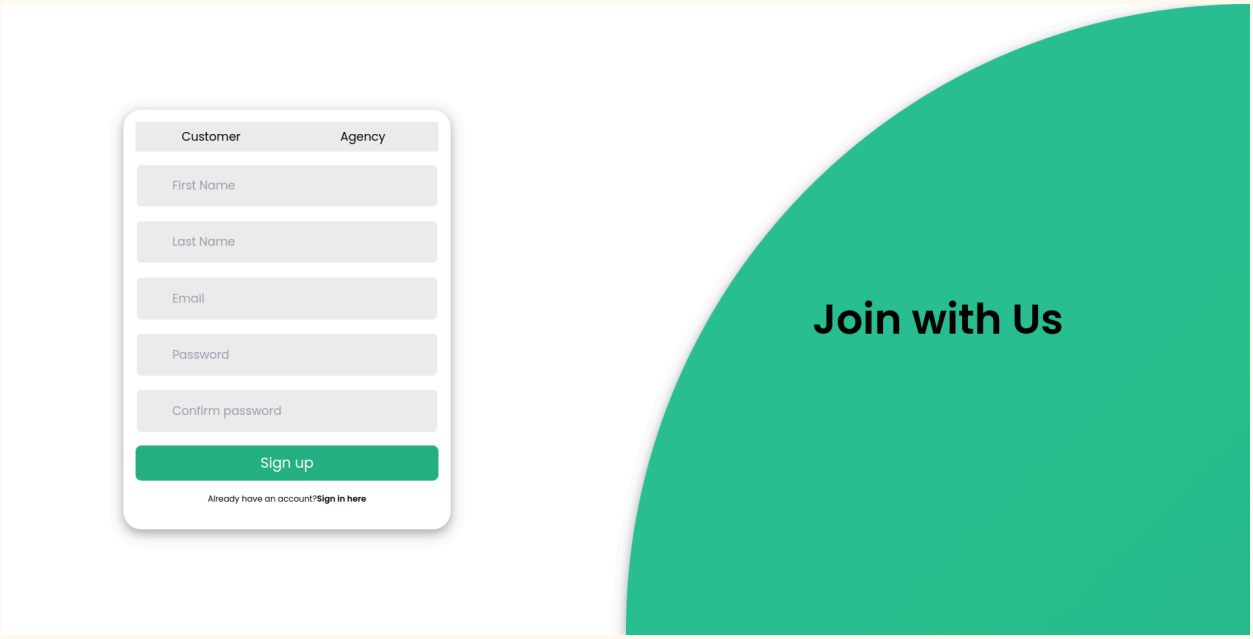
```
@Test  abhisheksharma402
@WithMockUser
void authenticate() throws Exception {

    AuthenticationRequest authenticationRequest = new AuthenticationRequest();
    AuthenticationResponse authenticationResponse = new AuthenticationResponse();
    when(userService.authenticate(any(AuthenticationRequest.class))).thenReturn(authenticationResponse);

    mockMvc.perform(post( urlTemplate: "/user/login")
                    .with(SecurityMockMvcRequestPostProcessors.csrf()) // Add CSRF token
                    .contentType(MediaType.APPLICATION_JSON)
                    .content("{\"username\":\"test@example.com\",\"password\":\"password\"}"))
            .andExpect(status().isOk());

}
```

3. User Registration: Tests user registration by sending a POST request with user details and verifying the response status.

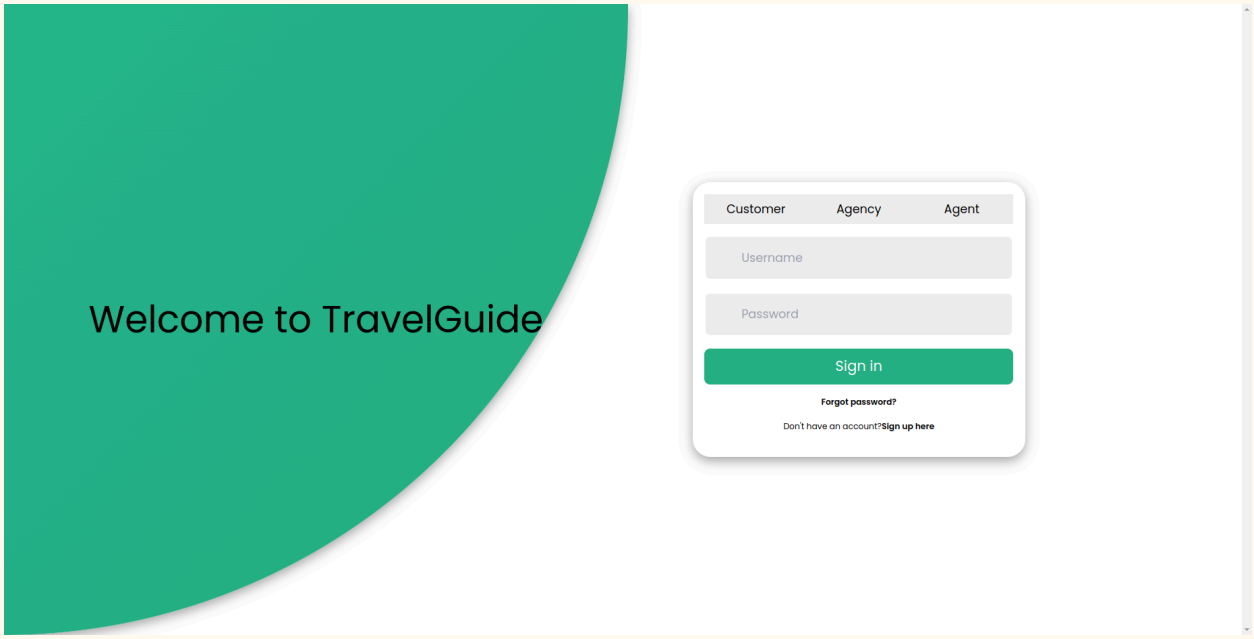
```
94      @Test  abhisheksharma402
95      @WithMockUser
96      void getPackages() throws Exception {
97          PackageEntity packageEntity = new PackageEntity();
98          packageEntity.setId(30);
99          packageEntity.setName("Test Package");
100         packageEntity.setDestination("port blair");
101
102         when(packageRepository.findByDestination(any(String.class))).thenReturn(List.of(packageEntity));
103
104         mockMvc.perform(get( urlTemplate: "/user/packages")
105                         .with(SecurityMockMvcRequestPostProcessors.csrf())
106                         .param( name: "destination",  ...values: "port blair"))
107                 .andExpect(status().isOk())
108                 .andExpect(jsonPath( expression: "$[0].name").value( expectedValue: "Test Package"));
109     }
110 }
```

Application Screenshots

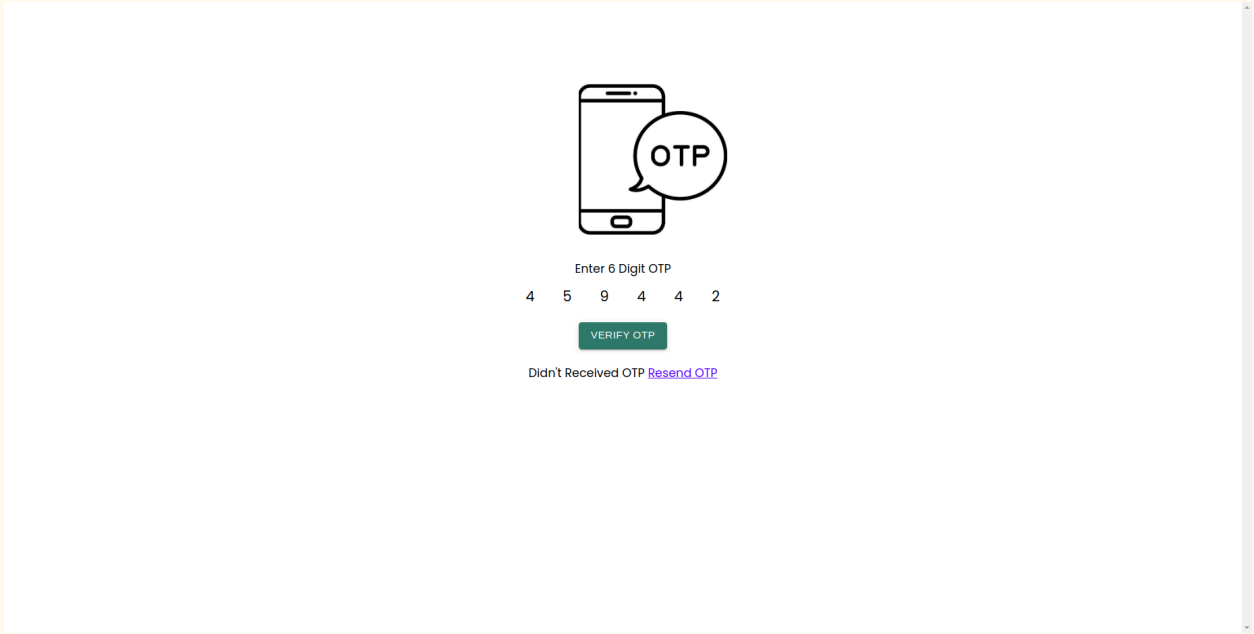
Register Component:



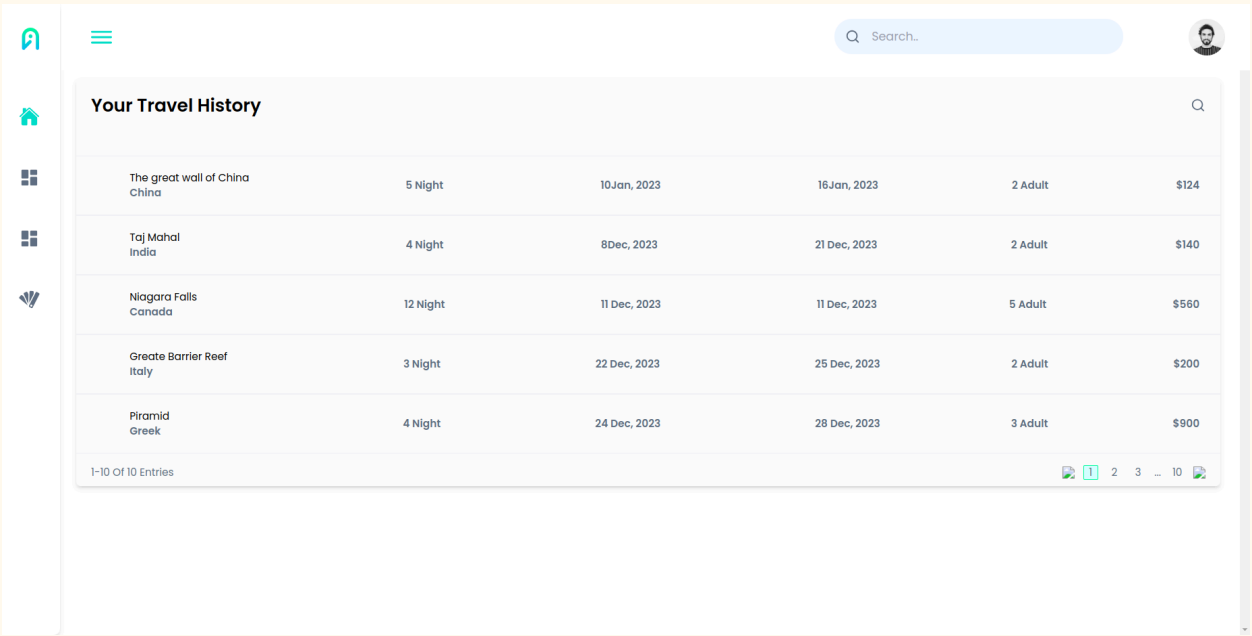
Login Component



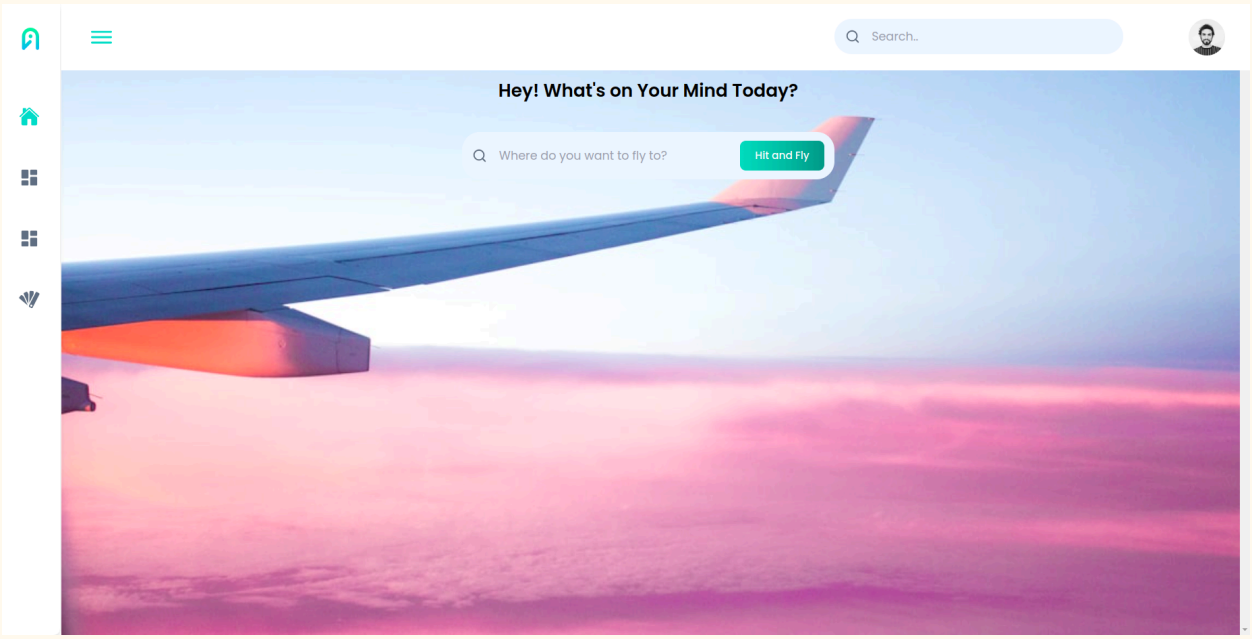
Activate Account Component



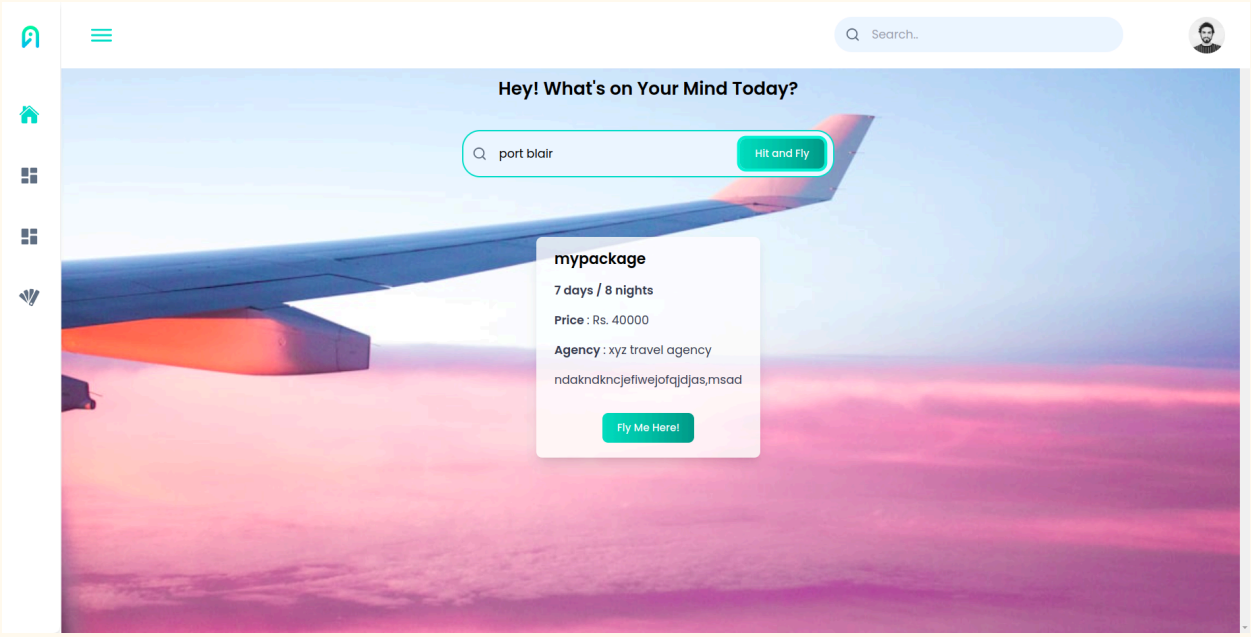
Customer Dashboard



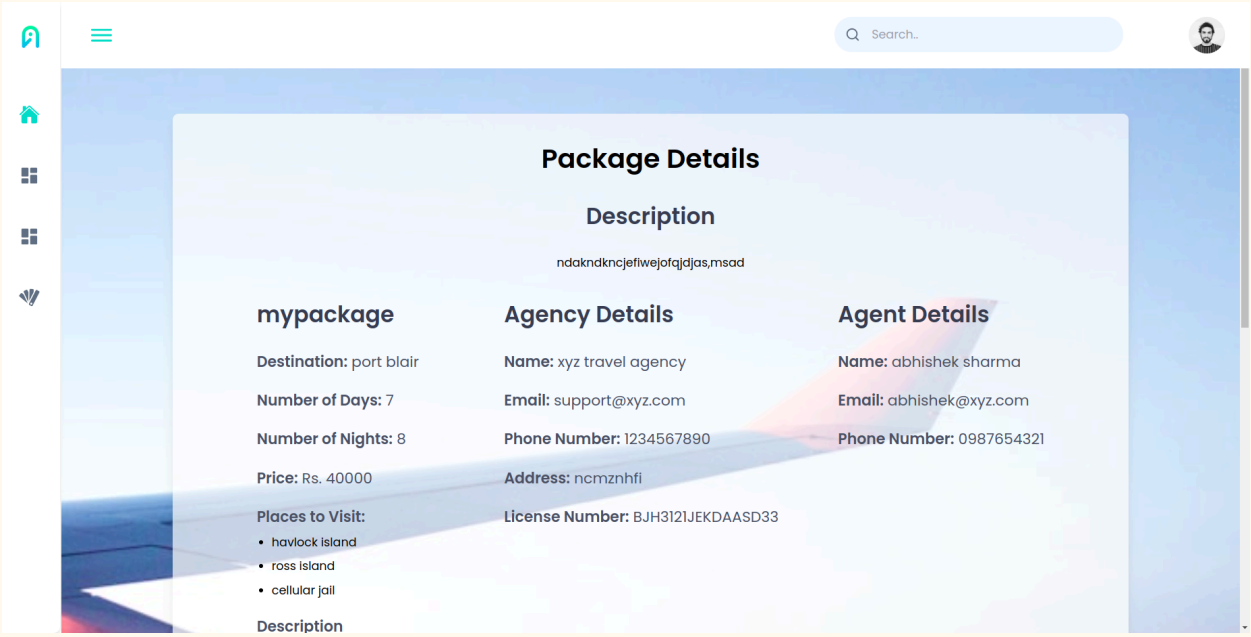
Plan a Trip Component

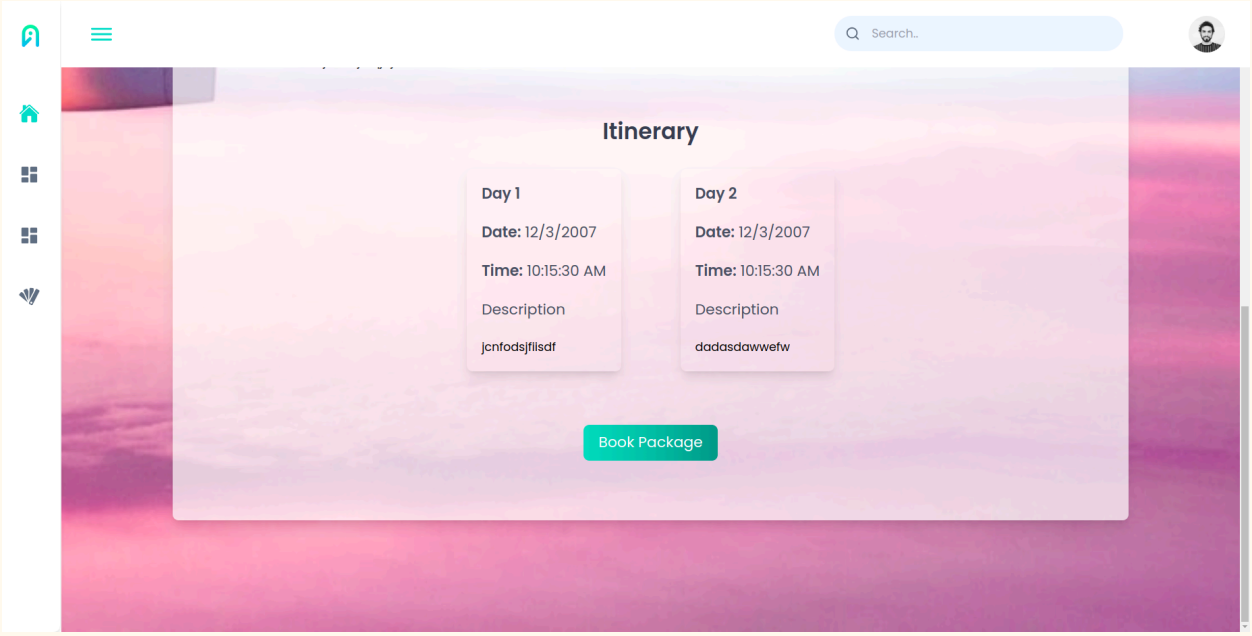


Displaying all the packages with the destination search query



Individual Package Details





Project Demo & GitHub Links

GitHub Link:

- Link: click here

DockerHub Registry Link:

- Frontend Image Repository Link: click here
- Backend Image Repository Link: click here