UniGo

# Test Documentation

## Test Plan

The test plan provides an overview of the testing approach to be followed for the UniGo project. It outlines the types of testing to be conducted, the testing tools to be used, the resources required, and the timelines for testing.

### Testing Framework and Setup

Testing framework used for the UniGo project is Jest. The tests are organized in a "describe" block, with "it" statements containing the individual test cases. The test cases use the request() method from the supertest library to send requests to the app's routes. To set up the testing environment, a beforeAll() block is used to handle user authentication and retrieve tokens required for testing protected routes. An afterAll() block is used to clean up the database and close the connection after all tests have been executed. The test cases are organized based on the route they are testing, with separate describe blocks for each route (e.g., Signup Route, Signin Route, etc.). Each test case is designed to cover specific functionality, with a combination of unit and integration tests to ensure proper functioning of the web app. To run the tests, execute the following command in the terminal: npm test. This will run all the test cases specified in the test file, and display the results in the terminal.

**Objectives The primary objectives of the test plan are as follows:**

- To ensure the functional requirements of the UniGo application are met
- To verify the reliability, performance, and usability of the application

**To ensure that the application complies with the specified quality standards and requirements.**

**The scope of the test plan includes the following:**

- Testing of the UniGo application across various devices and platforms
- Testing of the UniGo application for both functional and non-functional requirements
- Testing of the UniGo application for integration with Google Maps API
- Testing of the UniGo application for integration with Node.js and MongoDB.

**The testing strategy for UniGo is as follows:**

**Unit Testing:** Unit testing will be conducted to verify the functionality of individual components of the UniGo application.

**Integration Testing:** Integration testing will be conducted to verify the integration of individual components of the UniGo application with each other and with external systems such as Google Maps API, Node.js and MongoDB.

**System Testing:** System testing will be conducted to verify the functionality of the entire UniGo application as a whole.

**Test Environment The test environment for UniGo will include the following:**

- A variety of devices including smartphones, tablets, and laptops.
  Different operating systems including iOS, Android, and Windows.
- Google Maps API
- Node.js
- MongoDB.

**Test Plan Execution The test plan will be executed as follows:**

- Unit testing will be conducted by developers during the development phase of the application.
- Integration testing will be conducted by developers during the integration phase of the application.
- System testing will be conducted once the application is integrated and deployed.

**The following test deliverables will be produced as part of the UniGo testing process:**

- Test Plan
- Test Cases
- Test Scripts
- Test Reports
- Defect Reports.

The UniGo project requires a comprehensive testing plan to ensure that it meets the functional and non-functional requirements. This document has provided an overview of the testing approach to be followed, the testing tools and resources required. The test plan will be executed by developers to identify and report defects, and to ensure that the UniGo application meets the required quality standards. The test cases cover different aspects of the app, including user authentication, ride management, and error handling. The following sections provide detailed information on each test case, including the expected input and output.

## Backend testing

### Authentication

### Signup Route

**Unit Test:** Verify that the signup route returns a 400 status when provided with invalid user data.

**Integration Test:** Verify that the signup route returns a 200 status when provided with valid user data.

### Signin Route

**Unit Test:** Verify that the signin route returns a 400 status when provided with invalid user data.

**Integration Test:** Verify that the signin route returns a 200 status when provided with valid user data.

```
//Unit Test SignIn
describe(signin, () => {
  it("post" + signin, async () => {
    const response = await request(app).post(signin).send({
      email: "Ad4rwere4@gmil.com",
      password: "rahul1",
    });
    expect(response.status).toEqual(401);
  });
});
// Integration test SignIn
describe(signin, () => {
  it("post" + signin, async () => {
    const response = await request(app).post(signin).send({
      email: "Ad4rwere4@gmil.com",
      password: "rahul123",
    });
    expect(response.status).toEqual(200);
  });
});
```

### Signout Route

**Unit Test:** Verify that the signout route returns a 400 status when called without proper authentication.

```
describe(signout, () => {
  it("get" + signout, async () => {
    const response = await request(app).get(signout);
    expect(response.status).toEqual(400);
  });
});
```

**Users Route**

**Unit Test:** Verify that the users route returns a 200 status when called to fetch all user data.

**Ride Management**

Trip History Route

**Unit Test:** Verify that the trip history route returns a 200 status when called with proper authentication.

Drive Route (Additional test cases need to be created for the Drive Route)

Simar has been done to each component to ensure the functionality of the code.

## Frontend Testing

React component rendering test

- Use the React Testing Library to test if the main App component renders without crashing.

```
import { render, screen } from '@testing-library/react';
import App from './App';

it("renders without crashing", () => {
  render(<App />);
});
```

Test if the application renders with the App component and root div

- Use Jest to mock ReactDOM and its render method to assert that render is called with **<App />** and the HTML element with the id 'root'.

```
import React from "react"
import ReactDOM from "react-dom"
import App from "./App"

jest.mock("react-dom", () => ({ render: jest.fn() }))

test("renders with App and root div", () => {
  const root = document.createElement("div")
  root.id = "root"
  document.body.appendChild(root)

  require("./index.js")
})
```

**Trip history test**

- Use Cypress to test the trip history functionality by logging in and visiting the trip history page.

```
it.only('trip history test', function() {
  cy.visit("/login")

  cy.get('[data-test="email-form-control"]').type("user1@gmail.com")
  cy.get('[data-test="password-form-control"]').type("user1")
  cy.get('[data-test="login-button"]').click()
  cy.wait(3000)
})
```