University of Dhaka

Department of Computer Science and Engineering

CSE-3112 : Software Engineering Lab

Software Testing Document (STD)

GitGrub

A Canteen Management Web App

Submitted By:

Meherun Farzana, Roll: 05

Mehrajul Abadin Miraj, Roll: 20

Aniket Joarder, Roll: 48

Submitted On :

May 15, 2024

Submitted To :

Dr. Saifuddin Md. Tareeq

Redwan Ahmed Rizvee

Contents

Chapter 1

Unit Testing

1.1 Static Testing

1.1.1 Walk through

Forms

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

models

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

urls

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

views

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was undocumented empty block in this class. All the blocks were appropriately not commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

1.1.2 Code Review

Code Review - Jamal Uddin Mallick

GitGrub showcases a sophisticated and well-structured codebase, embodying modern software engineering principles and technologies. Its adoption of the Model-View-Controller (MVC) architecture, complemented by MongoDB for models, React with TypeScript for views, and Node.js for controllers, establishes a clear separation of concerns, facilitating maintainability and scalability. This architecture ensures that data manipulation, presentation, and business logic are distinct, promoting code organization and readability.

One notable aspect is the effective utilization of services throughout the application. These services encapsulate complex business logic, fostering modularity and code reusability. The User Management Service handles user authentication and authorization, ensuring secure access to the application. The Canteen Management Service streamlines operations for canteen owners, facilitating menu and inventory management. The Order Management Service orchestrates seamless ordering experiences, while the Employee Management Service enables efficient staff management. Additionally, the Sales and Analytics Service provides valuable insights into canteen performance.

The choice of React with TypeScript for the frontend offers a powerful combination of component-based architecture and static typing, enhancing code quality and developer productivity. On the backend, Node.js harnesses its non-blocking, event-driven nature to deliver efficient server-side processing. MongoDB serves as the database, offering flexibility and scalability to accommodate diverse data requirements.

Testing practices within the project exhibit a commitment to quality assurance. Unit testing ensures the reliability of individual components, while integration testing validates interactions between subsystems, ensuring robust system behavior. Method stubs and mock objects are utilized to simulate interactions and dependencies, enabling comprehensive testing coverage.

Overall, GitGrub exemplifies a comprehensive understanding of software engineering principles. Its well-structured architecture, effective use of services, modern technology stack, and rigorous testing practices collectively contribute to a scalable, maintainable, and robust solution, poised to meet the demands of managing canteen operations effectively.

Code Review - Md. Mahmudul Hasan

The GitGrub Canteen Management Web Application is a shining example of meticulous software engineering, blending a meticulously crafted codebase with a contemporary technological stack. Embracing the Model-View-Controller (MVC) architecture, GitGrub elegantly segments responsibilities, with MongoDB models orchestrating data interactions, React with TypeScript sculpting dynamic user interfaces, and Node.js controllers navigating business logic.

At the core of the application's architecture lies its adept utilization of services, each encapsulating intricate functionalities and fostering code reusability. The User Management Service ensures fortified access control through robust authentication mechanisms, while the Canteen Management Service equips proprietors with streamlined menu and inventory oversight. The Order Management Service orchestrates seamless transactions, while the Employee Management Service simplifies staffing logistics. Additionally, the Sales and Analytics Service furnishes invaluable insights into business performance, enabling data-driven decision-making.

The chosen technology stack, comprising React with TypeScript and Node.js with MongoDB, underscores GitGrub's commitment to modernity and scalability. React's component-based architecture empowers the creation of modular, reusable UI components, while TypeScript augments code quality with its static typing prowess. On the backend, Node.js facilitates high-performance server-side operations, seamlessly integrating with MongoDB to handle data storage and retrieval efficiently.

GitGrub's testing practices are robust, with a keen focus on ensuring reliability and correctness. Unit testing meticulously scrutinizes individual components, while integration testing validates interactions between modules, ensuring seamless system behavior. Leveraging method stubs and mock objects, the testing suite comprehensively evaluates the application's functionality, instilling confidence in its stability and resilience.

In summation, the GitGrub Canteen Management Web Application epitomizes excellence in software engineering, with its well-structured architecture, efficient service utilization, modern technology stack, and rigorous testing practices. By adhering to best practices and harnessing cutting-edge technologies, GitGrub delivers a scalable, maintainable, and feature-rich solution, poised to revolutionize canteen management operations with efficiency and effectiveness.

1.2 Dynamic Testing

1.2.1 Black box Testing

1.2.1.1 Range Partitioning

1.2.1.1.1 Sign Up Page

Test Case Identifier	1.2.1.1.1.1
Statement of Purpose	To test the register screen with an invalid password
Description of Preconditions	The user is not logged in.
Test Case Inputs	A password is less than 6 characters long and doesn't meet the strength requirements of case and numbers
Expected Outputs	The password field of the form will be marked red, with the error message shown.
Description of Expected Post-conditions	The error message "Enter a valid password" will appear below the email field. The field will be marked in red.
Execution History	The error message appears below the red-marked email field.

Test Case Identifier	1.2.1.1.1.2
Statement of Purpose	To test the register screen with correct values of email and password
Description of Preconditions	The user is not logged in.
Test Case Inputs	A valid email address and password
Expected Outputs	The form will show no errors.
Description of Expected Post-conditions	There will be no error messages or red-marked fields, allowing the user to proceed with the registration request.
Execution History	No error messages appeared below the fields.

1.2.1.1.2 Log In Page

Test Case Identifier	1.2.1.1.2.1
Statement of Purpose	To test the login screen with an empty email address
Description of Preconditions	The user is not logged in
Test Case Inputs	An invalid email address and a password
Expected Outputs	The email field of the form will be marked red, with the error message shown
Description of Expected Post-conditions	The error message "Email is required" will appear below the email field. The field will be marked in red
Execution History	The error message appears below the red-marked email field

Test Case Identifier	1.2.1.1.2.2
Statement of Purpose	To test the login screen with correct values of email and password
Description of Preconditions	The user is not logged in
Test Case Inputs	A valid email address and password
Expected Outputs	The form will show no errors
Description of Expected Post-conditions	There will be no error messages or red-marked fields, allowing the user to proceed with the login request
Execution History	No error messages appeared below the fields

1.2.1.1.3 Home Page

Test Case Identifier	1.2.1.1.3.1
Statement of Purpose	To test the canteen search functionality with inputs of range
Description of Preconditions	The user is on the home page
Test Case Inputs	
Expected Outputs	The system will display an error message indicating invalid input format
Description of Expected Post-conditions	The canteen search results will not be displayed, and the user will be prompted to correct the input errors
Execution History	After entering the invalid format inputs and clicking on the "Search" button, the system displayed an error message highlighting the invalid fields

1.2.1.1.4 Ordering Page

Test Case Identifier	1.2.1.1.4.1
Statement of Purpose	To test the ordering process when a selected menu is out of stock
Description of Preconditions	The user has chosen a canteen within the DU area
Test Case Inputs	Selected Menu: Chicken Fry in amount 3 pieces
Expected Outputs	The system will display an error message indicating that the selected menu is not available in required amount
Description of Expected Post-conditions	The menu selection page will be refreshed, allowing the user to choose another available menu
Execution History	After selecting menu Chicken Fry and clicking the "Add to cart" button, the system displayed an error message stating that the menu is not available for ordering

1.2.1.1.5 Employee Management Page

Test Case Identifier	1.2.1.1.5.1
Statement of Purpose	To test adding an employee with an invalid role
Description of Preconditions	The user has to login as a manager
Test Case Inputs	Role: "Waiter"
Expected Outputs	The system will display an error message indicating that no such role is available for an employee
Description of Expected Post-conditions	The employee information form will remain visible, allowing the user to enter other valid required information
Execution History	After leaving the employee role field empty and clicking the "Submit" button, the system displayed an error message stating that the employee information is incomplete

1.2.1.1.6 Inventory Management Page

Test Case Identifier	1.2.1.1.6.1
Statement of Purpose	To test the adding ingredients with invalid amount
Description of Preconditions	The user has to login as a manager
Test Case Inputs	Amount: -10 kg
Expected Outputs	System will show an error message indicating to input a valid amount
Description of Expected Post-conditions	The system will not add the ingredient
Execution History	After entering amount as "-10 kg", the system displayed an error message stating that the valid amount is required

1.2.1.1.7 Sales Record

Test Case Identifier	1.2.1.1.7.1
Statement of Purpose	To verify that sales records are correctly generated and stored.
Description of Preconditions	The system is operational and has the necessary dependencies such as a database connection.
Test Case Inputs Amount: 50.00 Timestamp: 2024-05-15 10:30:00	Order ID: 12345
Expected Outputs	Sales record entry is created in the database. The sales record entry includes the provided order ID, amount, and timestamp.
Description of Expected Post-conditions	The sales record is successfully stored in the database and can be retrieved for reporting and analysis purposes.
Execution History	The test case passes if the assertions are successful, indicating that the sales record functionality is working as expected.

1.2.1.2 Set Partitioning

1.2.1.2.1 Sign Up Page

Test Case Identifier	1.2.1.2.1.1
Statement of Purpose	To test the register screen with an invalid password
Description of Preconditions	The user is not logged in.
Test Case Inputs	'123'
Expected Outputs	The password field of the form will be marked red, with the error message shown.
Description of Expected Post-conditions	The error message "Enter a valid password" will appear below the email field. The field will be marked in red.
Execution History	The error message appears below the red-marked email field.

Test Case Identifier	1.2.1.2.1.2
Statement of Purpose	To test the register screen with correct values of email and password
Description of Preconditions	The user is not logged in.
Test Case Inputs	'abc@gmail.com' and '123'
Expected Outputs	The form will show no errors.
Description of Expected Post-conditions	There will be no error messages or red-marked fields, allowing the user to proceed with the registration request.
Execution History	No error messages appeared below the fields.

1.2.1.2.2 Log In Page

Test Case Identifier	1.2.1.2.2.1
Statement of Purpose	To test the login screen with an empty email address
Description of Preconditions	The user is not logged in
Test Case Inputs	
Expected Outputs	The email field of the form will be marked red, with the error message shown
Description of Expected Post-conditions	The error message "Email is required" will appear below the email field. The field will be marked in red
Execution History	The error message appears below the red-marked email field

Test Case Identifier	1.2.1.2.2.2
Statement of Purpose	To test the login screen with correct values of email and password
Description of Preconditions	The user is not logged in
Test Case Inputs	'meherun2001@gmai.com' and 'Meherun123'
Expected Outputs	The form will show no errors
Description of Expected Post-conditions	There will be no error messages or red-marked fields, allowing the user to proceed with the login request
Execution History	No error messages appeared below the fields

1.2.1.2.3 Home Page

Test Case Identifier	1.2.1.2.3.1
Statement of Purpose	To test the canteen search functionality with inputs of range
Description of Preconditions	The user is on the home page
Test Case Inputs	'DU'
Expected Outputs	The system will display an error message indicating invalid input format
Description of Expected Post-conditions	The canteen search results will not be displayed, and the user will be prompted to correct the input errors
Execution History	After entering the invalid format inputs and clicking on the "Search" button, the system displayed an error message highlighting the invalid fields

1.2.1.2.4 Ordering Page

Test Case Identifier	1.2.1.2.4.1
Statement of Purpose	To test the ordering process when a selected menu is out of stock
Description of Preconditions	The user has chosen a canteen within the DU area
Test Case Inputs	'Chicken Fry'
Expected Outputs	The system will display an error message indicating that the selected menu is not available in required amount
Description of Expected Post-conditions	The menu selection page will be refreshed, allowing the user to choose another available menu
Execution History	After selecting menu Chicken Fry and clicking the "Add to cart" button, the system displayed an error message stating that the menu is not available for ordering

1.2.1.2.5 Employee Management Page

Test Case Identifier	1.2.1.2.5.1
Statement of Purpose	To test adding an employee with an invalid role
Description of Preconditions	The user has to login as a manager
Test Case Inputs	'Waiter'
Expected Outputs	The system will display an error message indicating that no such role is available for an employee
Description of Expected Post-conditions	The employee information form will remain visible, allowing the user to enter other valid required information
Execution History	After leaving the employee role field empty and clicking the "Submit" button, the system displayed an error message stating that the employee information is incomplete

1.2.1.2.6 Inventory Management Page

Test Case Identifier	1.2.1.2.6.1
Statement of Purpose	To test the adding ingredients with invalid amount
Description of Preconditions	The user has to login as a manager
Test Case Inputs	-1
Expected Outputs	System will show an error message indicating to input a valid amount
Description of Expected Post-conditions	The system will not add the ingredient
Execution History	After entering the amount as "-10 kg", the system displayed an error message stating that the valid amount is required

1.2.1.2.7 Sales Record

Test Case Identifier	1.2.1.2.7.1
Statement of Purpose	To verify that sales records are correctly generated and stored.
Description of Preconditions	The system is operational and has the necessary dependencies such as a database connection.
Test Case Inputs	Order ID: 12345 Amount: 50.00
Expected Outputs	Sales record entry is created in the database. The sales record entry includes the provided order ID, amount, and timestamp.
Description of Expected Post-conditions	The sales record is successfully stored in the database and can be retrieved for reporting and analysis purposes.
Execution History	The test case passes if the assertions are successful, indicating that the sales record functionality is working as expected.

1.2.2 White box Testing

Statement/Code Coverage

Statement 1: Sign up as a new user

Code snippet

```
      describe('Admin creates new user', () => {
    it('should create new user when valid data is provided', async () => {
        // Mock admin login
        const adminCredentials = { username: 'admin', password: 'adminpass' };
        const adminToken = await generateAdminToken(adminCredentials);

        // Mock request to create new user
        const userData = {
            username: 'newuser',
            password: 'newpass',
            email: 'newuser@example.com',
            role: 'user' // Assuming admin assigns a role
        };
        const response = await request(app)
            .post('/admin/users')
            .set('Authorization', `Bearer ${adminToken}`)
            .send(userData);

        // Assertions
        expect(response.status).toBe(201);
        expect(response.body.status).toBe('success');
        expect(await User.countDocuments()).toBe(2); // Assuming User model is used
    });
});
```

Flow chart

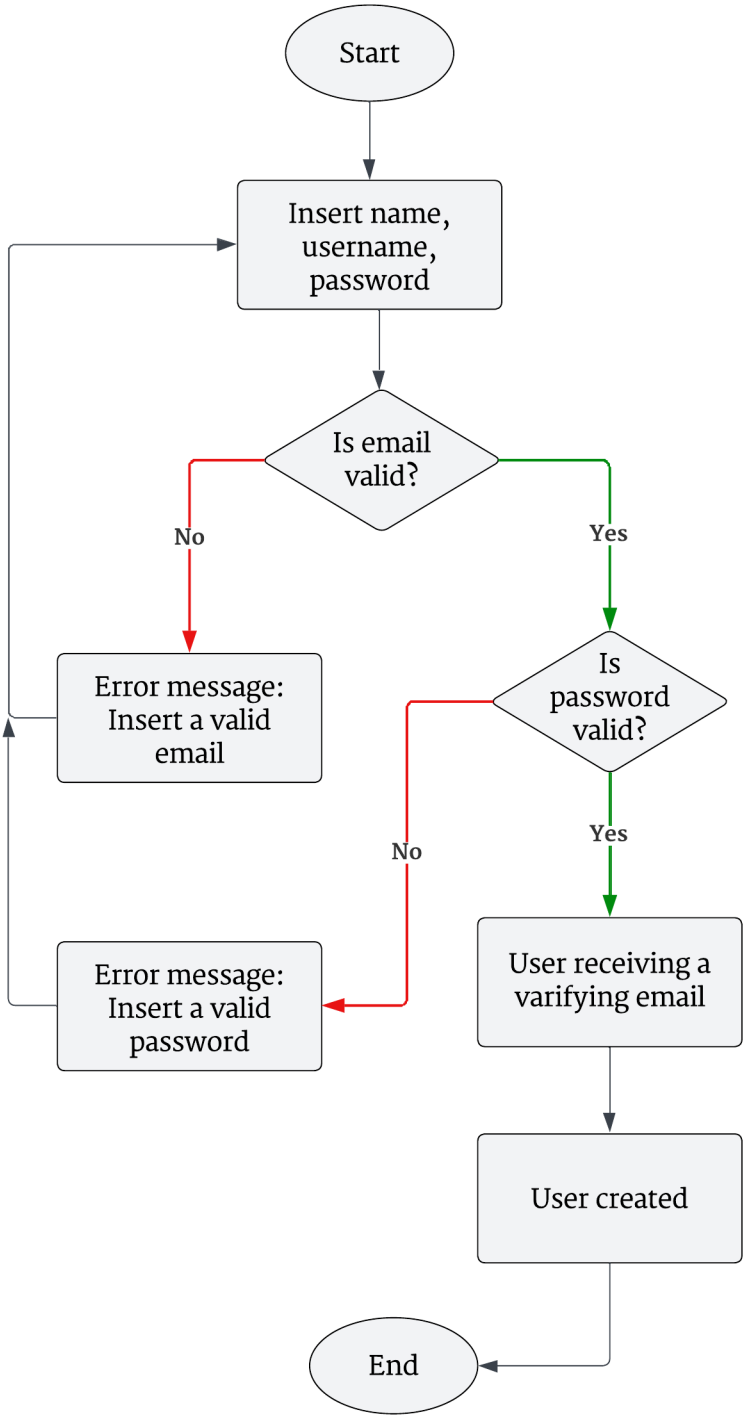


Figure 1.1: Statement 1

Test Case 1.1: is email valid = False
Output: Error Message
Statement coverage: $5/5 * 100 = 100\%$

Test Case 1.2: is email valid = True and and password valid = False
Output: Error message
Statement coverage: $4/5 * 100 = 80\%$ **Test Case 1.3**: is email valid = True and password valid = True
Output: Create new account
Statement coverage: $5/5 * 100 = 100\%$

These three test cases cover the entire statement.

Statement 2: Making an order:

Code snippet

```
      describe('User makes an order', () => {
    it('should create a new order when valid items are added to the cart',
     async () => {
        // Mock user login
        const userCredentials = { username: 'user', password: 'userpass'
    };
        const userToken = await generateUserToken(userCredentials);

        // Mock request to make an order
        const orderData = {
            items: [
                { itemId: 'item1', quantity: 2 },
                { itemId: 'item2', quantity: 1 }
            ],
            deliveryOption: 'delivery', // Or 'dine-in'
            // Additional order data as needed
        };
        const response = await request(app)
            .post('/orders')
            .set('Authorization', `Bearer ${userToken}`)
            .send(orderData);

        // Assertions
        expect(response.status).toBe(201);
        expect(response.body.status).toBe('success');
        // Add more assertions as needed
    });
});
```

Flow chart

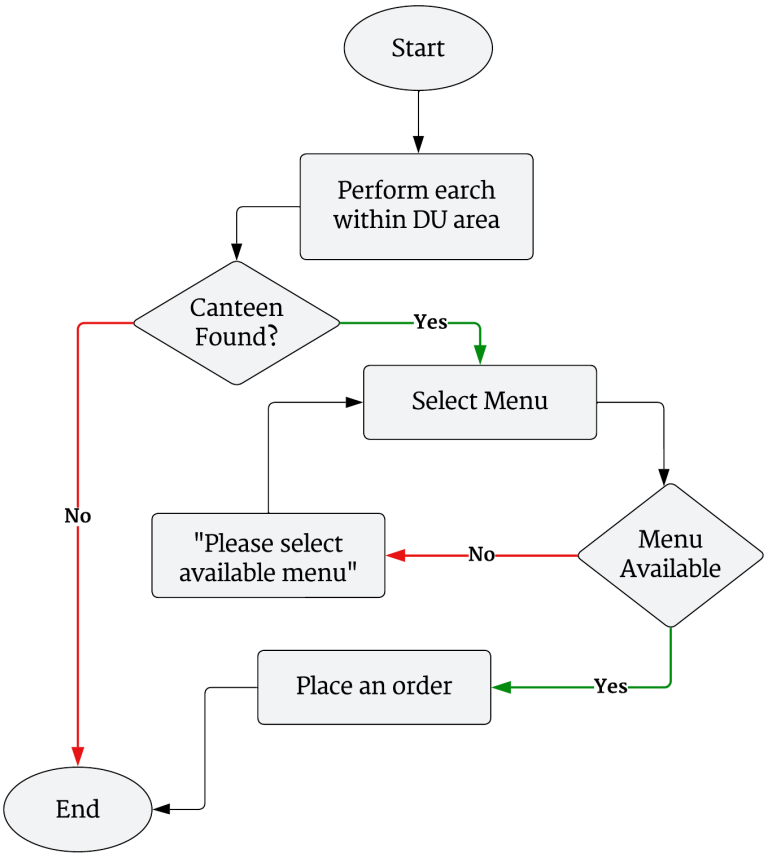


Figure 1.2: Statement 2

Test Case 1: Is canteen found = False
Output: End
Statement coverage: $5/7 * 100 = 71.42\%$

Test Case 2: Is canteen found = True and Menu Available = False
Output: Please select available menu
Statement coverage: $6/7 * 100 = 85.7\%$

Test Case 3: Is canteen found = True and Menu Available = True
Output: Ordered placed
Statement coverage: $6/7 * 100 = 85.7\%$

These three test cases cover the entire statement.

Statement 3: Update Order Status

Code snippet

```
    describe('Canteen owner updates order status', () => {
    it('should update order status when valid order ID and status are provided', async () => {
        // Mock canteen owner login
        const ownerCredentials = { username: 'owner', password: 'ownerpass' };
        const ownerToken = await generateCanteenOwnerToken(ownerCredentials);

        // Mock request to update order status
        const orderUpdateData = {
            orderId: 'order123', // Assuming this is the ID of the order to be updated
            status: 'completed' // New status value
        };
        const response = await request(app)
            .put(`/canteen/orders/${orderUpdateData.orderId}`)
            .set('Authorization', `Bearer ${ownerToken}`)
            .send(orderUpdateData);

        // Assertions
        expect(response.status).toBe(200);
        expect(response.body.status).toBe('success');
        // Add more assertions as needed
    });
});
```

Flow chart

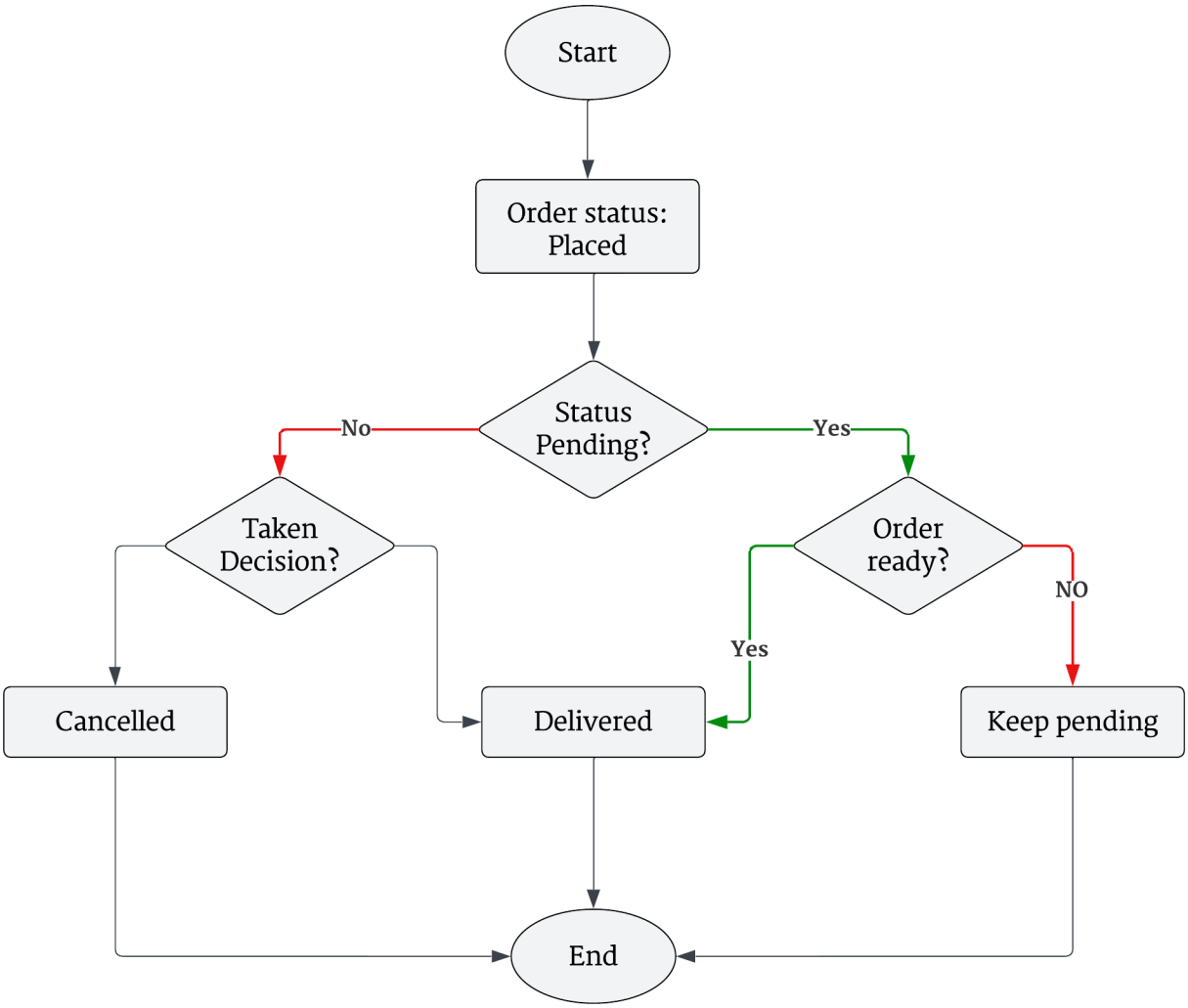


Figure 1.3: Statement 3

Test Case 1: Is Status Pending = False and Taken Decision = False
Output: Cancelled
Statement coverage: $4/6 * 100 = 66.67\%$

Test Case 2: Is Status Pending = False and Taken Decision = True
Output: Delivered
Statement coverage: $5/6 * 100 = 83.33\%$

Test Case 3: Is Status Pending = True and Order Ready = True
Output: Delivered
Statement coverage: $5/6 * 100 = 83.33\%$ **Test Case 3**: Is Status Pending = True and

Order Ready = False
Output: Keep pending
Statement coverage: $5/6 * 100 = 83.33\%$

These four test cases cover the entire statement.

Statement 4: Add an employee

Code snippet

```
     describe('Admin adds a new employee', () => {
    it('should add a new employee when valid data is provided', async () => {
        // Mock admin login
        const adminCredentials = { username: 'admin', password: 'adminpass' };
        const adminToken = await generateAdminToken(adminCredentials);

        // Mock request to add a new employee
        const employeeData = {
            username: 'newemployee',
            password: 'newpass',
            role: 'employee'
        };
        const response = await request(app)
            .post('/admin/employees')
            .set('Authorization', `Bearer ${adminToken}`)
            .send(employeeData);

        // Assertions
        expect(response.status).toBe(201);
        expect(response.body.status).toBe('success');
        // Add more assertions as needed
    });
});
```

Flow chart

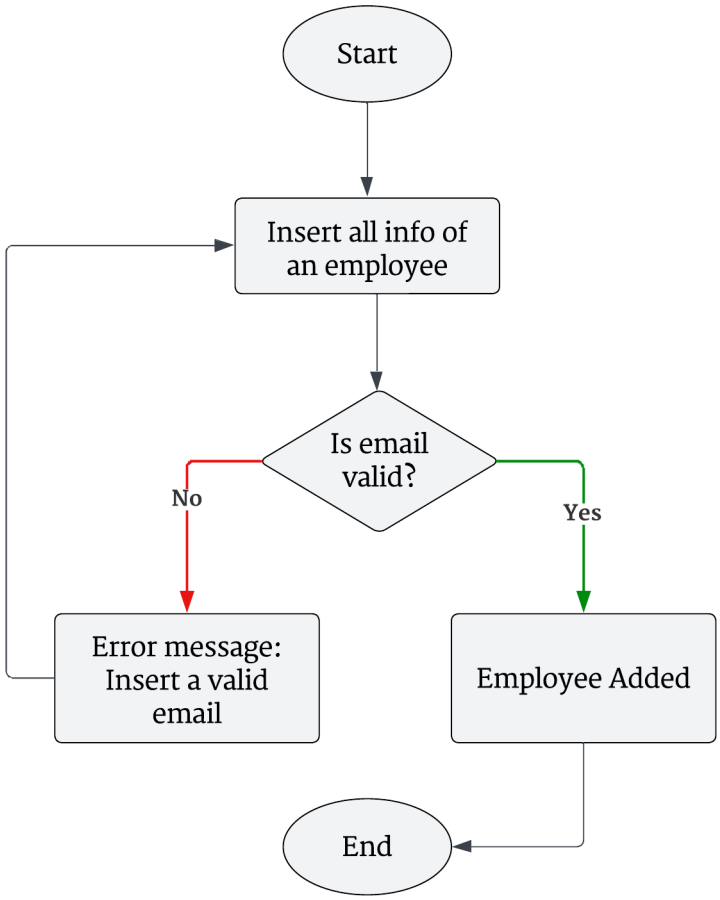


Figure 1.4: Statement 4

Test Case 1: is Email valid = True
Output: Employee Added
Statement coverage: $4/5 * 100 = 80\%$

Test Case 2: is Email valid = False
Output: Error message: Insert a valid email
Statement coverage: $4/5 * 100 = 80\%$

These two test case ensures the current status.

Branch and Conditional Coverage

Statement 1: Search a Canteen & validation:

Code snippet

```
      const assert = require('assert');
const app = require('../app'); // Assuming the Express app is exported from app.js
const Canteen = require('../models/Canteen');

describe('White-box Testing: Branch Coverage - Search Canteen and Validation', () => {
    beforeEach(async () => {
        // Clear the database or perform setup as needed
        await Canteen.deleteMany();
    });

    it('should return canteen details when a valid canteen name is provided', async () => {
        // Insert a canteen into the database
        const canteen = new Canteen({ name: 'Test Canteen', location: 'Test Location' });
        await canteen.save();

        // Mock request object
        const req = { query: { name: 'Test Canteen' } };
        let res = {};

        // Mock response object
        res.status = (statusCode) => {
            res.statusCode = statusCode;
            return res;
        };
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the search function
        await app.searchCanteen(req, res);

        // Assertions
        assert.strictEqual(res.statusCode, 200);
        assert.strictEqual(res.body.name, 'Test Canteen');
        assert.strictEqual(res.body.location, 'Test Location');
    });

    it('should return a validation error when an invalid canteen name is provided', async () => {
        // Mock request object
        const req = { query: { name: '' } };
        let res = {};

        // Mock response object
        res.status = (statusCode) => {
            res.statusCode = statusCode;
            return res;
        };
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the search function
        await app.searchCanteen(req, res);

```

```
57          // Assertions
58          assert.strictEqual(res.statusCode, 400);
59          assert.strictEqual(res.body.error, 'Canteen name is required');
60      });
61  });
```

Flow chart

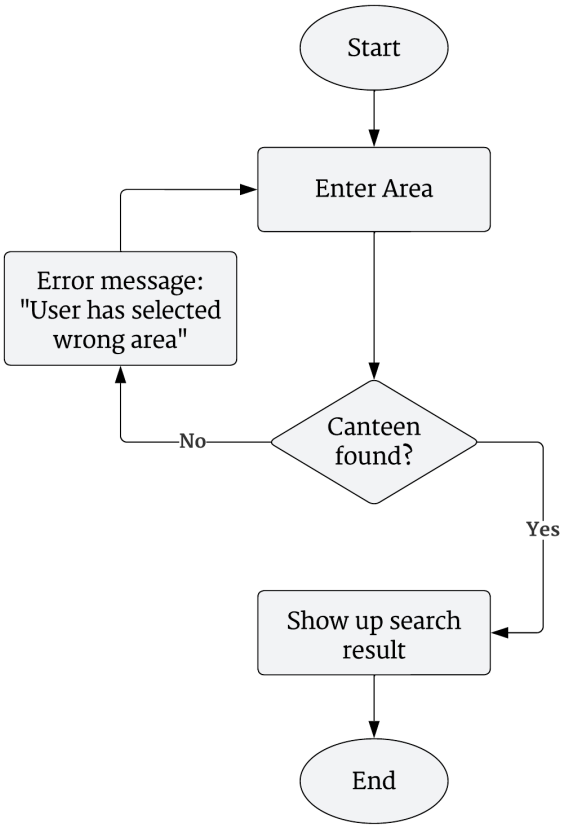


Figure 1.5: Branch 1

Test Case 1: Is Canteen found = False
Statement coverage: $5/7 * 100 = 71.42\%$

Test Case 2: Is Canteen found = True
Statement coverage: $6/7 * 100 = 85.7\%$

These two test cases cover the entire statement.

Statement 2: Selecting User Type

Code snippet

```
     const assert = require('assert');
const app = require('../app'); // Assuming the application logic is
    exported from app.js

describe('White-box Testing: Branch Coverage - Selecting User Type', () =>
     {
    it('should return "Customer" when userType is "customer"', () => {
        // Mock request object
        const req = { body: { userType: 'customer' } };

        // Mock response object
        let res = {};
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the selectUserType function
        app.selectUserType(req, res);

        // Assertions
        assert.strictEqual(res.body.userType, 'Customer');
    });

    it('should return "Admin" when userType is "admin"', () => {
        // Mock request object
        const req = { body: { userType: 'admin' } };

        // Mock response object
        let res = {};
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the selectUserType function
        app.selectUserType(req, res);

        // Assertions
        assert.strictEqual(res.body.userType, 'Admin');
    });

    it('should return "Guest" when userType is not provided', () => {
        // Mock request object
        const req = { body: {} };

        // Mock response object
        let res = {};
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the selectUserType function
        app.selectUserType(req, res);

```

```
55          // Assertions
56          assert.strictEqual(res.body.userType, 'Guest');
57      });
58
59      it('should return "Invalid User Type" when userType is invalid', () =>
        {
60          // Mock request object
61          const req = { body: { userType: 'invalid' } };
62
63          // Mock response object
64          let res = {};
65          res.json = (data) => {
66              res.body = data;
67              return res;
68          };
69
70          // Call the selectUserType function
71          app.selectUserType(req, res);
72
73          // Assertions
74          assert.strictEqual(res.body.error, 'Invalid User Type');
75      });
76  });
```

Flow chart

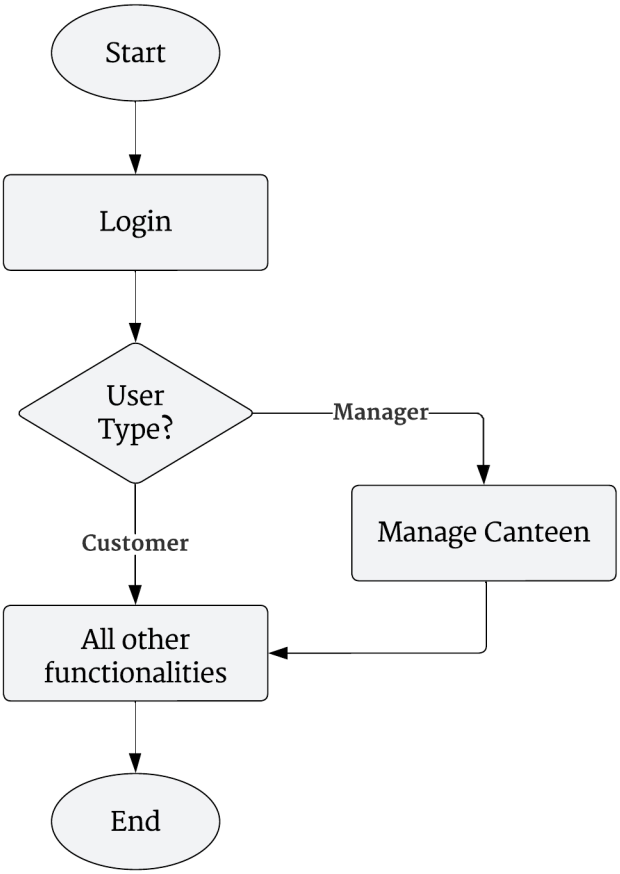


Figure 1.6: Condition 1

Test Case 1: Is User Type = Manager
Statement coverage: $4/6 * 100 = 66.67\%$

Test Case 2: Is User Type = Customer
Statement coverage: $5/6 * 100 = 83.33\%$

These two test cases cover the entire statement.

Statement 3: Selecting Order Type

Code snippet

```
     const assert = require('assert');
const app = require('../app'); // Assuming the application logic is exported from app.js

describe('White-box Testing: Branch Coverage - Selecting Order Type', () => {
    it('should return "Delivery" when orderType is "delivery"', () => {
        // Mock request object
        const req = { body: { orderType: 'delivery' } };

        // Mock response object
        let res = {};
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the selectOrderType function
        app.selectOrderType(req, res);

        // Assertions
        assert.strictEqual(res.body.orderType, 'Delivery');
    });

    it('should return "Dine-In" when orderType is "dine-in"', () => {
        // Mock request object
        const req = { body: { orderType: 'dine-in' } };

        // Mock response object
        let res = {};
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the selectOrderType function
        app.selectOrderType(req, res);

        // Assertions
        assert.strictEqual(res.body.orderType, 'Dine-In');
    });

    it('should return "Takeaway" when orderType is "takeaway"', () => {
        // Mock request object
        const req = { body: { orderType: 'takeaway' } };

        // Mock response object
        let res = {};
        res.json = (data) => {
            res.body = data;
            return res;
        };

        // Call the selectOrderType function
        app.selectOrderType(req, res);

```

```
55          // Assertions
56          assert.strictEqual(res.body.orderType, 'Takeaway');
57      });
58
59      it('should return "Unknown Order Type" when orderType is invalid', ()
        => {
60          // Mock request object
61          const req = { body: { orderType: 'invalid' } };
62
63          // Mock response object
64          let res = {};
65          res.json = (data) => {
66              res.body = data;
67              return res;
68          };
69
70          // Call the selectOrderType function
71          app.selectOrderType(req, res);
72
73          // Assertions
74          assert.strictEqual(res.body.error, 'Unknown Order Type');
75      });
76  });
```

Flow chart

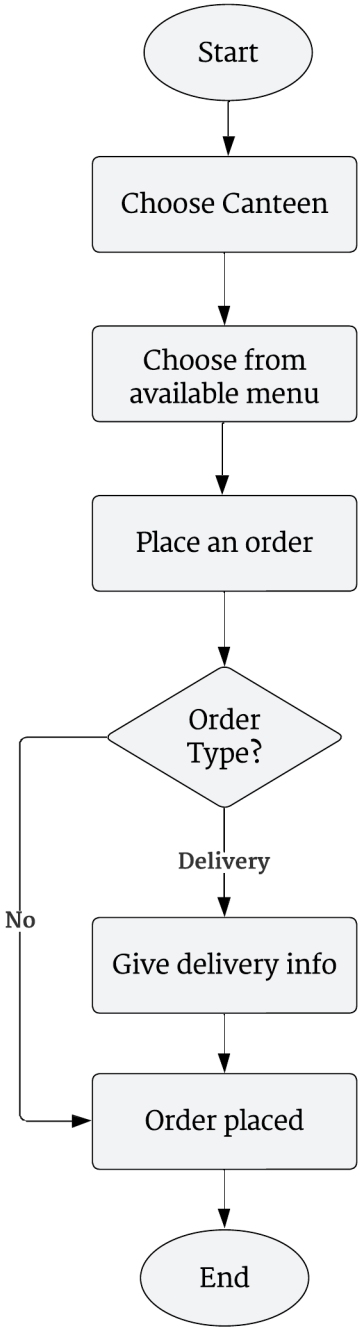


Figure 1.7: Condition 2

Test Case 1: Is Order Type = Delivery
Statement coverage: $4/5 * 100 = 80\%$

Test Case 2: Is Order Type = Dine In
Statement coverage: $4/5 * 100 = 80\%$

This test case ensures the current status.

Path Coverage

Statement 1: Adding a menu in the menu list

Flow chart

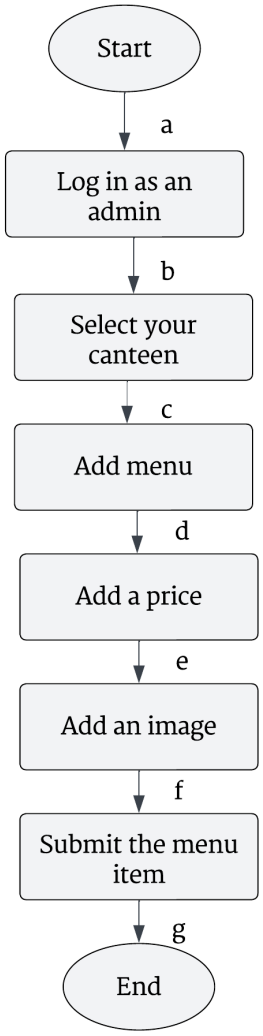


Figure 1.8: Statement 1

Test Case 1: Is username available = True
Path: a-b-c-d-e-f-g

This path covers the entire statement.

Statement 2: Adding a rating and feedback

Flow chart

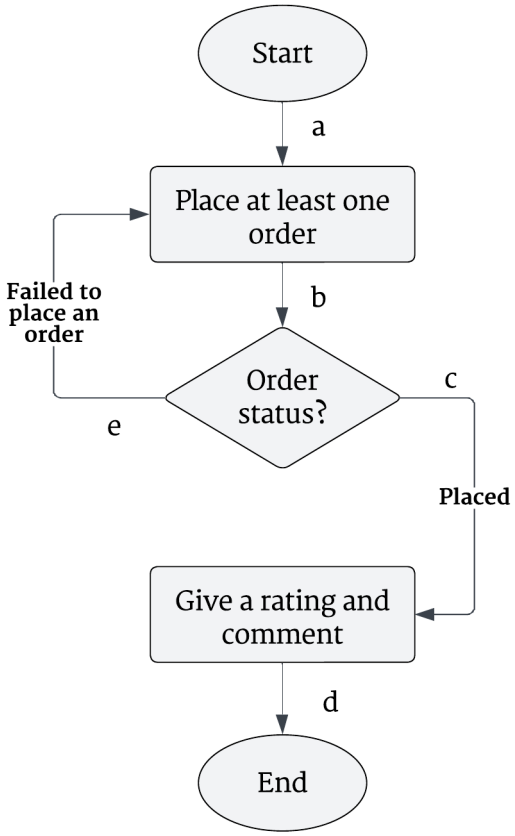


Figure 1.9: Statement 2

Test Case 1: Is Order Status = Placed
Path: a-b-c-d

Test Case 2: Is Order Status = Failed to place
Path: a-b-e-b-c-d

These paths cover the entire statement.

Statement 3: Adding an ingredient in the inventory

Flow chart

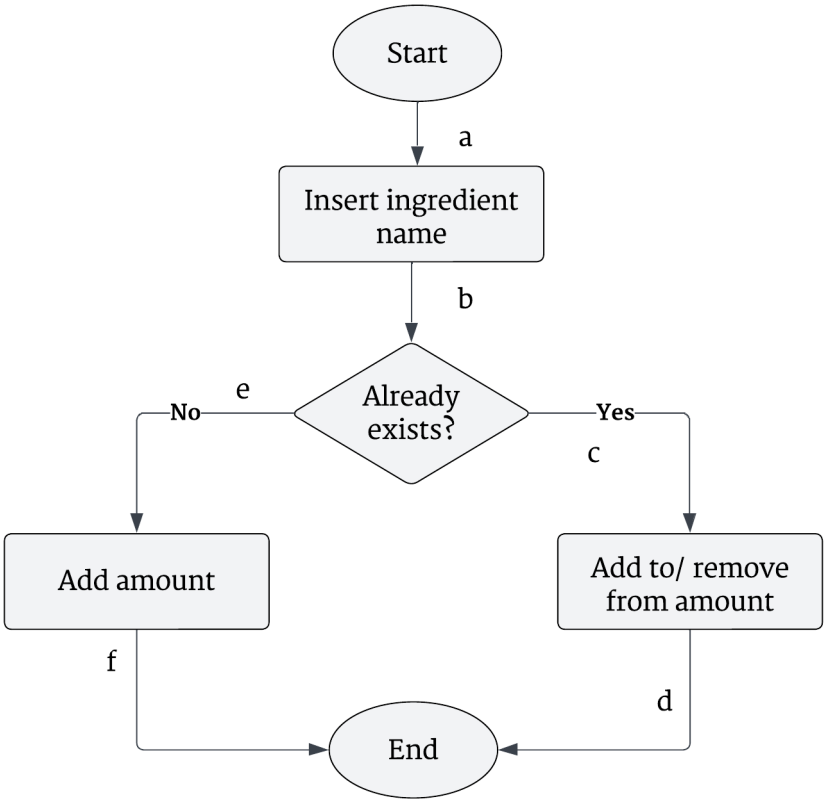


Figure 1.10: Statement 2

Test Case 1: Ingredient exists = True
Path: a-b-c-d

Test Case 2: Ingredient exists = false
Path: a-b-e-f

These paths cover the entire statement.

Statement 4: Add a new canteen

Flow chart

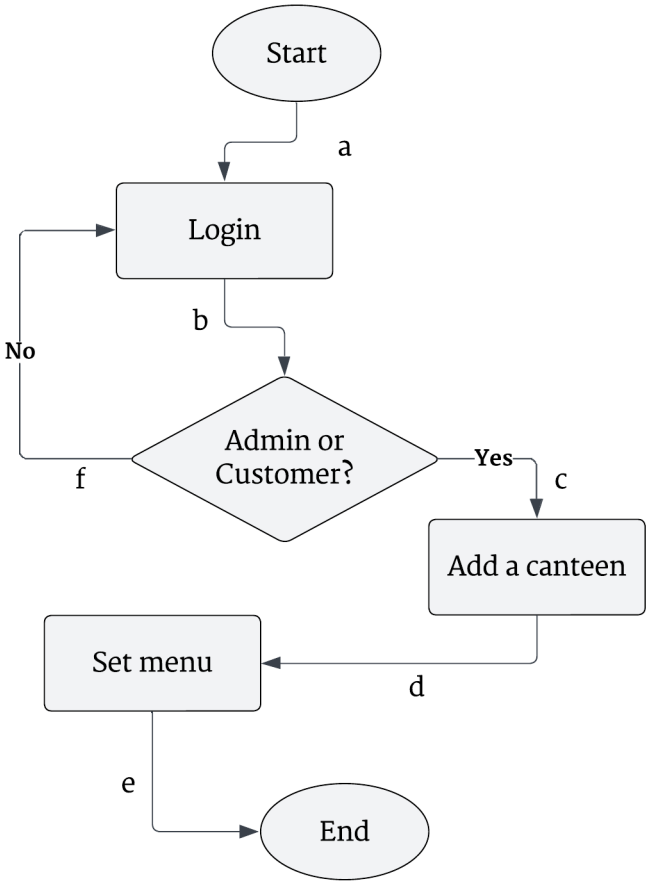


Figure 1.11: Statement 3

Test Case 1: is user admin = True
Path: a-b-c-d-e

Test Case 2: is user admin = False
Path: a-b-f-b-c-d-e

These paths cover the entire statement.

Chapter 2

Integration Testing

For our application, we have decided to use a Bottom-up approach for integration testing. The reason for this is that our application follows an object-oriented approach and this approach would be quite suitable.

2.1 Top-Down Testing

Test 1: Validate Integration Between Canteen Area Search and Canteen Database Interface

Test Case Number	
Statement of Purpose	Validate Integration Between Canteen Area Search and Canteen Database Interface
Interface(s) Being Tested	CanteenAreaSearchInterface CanteenDatabaseInterface
Description of Precondition	The canteen database contains a list of canteens with their respective areas.
Test Case Inputs	Search query: "Campus A"
Expected Output	A list of canteens in "Campus A".
Description of Expected Post-conditions	The response contains only canteens located in "Campus A".
Execution History	The test successfully returns canteens within the specified campus area, indicating correct integration between the search and database interfaces.

Stub code

```
1 describe('Integration Test: Canteen Area Search and Canteen Database
      Interface', () => {
2     it('should return canteens within the specified campus area', async ()
       => {
3         // Mock data for canteen database
```

```
        const mockCanteens = [
            { id: 1, name: 'Canteen A', area: 'Campus A' },
            { id: 2, name: 'Canteen B', area: 'Campus B' }
        ];

        // Mock canteen database interface
        jest.mock('../models/canteen', () => ({
            find: jest.fn().mockImplementation((query) => {
                return mockCanteens.filter(canteen => canteen.area === query.area);
            })
        }));

        // Make a request to search canteens in "Campus A"
        const response = await request(app)
            .get('/canteens/search')
            .query({ area: 'Campus A' });

        // Assertions
        expect(response.status).toBe(200);
        expect(response.body).toEqual([{ id: 1, name: 'Canteen A', area: 'Campus A' }]);
    });
});
```

Test 2: Validate Integration Between the LoginInterface and HomePageInterface

Test Case Number	
Statement of Purpose	Validate Integration Between the LoginInterface and HomePageInterface
Interface(s) Being Tested	LoginInterface HomePageInterface
Description of Precondition	The user has a valid account and is on the login page.
Test Case Inputs	Username: "testuser" Password: "testpass"
Expected Output	Redirection to the home page upon successful login.
Description of Expected Post-conditions	The user is redirected to the home page.
Execution History	The test successfully logs in the user and redirects to the home page, indicating correct integration between the login and home page interfaces.

Stub code

```
describe('Integration Test: LoginInterface and HomePageInterface', () => {
    it('should redirect to the home page upon successful login', async () => {
        // Mock user data
        const mockUser = { username: 'testuser', password: 'testpass' };

        // Mock user authentication interface
```

```
        jest.mock('../models/user', () => ({
            findOne: jest.fn().mockImplementation((query) => {
                if (query.username === mockUser.username && query.password
    === mockUser.password) {
                    return mockUser;
                }
                return null;
            })
        }));

        // Mock request to login
        const response = await request(app)
            .post('/login')
            .send({ username: 'testuser', password: 'testpass' });

        // Assertions
        expect(response.status).toBe(200);
        expect(response.body.redirectTo).toBe('/home');
    });
});
```

Test 3: Validate Integration Between the OrderingInterface and InventoryInterface

Test Case Number	
Statement of Purpose	Validate Integration Between the OrderingInterface and InventoryInterface
Interface(s) Being Tested	LoginInterface HomePageInterface
Description of Precondition	The inventory contains items available for order.
Test Case Inputs	Order details: itemId: 1, quantity: 2
Expected Output	Inventory is updated to reflect the order.
Description of Expected Post-conditions	The inventory quantity for the ordered item is decreased by the ordered quantity.
Execution History	The test successfully place an order and updates the inventory, indicating correct integration between the ordering and inventory interfaces.

Stub code

```
describe('Integration Test: OrderingInterface and InventoryInterface', ()
    => {
    it('should update inventory upon placing an order', async () => {
        // Mock inventory data
        const mockInventory = { itemId: 1, quantity: 10 };

        // Mock inventory interface
        jest.mock('../models/inventory', () => ({
            findOne: jest.fn().mockImplementation((query) => {
                return mockInventory;
            }),
            updateOne: jest.fn().mockImplementation((query, update) => {
```

```
                mockInventory.quantity -= update.$inc.quantity;
                return mockInventory;
            })
        }));

        // Mock request to place an order
        const response = await request(app)
            .post('/orders')
            .send({ itemId: 1, quantity: 2 });

        // Assertions
        expect(response.status).toBe(201);
        expect(mockInventory.quantity).toBe(8); // Inventory should be
    updated
    });
});
```

Test 4: Validate Integration Between the PaymentInterface and SalesRecordInterface

Test Case Number	
Statement of Purpose	Validate Integration Between the PaymentInterface and SalesRecordInterface
Interface(s) Being Tested	PaymentInterface and SalesRecordInterface
Description of Precondition	The system is ready to process a payment for an order.
Test Case Inputs	Payment details: orderId: 1, amount: 20.00
Expected Output	Sales record is updated to reflect the payment.
Description of Expected Post-conditions	The sales record for the paid order is updated with the payment amount.
Execution History	The test successfully processes the payment and updates the sales record, indicating correct integration between the payment and sales record interfaces.

Stub code

```
describe('Integration Test: PaymentInterface and SalesRecordInterface', ()
    => {
    it('should update sales records upon successful payment', async () =>
    {
        // Mock sales record data
        const mockSalesRecord = { orderId: 1, amount: 0 };

        // Mock sales record interface
        jest.mock('../models/salesRecord', () => ({
            findOne: jest.fn().mockImplementation((query) => {
                return mockSalesRecord;
            }),
            updateOne: jest.fn().mockImplementation((query, update) => {
                mockSalesRecord.amount = update.$set.amount;
                return mockSalesRecord;
```

```
            })
        }));

        // Mock request to process payment
        const response = await request(app)
            .post('/payments')
            .send({ orderId: 1, amount: 20.00 });

        // Assertions
        expect(response.status).toBe(200);
        expect(mockSalesRecord.amount).toBe(20.00); // Sales record should
     be updated
    });
});
```

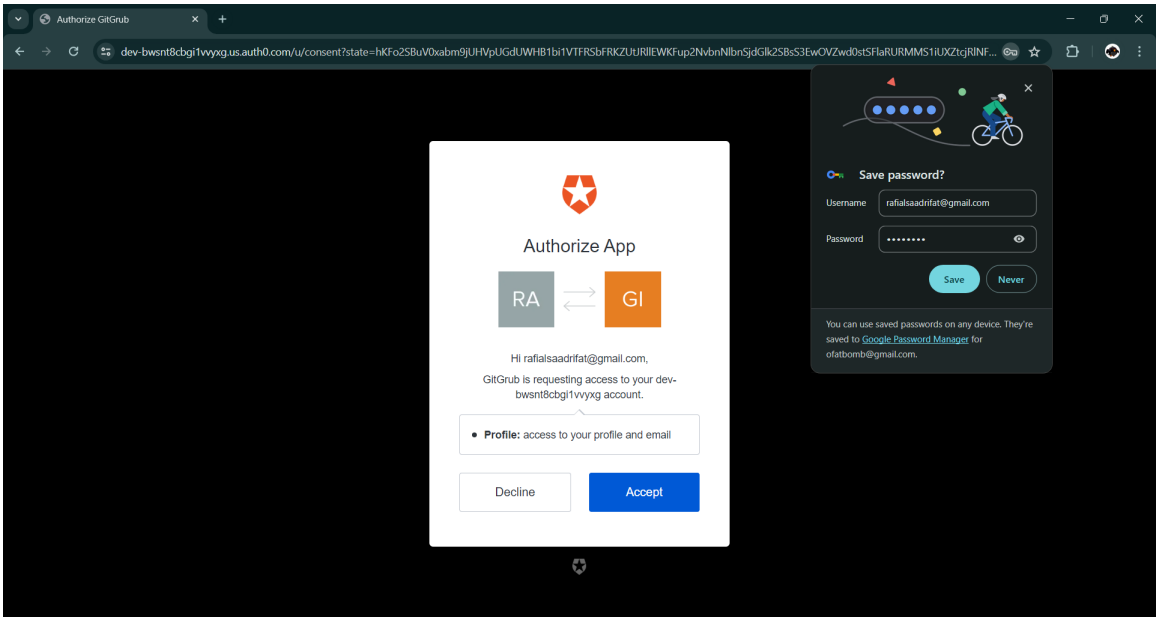
Chapter 3

Acceptance Testing

3.1 Functional Testing

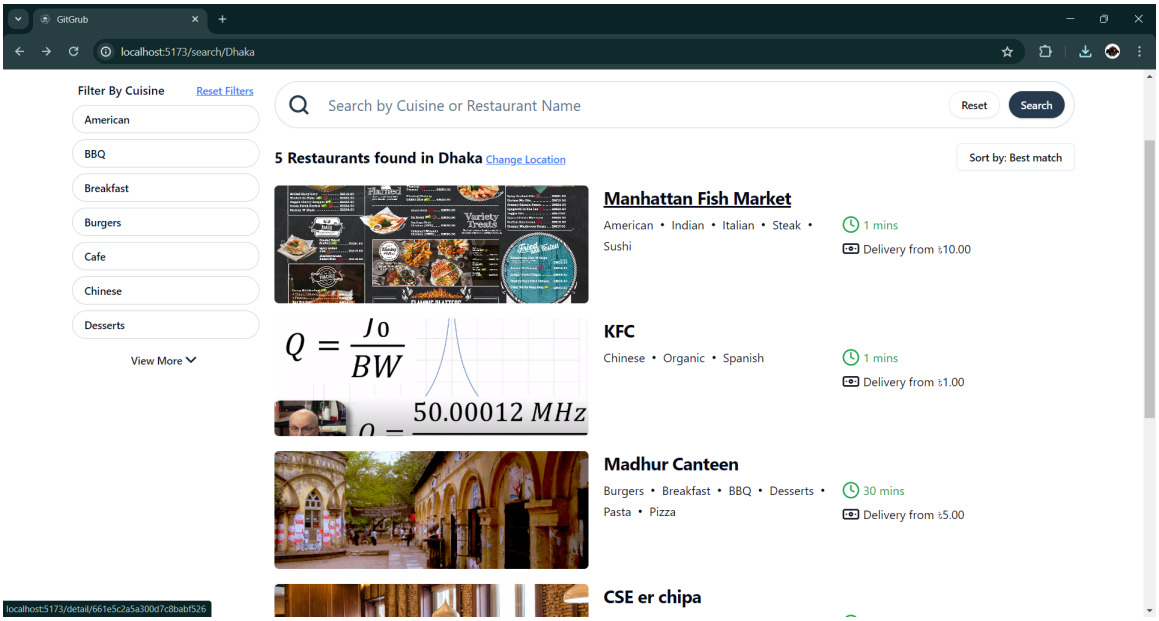
1. Sign up as a new user

Result: A new user account was created



Status: Passed

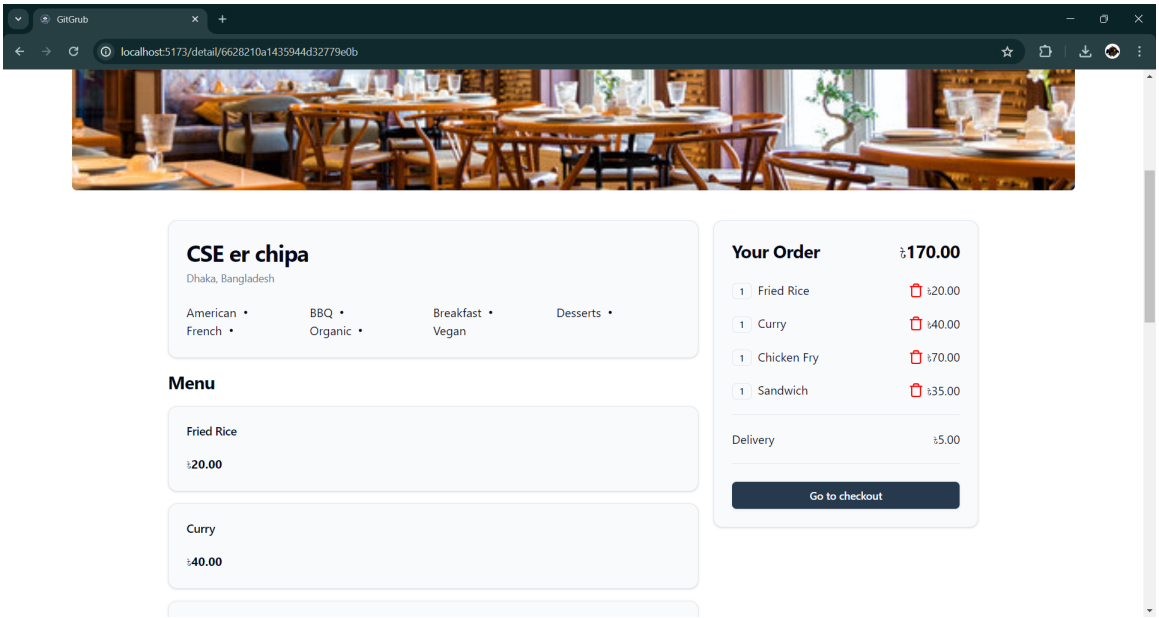
2. Select Canteens via the exclusive search bar that can search upon cuisine type or canteen name



Result: All the canteens within the searched area were selected

Status: Passed

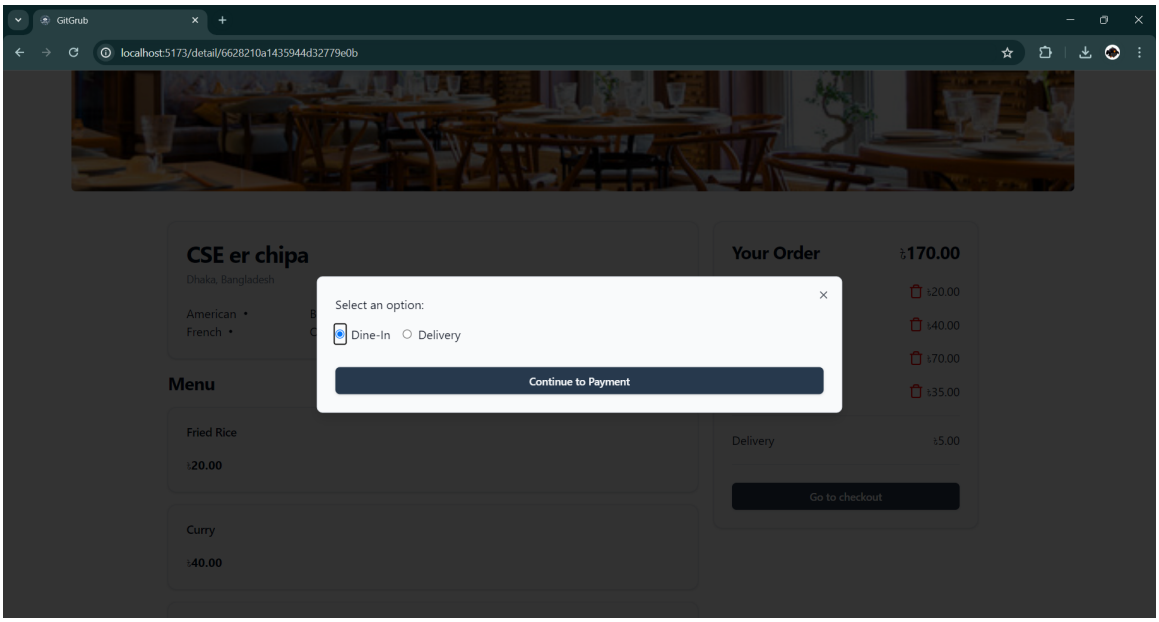
3. Place an order by selecting multiple menu items



Result: An order was placed

Status: Passed

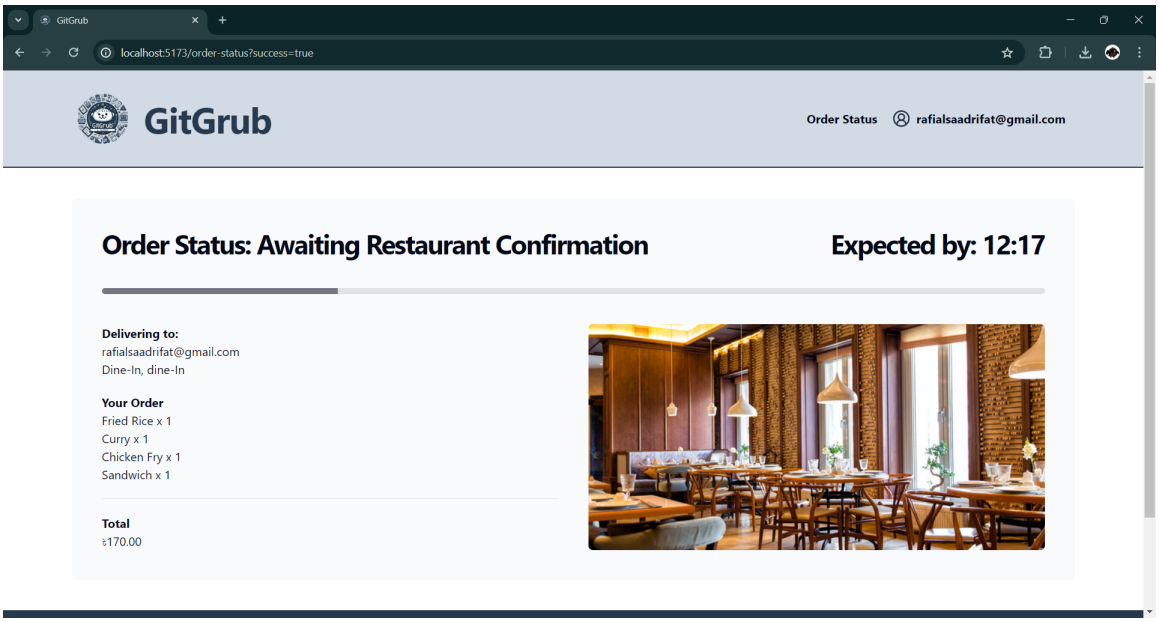
4. Choosing order type: Dine-in or Delivery

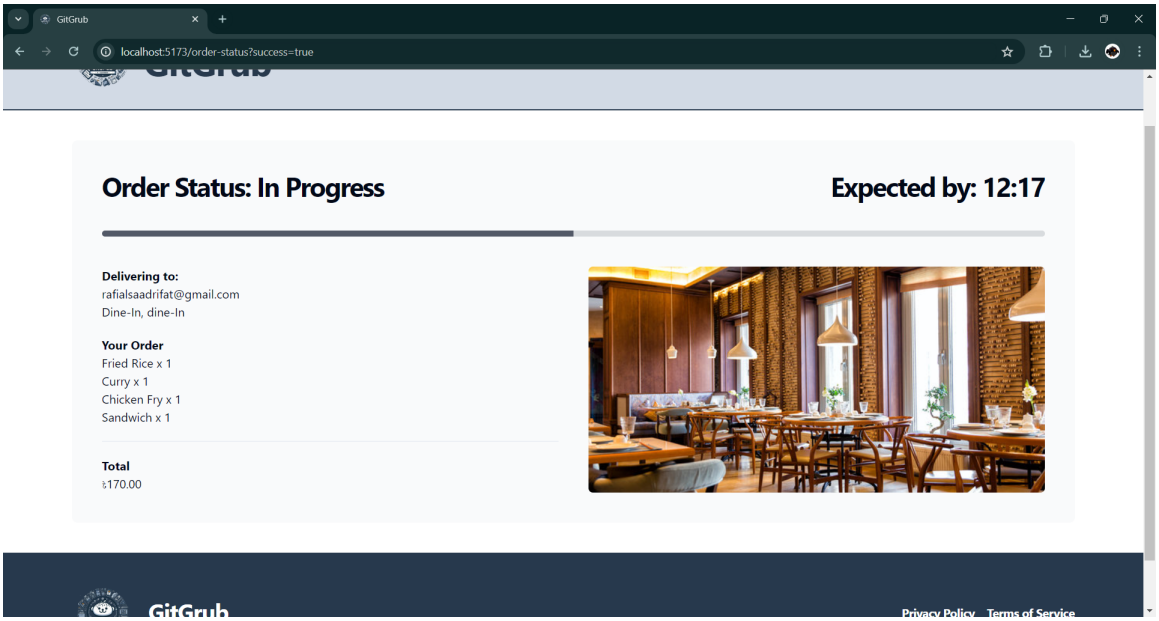


Result: Order type was chosen

Status: Passed

5. Updating Order Status

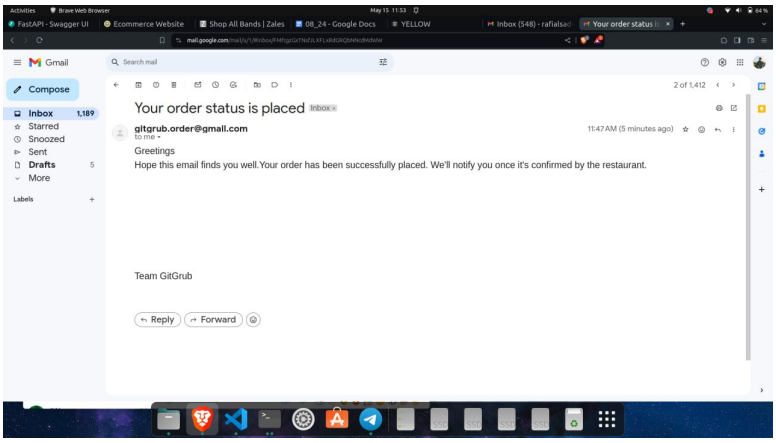


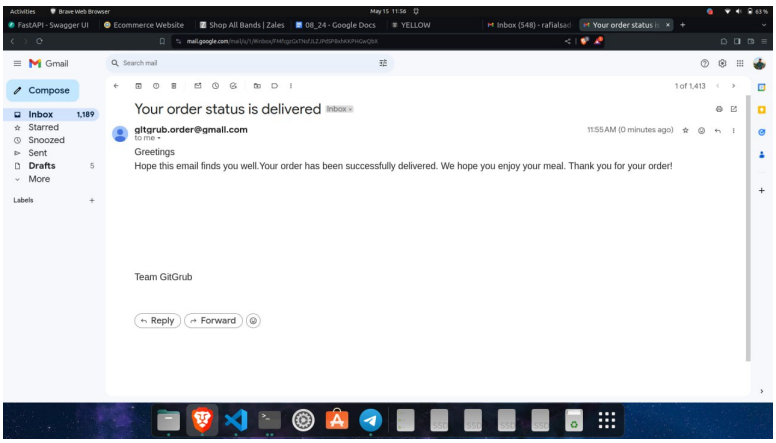


Result: Order status got up to date from time to time

Status: Passed

6. Customer receiving email when order status is delivered

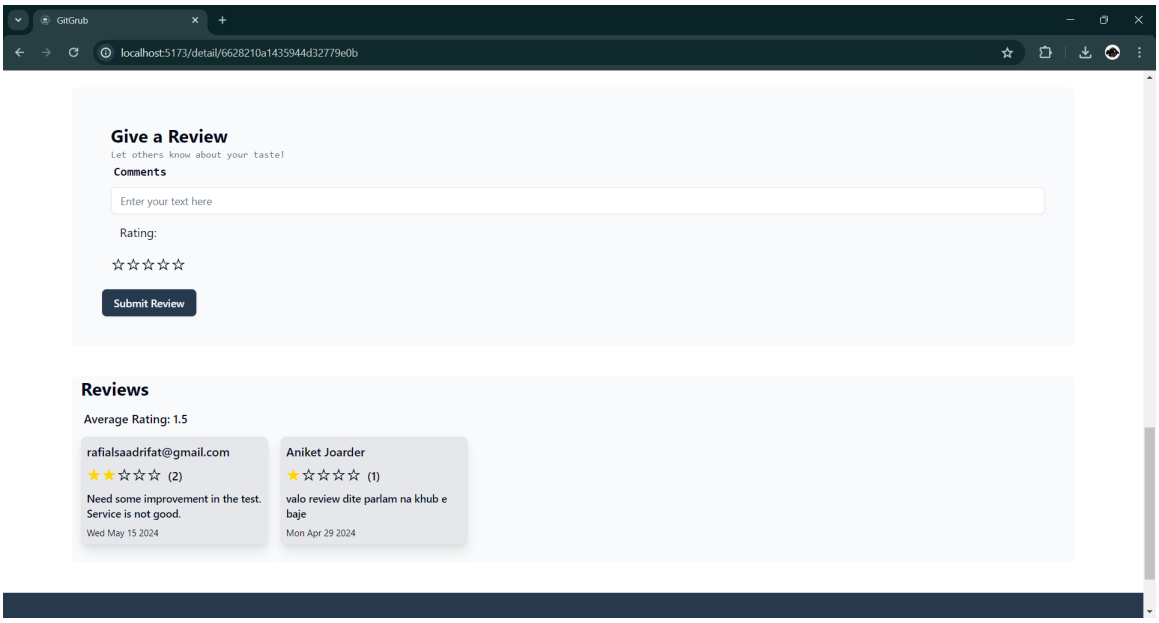




Result: The customer received an email when his/her order is ready to deliver

Status: Passed

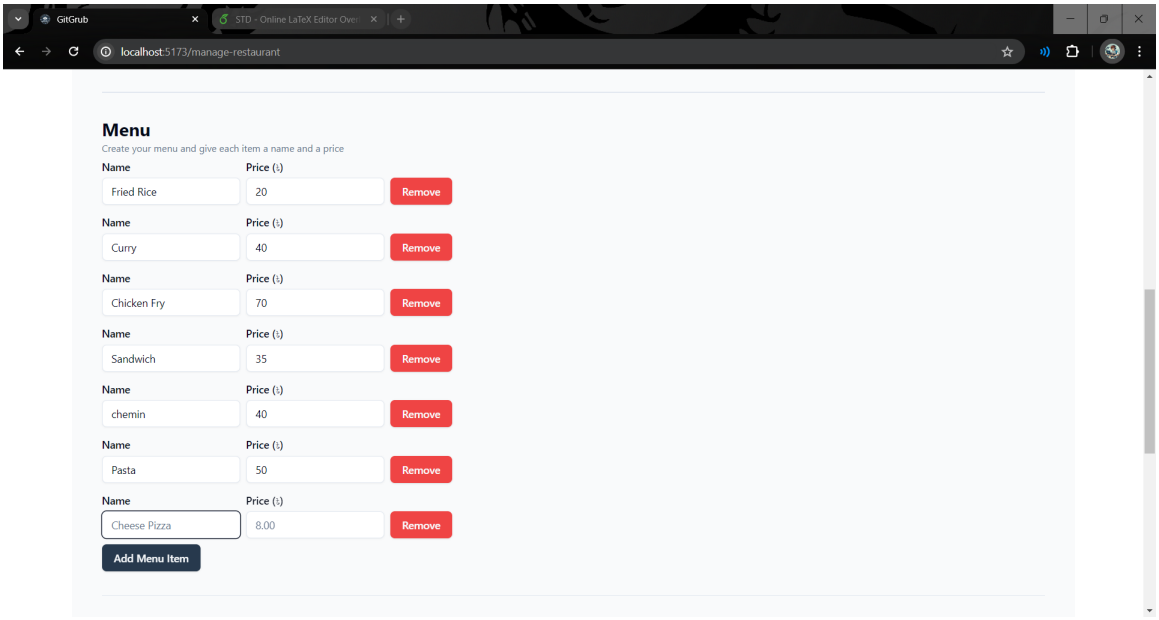
7. Giving and viewing ratings and feedback



Result: A customer was able to give his/her opinion about the canteen and any menu item

Status: Passed

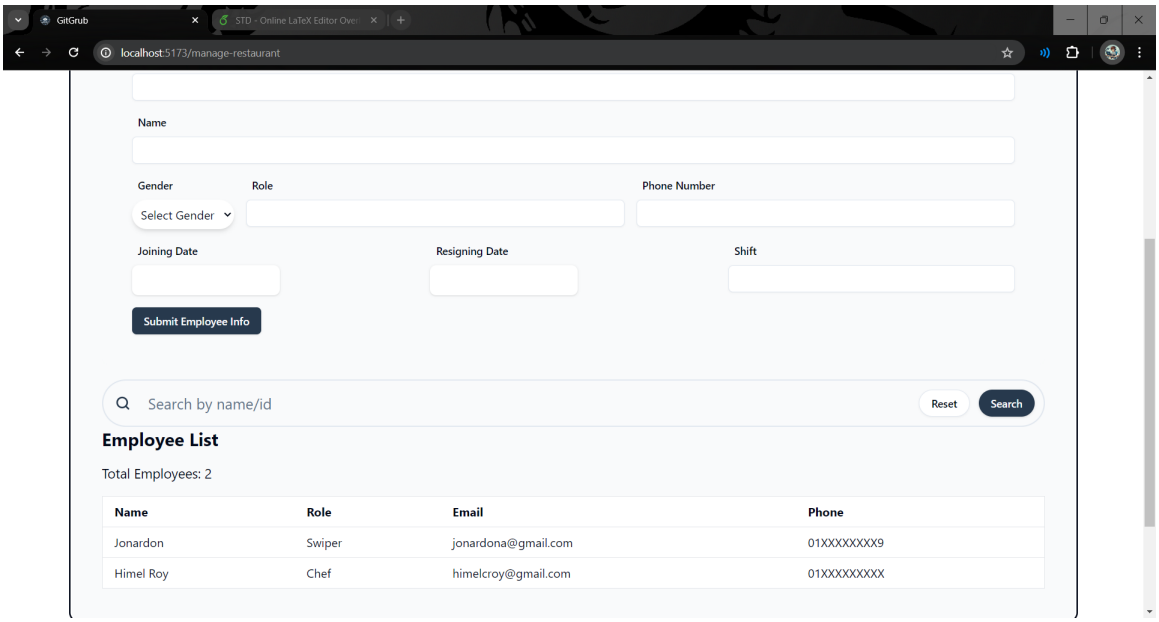
8. Managing the menu items of any canteen

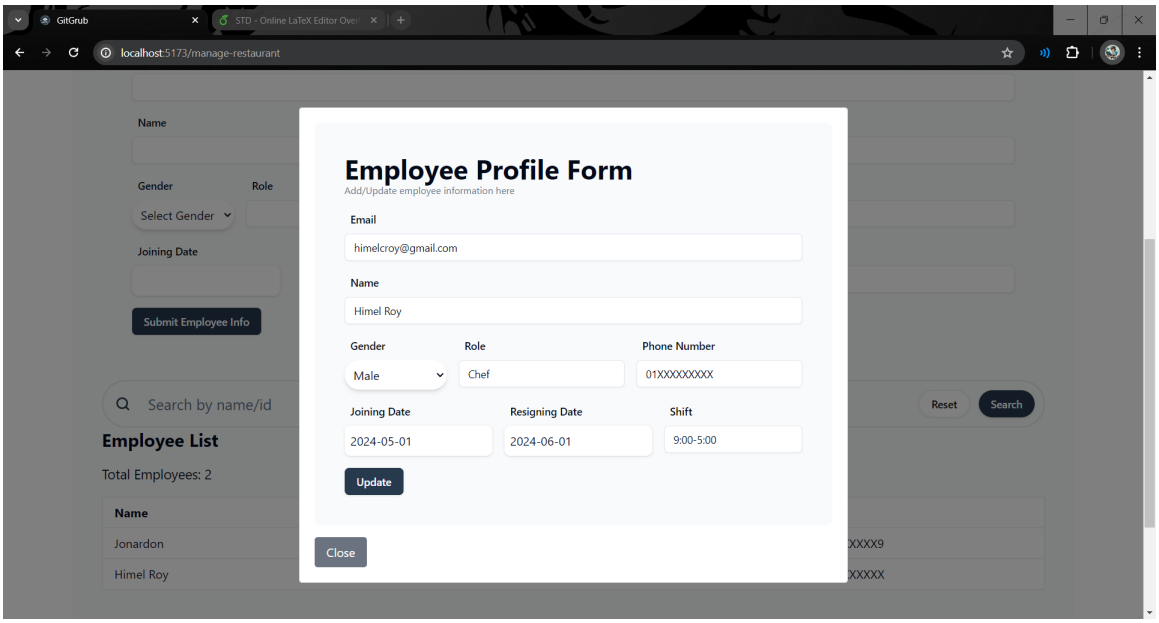


Result: The manager was able to successfully add or remove any menu item from the canteen

Status: Passed

9. Managing the employees of a canteen

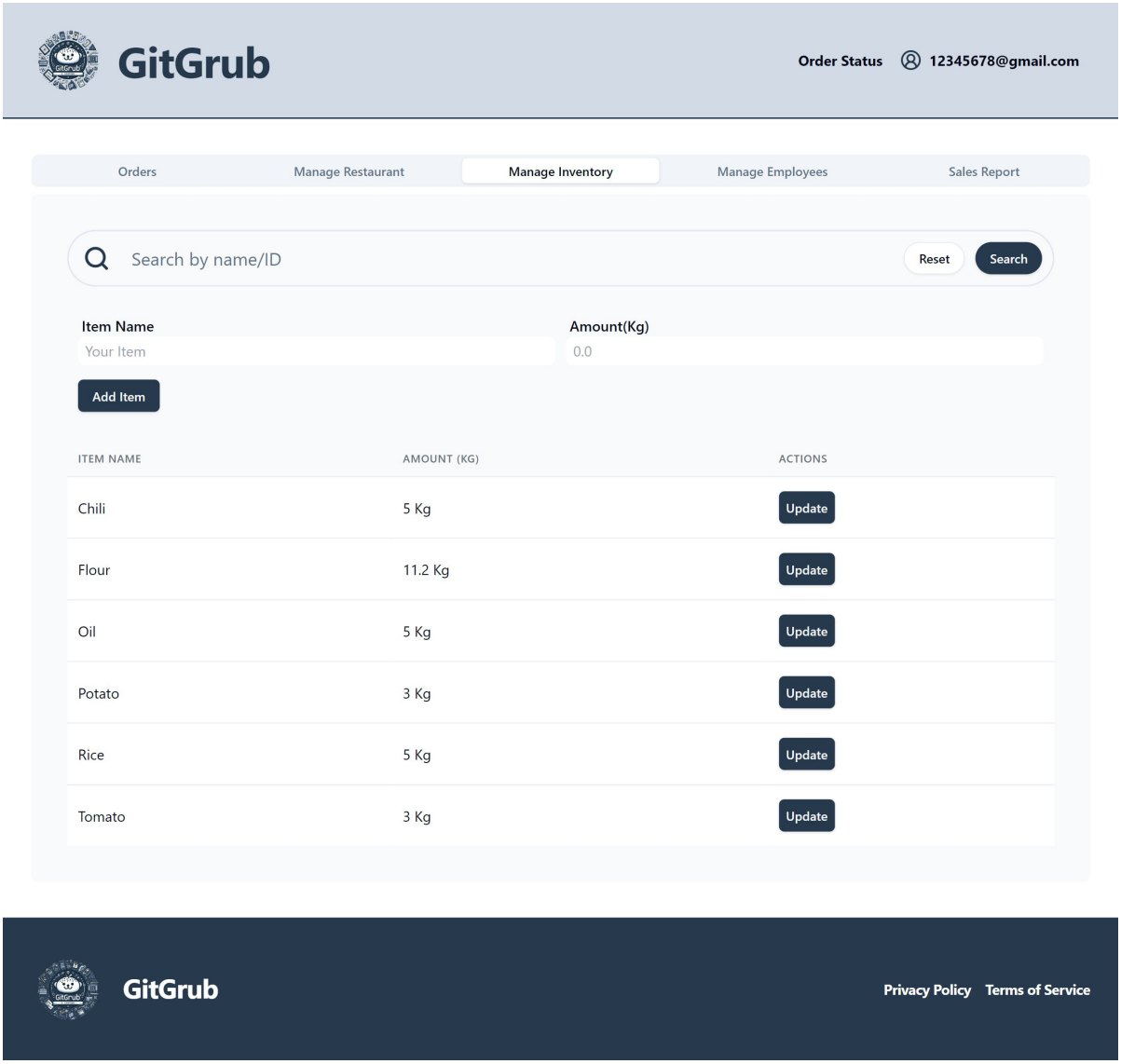




Result: The manager was able to successfully add or search any employee item from the canteen

Status: Passed

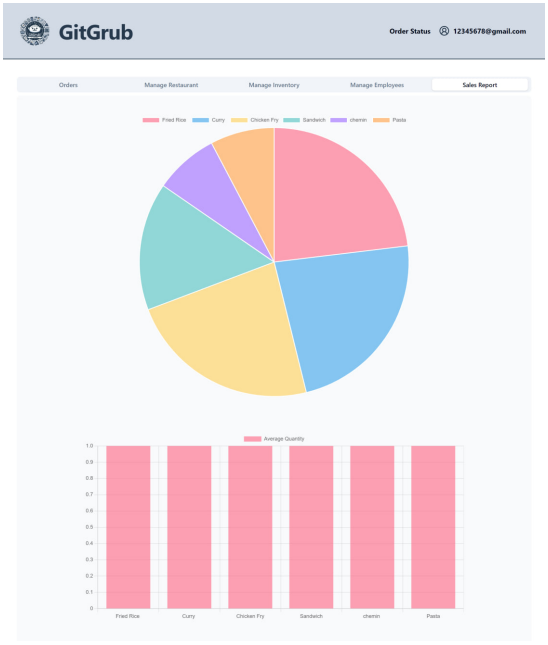
10. Managing the inventory of a canteen



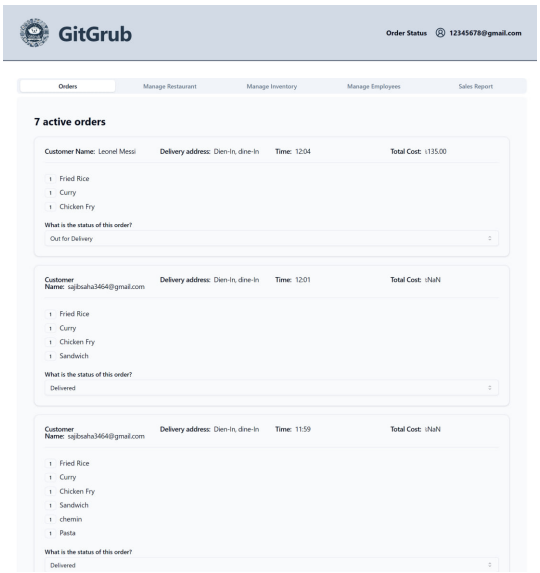
Result: The manager was able to successfully add or remove or update any ingredient from the canteen

11. Track the sales record of a canteen

Result: The manager was able to track the sales record of the canteen

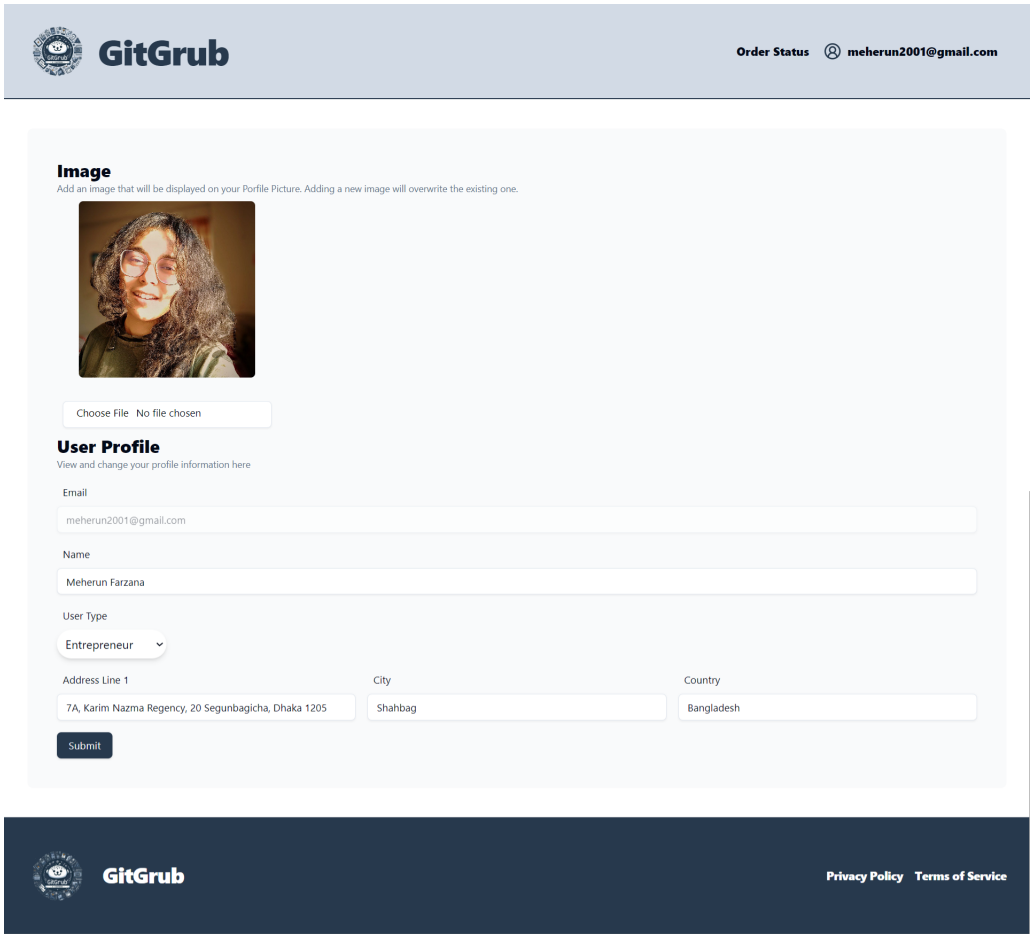


12. Track all orders of a canteen



Result: The manager was able to track all orders of a canteen

13. Updating user profile as a manager and a customer



Result: Any user was able to update their profile

3.2 Performance Testing

3.2.1 Security Testing

Test Case Identifier	Description	Outcomes
Test-Case-01	User Authorization	We verified that a registered user can successfully log in with valid credentials and is granted access to search canteens, place an order, track order, and give and read feedbacks.
Test-Case-02	Verify Admin Authorization	We verify that an users who are manager of canteens are have some privileged functionalities such as managing the canteens, inventory, employees and sales record.
Test-Case-03	Verify User Input Validation	We verified that the application performs proper validation and handling of user inputs. We verified this by testing for potential vulnerabilities such as SQL injection, cross-site scripting (XSS), and other common security risks.
Test-Case-04	Verify Password Security	We verified that the application securely stores passwords using appropriate hashing and encryption techniques. We verified this by ensuring that passwords are not stored in plain text and cannot be easily accessed or decrypted.
Test-Case-05	Verify Error Handling	We verified that the application handles errors appropriately by displaying meaningful error messages and avoiding the leakage of sensitive information.
Test-Case-06	Verify Data Confidentiality	We verified that sensitive user information such as passwords, booking details, and financial information are securely stored and transmitted to maintain data confidentiality.

3.2.2 Timing Testing

The test was done on windows and Mac platform and it was perfectly run well.

Operation	**Number of Requests**	**Average Execution Time (s)**	**Description**
User Registr nation	100	0.2	We verified the average execution time for registering 100 users, measuring the time taken for each request.
Canteen Search	50	0.1	We verified the average execution time for searching canteens 50 times, measuring the time taken for each search request.
Placing orders	20	0.3	We verified the average execution time for placing 20 orders, measuring the time taken for each order request.
Sales Report	10	0.5	We verified the average execution time for generating sales records, measuring the time taken for each record generation request.
Managing canteens	10	0.4	We verified the average execution time for updating 10 canteen info, measuring the time taken for each request.
Updating user profiles	20	0.2	We verified the average execution time for profile updates of 20 users, measuring the time taken for each request.

3.2.3 Volume Testing

Test1: Creating 700 users

Stub code

```
1
2 const mongoose = require('mongoose');
3 const User = require('../models/User');
4
5 // Function to generate random usernames and passwords
6 function generateUsernamesAndPasswords(count) {
7     const users = [];
8     for (let i = 1; i <= count; i++) {
9         const username = `user${i}`;
10        const password = `password${i}`;
11        users.push({ username, password });
12    }
13    return users;
14 }
15
16 // Test Case: Creating 700 users
17 describe('Volume Testing: Creating 700 Users', () => {
18     before(async () => {
19         // Connect to MongoDB
20         await mongoose.connect('mongodb://localhost:27017/
       canteen_management', { useNewUrlParser: true, useUnifiedTopology: true
       });
21     });
22
23     it('should create 700 users in the database', async () => {
24         // Generate user data
25         const userData = generateUsernamesAndPasswords(700);
26
27         // Create users in the database
28         const createdUsers = await User.create(userData);
29
30         // Assertions
31         expect(createdUsers.length).to.equal(700);
32     });
33
34     after(async () => {
35         // Disconnect from MongoDB
36         await mongoose.disconnect();
37     });
38 });
```

This script uses Mocha as the testing framework and Chai for assertions. It first generates 700 random usernames and passwords, then creates users in the database using the User.create() method from Mongoose. Finally, it asserts that the number of created users matches the expected count.

Test2: Searching for canteens 5000 times

Stub code

```
const mongoose = require('mongoose');
const Canteen = require('../models/Canteen');

// Function to perform canteen search
async function searchCanteens() {
    try {
        // Perform canteen search 5000 times
        for (let i = 0; i < 5000; i++) {
            const searchTerm = `search${i}`;
            const canteens = await Canteen.find({ name: searchTerm });
            // Optionally, you can perform assertions on the search results
        }
    } catch (error) {
        console.error('Error searching for canteens:', error);
    }
}

// Test Case: Searching for canteens 5000 times
describe('Volume Testing: Searching for Canteens 5000 Times', () => {
    before(async () => {
        // Connect to MongoDB
        await mongoose.connect('mongodb://localhost:27017/canteen_management', { useNewUrlParser: true, useUnifiedTopology: true });
    });

    it('should search for canteens 5000 times without errors', async () => {
        // Perform canteen search
        await searchCanteens();
    });

    after(async () => {
        // Disconnect from MongoDB
        await mongoose.disconnect();
    });
});
```

This script simulates searching for canteens 5000 times by iterating over a loop and executing the search operation in each iteration. You can adjust the search term generation and search criteria according to your project requirements.

Test3: Place an order 10,000 times

```
    const mongoose = require('mongoose');
const Order = require('../models/Order');
const { generateRandomOrder } = require('./helpers'); // Assuming a helper function to generate random orders

// Function to place orders
async function placeOrders() {
```

```
    try {
        // Place orders 10,000 times
        for (let i = 0; i < 10000; i++) {
            const orderData = generateRandomOrder(); // Generate random order data
            await Order.create(orderData); // Place the order
        }
    } catch (error) {
        console.error('Error placing orders:', error);
    }
}

// Test Case: Placing an order 10,000 times
describe('Volume Testing: Placing an Order 10,000 Times', () => {
    before(async () => {
        // Connect to MongoDB
        await mongoose.connect('mongodb://localhost:27017/canteen_management', { useNewUrlParser: true, useUnifiedTopology: true });
    });

    it('should place orders 10,000 times without errors', async () => {
        // Place orders
        await placeOrders();
    });

    after(async () => {
        // Disconnect from MongoDB
        await mongoose.disconnect();
    });
});
```

This script simulates placing an order 10,000 times by iterating over a loop and creating a random order using a helper function in each iteration. You can adjust the order data generation logic according to your project's requirements.

Test4: Update Inventory 7000 times

```
    const mongoose = require('mongoose');
const Inventory = require('../models/Inventory');
const { generateRandomInventoryUpdate } = require('./helpers'); // Assuming a helper function to generate random inventory updates

// Function to update inventory
async function updateInventory() {
    try {
        // Update inventory 7000 times
        for (let i = 0; i < 7000; i++) {
            const updateData = generateRandomInventoryUpdate(); // Generate random inventory update data
            // Assuming updateData contains the inventory item ID and quantity to update
            await Inventory.findByIdAndUpdate(updateData.itemId, { $inc: { quantity: updateData.quantity } });
        }
    } catch (error) {
        console.error('Error updating inventory:', error);
    }
```

```
}

// Test Case: Updating Inventory 7000 times
describe('Volume Testing: Updating Inventory 7000 Times', () => {
    before(async () => {
        // Connect to MongoDB
        await mongoose.connect('mongodb://localhost:27017/canteen_management', { useNewUrlParser: true, useUnifiedTopology: true });
    });

    it('should update inventory 7000 times without errors', async () => {
        // Update inventory
        await updateInventory();
    });

    after(async () => {
        // Disconnect from MongoDB
        await mongoose.disconnect();
    });
});
```

This script simulates updating inventory 7000 times by iterating over a loop and updating inventory items using a helper function in each iteration. You can adjust the inventory update data generation logic according to your project's requirements.

3.3 Acceptance Testing

3.3.1 Alpha Testing

The feedback got from mock clients (our friends) in the development environment made us discover quite a few bugs and inconsistencies in our system and we had to make some modifications and corrections based on those reviews. Those feedbacks and modifications of the alpha testing phase are described below:

No	Feedback	Modification
1	Sometimes the search bar was not working	We fixed that later
2	Some variables and methods were not named well which made the code hard to understand	Such variables and methods were renamed to make the code more readable
3	There was no user profile picture adding feature	We added that later
4	We forgot to keep the option for updating or modifying ingredient amounts	After mock testing we have implanted it .

3.3.2 Beta Testing

The feedback gotten from mock clients (our friends) after finishing our system made us discover quite a few bugs and inconsistencies in our system and we had to make some modifications and corrections based on those reviews. Those feedbacks and modifications of the beta testing phase are described below:

No	Feedback	Modification
1	The app was not loading properly.	This was a compatibility issue at the user end , because the user already had using the port number that was allocated for the app in some other process.
2	The search bar was a bit buggy not being flexible and responsive	The problem was fixed by making necessary changes
3	User was able to manage the canteens without being managers	We fixed that