UNIVERSITY OF PLYMOUTH

PUSL3120
FULL-STACK DEVELOPMENT

FOOD ZONE (E commerce system)

Module leader: Dr. Mark Dixon

Group number: 98

GitHub link: https://github.com/Plymouth-University/coursework-group_98.git

You Tube link: https://youtu.be/UNPlMxCHUjs

Group members

Plymouth name	Plymouth index
Akurgoda Dilmith	10898448
Laddu Silva	10898666
Aluthgama Shamentha	10899699
Diwankara Gamage Tharusha	10898427
Wijekoon Wijekoon	10898706
Vithanage Bandara	10818157

Table of Contents

1. Requirements

1.1 Who is the application aimed at?

Our application, Food Zone, is aimed at consumers seeking a convenient and efficient online platform for grocery and food shopping. It is designed to serve:

- General Consumers**:** Individuals looking for quick, reliable, and user-friendly access to daily essentials, groceries, and other food-related products.
- Admin Users**:** Store managers or administrators who need tools to manage product listings, categories, and inventory efficiently.
- Retailers and Vendors: Businesses aiming to sell their products online with support for seamless uploads and inventory management.

1.2 What functionality was included and why?

Product Uploads:
Allows vendors and administrators to add products to the platform, ensuring an up-to-date and diverse inventory for consumers.

Admin Panel:
Provides tools for managing categories, subcategories, and products, giving administrators full control over platform content.

Category and Subcategory Management:
Facilitates easy organization and navigation of products, enhancing the user experience for shoppers.

Secure User Authentication (Access and Refresh Tokens):
Ensures that user data is protected and maintains secure session handling for logged-in users.

CI/CD Pipeline with GitHub Actions:
Streamlines development workflows, enabling rapid updates and deployments while ensuring code quality and minimizing downtime.

Cloudinary for Storage:
Manages image uploads for product listings efficiently, ensuring high-quality visuals with optimized performance.

JEST for testing: JavaScript testing framework. Helps developers write and run tests to ensure their code works as expected, improving code quality and reliability.

2. Design

Our system follows a client-server architecture based on the MERN stack (MongoDB, Express, React, and Node.js). The client acts as the user interface, providing interactive features through React, while the server processes requests, implements business logic, and interacts with the MongoDB database.

The client and server communicate through RESTful APIs, ensuring seamless interaction between frontend and backend. The server is hosted on Vercel, as indicated by the inclusion of the vercel.json file and handles secure data transactions using middleware for authentication and authorization.

2.1 Main Components and Their Interaction

1. Client (Frontend)

   The client is responsible for rendering the user interface and enabling users to interact with the application. Key components and pages include:
   - Components:
     - Examples such as CategoryWiseProductDisplay.jsx and CartMobile.jsx provide specific functionalities like displaying products by category and managing the cart on mobile devices.
     - Admin-specific components like ProductCardAdmin.jsx enable managing inventory directly from the UI.
   - Pages:
     - Pages like Home.jsx, ProductListPage.jsx, and CheckoutPage.jsx represent specific application routes. These pages aggregate components and allow users to browse products, manage their cart, and complete purchases.
   - Other Folders:
     - Assets, Common, Hooks, Provider, Store, Utils serve as resources for styling, state management, and context provision, enabling smoother and more efficient application functionality.

Interaction: The client sends requests to the server to fetch or manipulate data, such as fetching product details, managing the shopping cart, or processing user authentication.

2. Server (Backend)

   The server handles API requests, processes business logic, and manages data persistence. Key components include:

   - Config: Files such as connectDB.js and stripe.js handle database connections and payment gateway configurations, ensuring secure and reliable operations.
   - Controllers: These files, including product.controller.js and cart.controller.js, contain the logic for handling specific requests such as fetching products or updating cart items.
   - Middleware: Authentication (auth.js) and admin role validation (Admin.js) middleware ensure that sensitive operations are protected.
   - Models: The MongoDB schemas, such as product.model.js and order.model.js, structure the data stored in the database.
   - Routes: Define API endpoints such as product.route.js and cart.route.js that link specific requests to the appropriate controllers.

Interaction: When a client makes a request (e.g., adding a product to the cart), the server processes it by validating inputs, interacting with the database through models, and returning a response.

2.3 Data and Code Structure

Below is the UML diagram to demonstrate the design and code architecture.

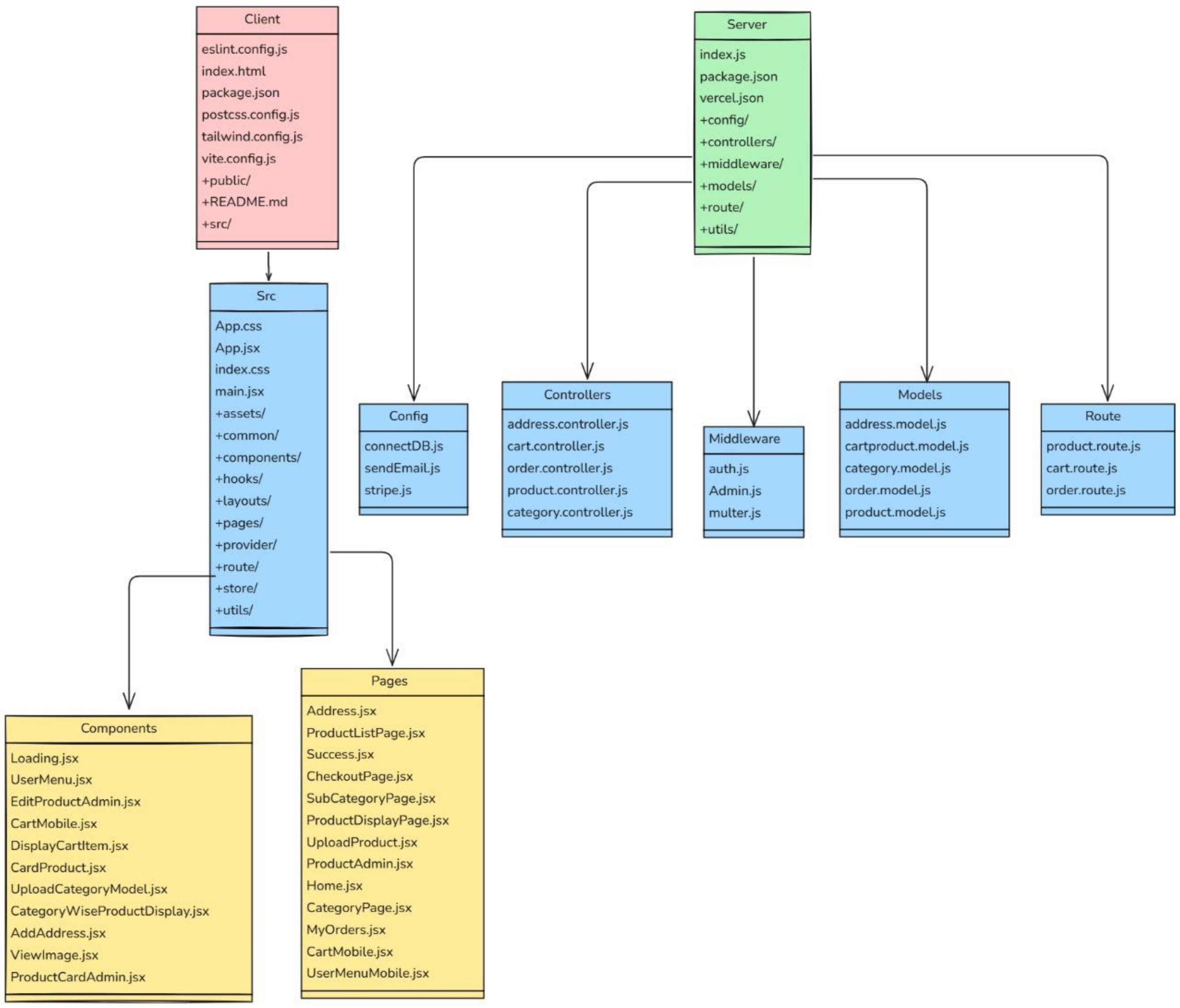


Figure 1

The application is organized to promote modularity, maintainability, and scalability:

1. Frontend Structure:

   - Components:
     Contains reusable UI elements like buttons, product displays, and modals.

   - Pages:
     Defines route-specific views, grouping related components for each major application feature (e.g., product pages, checkout, and user orders).

   - State Management:
     The store folder centralizes global state, ensuring consistent data flow throughout the application.

   - Utilities:
     Common helper functions (in utils) improve code efficiency and reduce duplication.

2. Backend Structure:

   - Separation of Concerns:
     Configurations, middleware, models, and controllers are isolated into specific folders, ensuring a clear division of responsibilities.

   - Mongoose Models:
     Define the structure and behavior of MongoDB collections, making database operations straightforward and efficient.

   - Middleware:
     Ensures route-specific logic (e.g., role-based access control) is centralized for easy updates and scalability.

2.3 Why These Structures Are Appropriate

The chosen architecture and code organization align with modern web development practices, ensuring robustness, maintainability, and scalability.

- Client-Server Architecture:
  This separation enables independent development and scaling of the frontend and backend.

- Modular Structure:
  Isolating components, pages, and utilities makes the codebase easier to manage and extend.

- RESTful APIs:
  These ensure a standardized communication protocol between the client and server, facilitating easy integration with third-party services.

- Middleware and Models:
  Middleware centralizes authentication and validation, improving security and maintainability. Mongoose models enforce data consistency in the database.

3. Testing

3.1 Manual Testing

Manual testing focused on validating core functionalities of the system to ensure a seamless user experience. Key manual tests included:

1. Add Product to Cart: Verified the correct addition of products to the cart. This test checked the product name, price, and quantity.
2. Checkout Process: Tested the complete checkout flow, from selecting an address to placing an order, ensuring proper error handling for incomplete details.
3. Admin Product Upload: Confirmed that admins could upload products with all mandatory fields and images, with validation for incorrect data input.
4. Responsive Design: Conducted device-specific testing to validate the UI across desktop, tablet, and mobile. Layout adjustments and interactive components like buttons and forms were inspected.

Below is the test case regarding the manual testing.

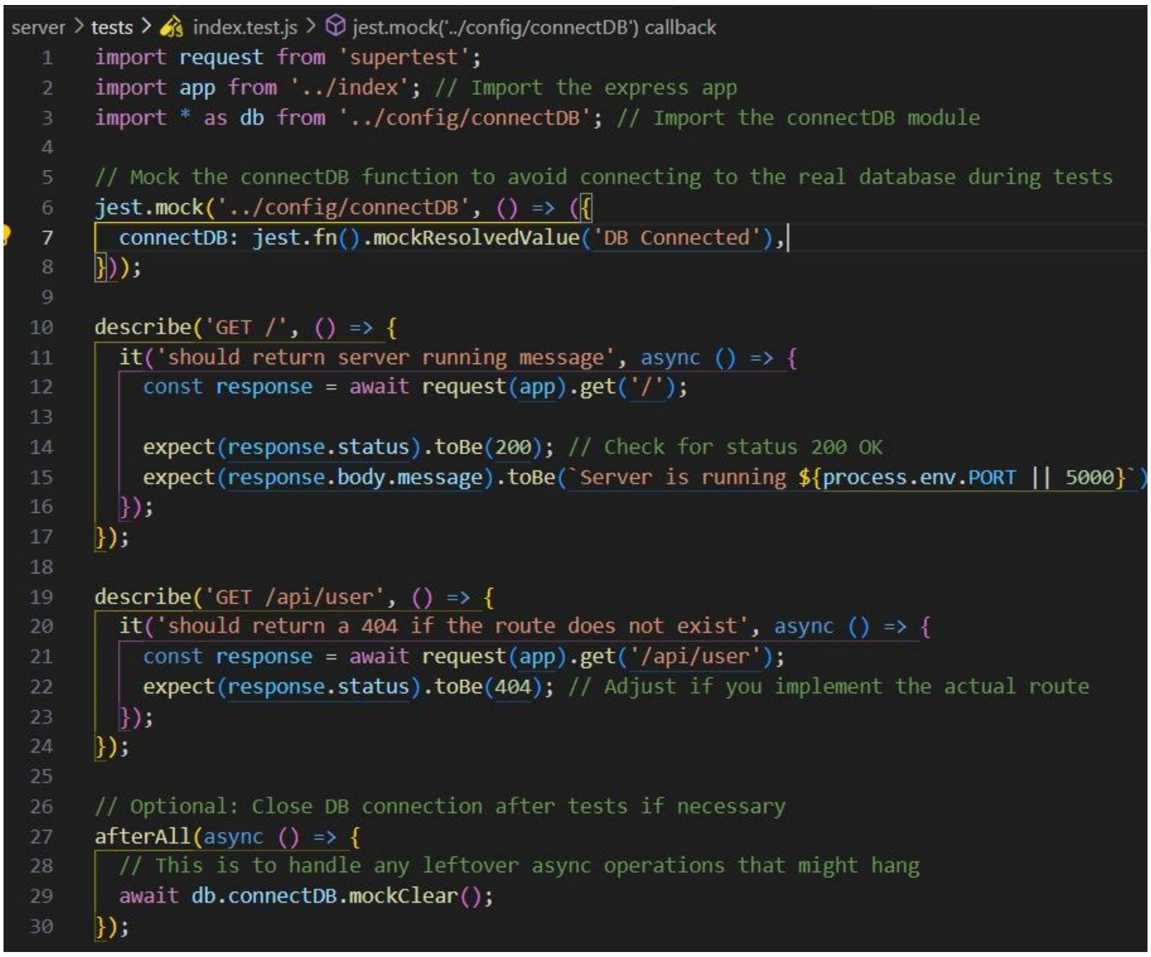
Test Case ID	Test Name	Test Description	Preconditions	Test Steps	Expected Result	Actual Result	Status
TC001	Add Product to Cart	Ensure a user can add products to their shopping cart.	User is logged in and navigates to the product page.	1. Open a product page. 2. Click the "Add to Cart" button. 3. Verify that the product appears in the cart.	Product is added to the cart with correct details (name, price, quantity).	Product added successfully.	Pass
TC002	Validate Checkout Functionality	Ensure the checkout process works, including address selection, order summary display, and successful order placement.	User is logged in and has at least one product in the cart.	1. Navigate to the "Cart" page. 2. Click "Checkout". 3. Select an address. 4. Review the order summary. 5. Confirm the order.	Order is placed successfully, and a confirmation message is displayed.	Order placed successfully.	Pass

TC003	Product Upload (Admin)	Verify that an admin can upload a new product with valid details.	User is logged in as an admin.	1. Navigate to the "Admin Panel". 2. Click "Add Product". 3. Enter product details. 4. Submit the form.	Product is added to the database and displayed on the product listing page.	Product uploaded successfully.	Pass
TC004	Search Bar Functionality	Ensure the search bar retrieves relevant products based on the user's query.	The product database contains searchable data.	1. Navigate to the homepage. 2. Enter a product name in the search bar. 3. Press "Search".	Search results display products matching the query.	Search returned correct results.	Pass
TC005	Responsive Design	Verify that the platform's UI is responsive across different screen sizes (desktop, tablet, mobile).	System supports responsive design.	1. Open the application on desktop, tablet, and mobile devices. 2. Inspect layout and functionality. 3. Resize the browser window to test responsiveness.	UI adjusts seamlessly for all screen sizes with no overlapping or layout issues.	UI is responsive across devices.	Pass

Table 1

3.2 Automated Testing

Automated testing was implemented using Jest for unit tests and Postman for API testing. Below are examples of the most significant automated tests:


server > tests > index.test.js > jest.mock('../config/connectDB') callback

```javascript
import request from 'supertest';
import app from '../index'; // Import the express app
import * as db from '../config/connectDB'; // Import the connectDB module

// Mock the connectDB function to avoid connecting to the real database during tests
jest.mock('../config/connectDB', () => ({
  connectDB: jest.fn().mockResolvedValue('DB Connected'),
}));

describe('GET /', () => {
  it('should return server running message', async () => {
    const response = await request(app).get('/');

    expect(response.status).toBe(200); // Check for status 200 OK
    expect(response.body.message).toBe(`Server is running ${process.env.PORT || 5000}`)
  });
});

describe('GET /api/user', () => {
  it('should return a 404 if the route does not exist', async () => {
    const response = await request(app).get('/api/user');
    expect(response.status).toBe(404); // Adjust if you implement the actual route
  });
});

// Optional: Close DB connection after tests if necessary
afterAll(async () => {
  // This is to handle any leftover async operations that might hang
  await db.connectDB.mockClear();
});
```

The code is a unit test suite for an Express app using Jest and Supertest. It tests the server's behavior without connecting to the real database by mocking the connectDB function.

Key Points:
Mocking Database:

connectDB is mocked to avoid actual database connections, resolving with 'DB Connected'.
Test Cases:

GET /: Verifies that the root route (/) responds with a status 200 and a message like "Server is running [PORT]".
GET /api/user: Ensures that a non-existent route (/api/user) returns a 404 status.
Cleanup:

Clears the mock for connectDB after all tests to ensure no leftover async operations.
Purpose:
Test the server routes in isolation from external dependencies (like the database).

3.3 Usability Testing

Usability testing was conducted to assess the system's intuitiveness and user satisfaction.

1. Participants: A total of 10 participants were involved, including five potential users and five testers unfamiliar with the system.
2. Process: Participants were tasked with completing common actions like browsing products, adding items to the cart, and completing a purchase. Their feedback was recorded on:
    - Ease of navigation.
    - Clarity of buttons and forms.
    - Response time.
3. Results: Feedback highlighted the following:
    - Strengths: Intuitive navigation and fast response times.
    - Issues: Initial difficulty in locating the search bar on mobile devices.
4. Modifications: Based on the feedback, the mobile UI was adjusted to place the search bar prominently at the top of the screen.

4. DevOps pipeline

4.1 Development environment

The development environment for the e-commerce project is designed using the MERN stack (MongoDB, Express.js, React, and Node.js). This modern framework ensures a robust and scalable architecture for both frontend and backend operations.

- Frontend: The user interface is built using React, styled with Tailwind CSS, and optimized with Vite. React enables the creation of reusable components, while Tailwind CSS allows for streamlined and consistent styling across the application. Vite ensures fast development and build processes, enhancing productivity.
- Backend: The server-side functionality is implemented with Node.js and Express.js, leveraging MongoDB as the database. Mongoose, an Object Data Modeling (ODM) library, is utilized for managing MongoDB operations.
- Version Control: The project uses Git for version control, with a GitHub repository hosting the code. The branches include dev for ongoing development and main for production-ready code.
- Testing: The backend is tested using Jest, a powerful testing framework that supports unit and integration tests. Jest ensures that all server-side functionalities are validated.
- Hosting and Deployment: The project is deployed on Vercel, which provides efficient hosting and integration with the CI/CD pipeline. Critical environment variables, such as MONGO_URI for the database connection and VERCEL_TOKEN for deployment, are securely managed using GitHub Secrets.

This structured environment enables effective collaboration, streamlined development, and reliable deployment.

4.2 Continuous Integration and Deployment Pipeline

The CI/CD pipeline is implemented using GitHub Actions, facilitating automated testing and deployment workflows. This pipeline ensures high-quality code and seamless updates to production.

Continuous Integration (CI)

The CI process is triggered by pushes to the dev or main branches and pull requests to main. The steps include:

1. Code Checkout: The latest code is fetched using actions/checkout@v3.
2. Node.js Setup: Node.js version 17 or higher is installed for compatibility with the project.
3. Dependency Installation: The backend dependencies are installed using npm install --prefix server.
4. Environment Configuration: MongoDB URI is configured as an environment variable using GitHub Secrets.
5. Testing Environment: Jest is installed for running server tests.
6. Test Execution: All backend tests are run using Jest to validate the code.

Continuous Deployment (CD)

The deployment workflow is triggered for the main branch after successful testing. Steps include:

1. Code Checkout: The latest version of the repository is pulled.
2. Vercel Setup: The Vercel CLI is installed for deployment.
3. Frontend Build: The React application is built using npm run build.
4. Production Deployment: The application is deployed to Vercel using the configured VERCEL_TOKEN.

5. Personal Reflection

This project was a challenging yet rewarding experience, providing valuable insights into full-stack development and project management.

5.1 What Worked Well

The MERN stack proved to be a robust choice for building the e-commerce platform. React's component-based architecture streamlined the front-end development, enabling a dynamic and responsive user interface. On the back end, Node.js and Express.js efficiently handled API requests, while MongoDB offered a flexible schema design suitable for the application's complex data requirements, such as products, categories, and user orders.

The implementation of the CI/CD pipeline using GitHub Actions was particularly effective in automating testing and deployment. Automated workflows ensured code integrity by running tests before deployment, reducing errors, and saving time. Additionally, integrating Cloudinary for image storage simplified the handling of product images, making the system scalable.

5.2 What Did not Work Well

Initially, the project faced challenges in implementing **secure authentication** using access and refresh tokens. Understanding token expiry and refresh workflows took more time than anticipated. Another issue arose with the responsiveness of the mobile interface, where the placement of certain UI elements caused confusion for users during usability testing. This required iterative adjustments based on feedback.

While Jest was effective for unit testing, the team struggled with setting up integration tests for some complex workflows, such as the checkout process. This highlighted the need for a better understanding of advanced testing techniques.

5.3 Lessons for Future Projects

1. Effective Planning: Early identification of potential bottlenecks, such as authentication complexities, could have saved time. In the future, allocating time for research on new techniques is essential.
2. Continuous Feedback: Usability testing played a critical role in improving the system. Incorporating regular user feedback earlier in the development cycle would enhance product design and usability.
3. Scalable Architecture: The use of modular design principles in both front-end and back-end development emphasized the importance of scalability, a lesson that will influence future projects.
4. Team Collaboration: Even in an individual project, leveraging tools like GitHub effectively showcased the value of organized version control and streamlined workflows.

6. Appendices

6.1 Appendix A: Functional requirements

1. Add to Cart Functionality:
   Allows users to add products to their shopping cart directly from the product listing or detail pages.

2. Loading Indicators:
   Displays loading animations or placeholders while data is being fetched or processed, improving user experience during delays.

3. Cart Management:
   Enables users to view, manage, and edit items in their shopping cart, supporting both desktop and mobile views.

4. Category-Wise Product Display:
   Displays products grouped by categories, allowing users to browse items based on specific categories for easier navigation.

5. Confirmation Dialogs:
   Provides confirmation prompts for critical actions such as deleting items, updating product information, or making other irreversible changes.

6. Product Display and Management:
   Offers options to view products in card or tabular formats and supports admin-specific functionality for editing or removing products from the inventory.

7. Category and Subcategory Management:
   Enables admins to create, update, or delete categories and subcategories, ensuring the product catalog is well-organized and up to date.

8. User Profile Management:
   Allows users to update their profile details, including personal information, address, and avatar images, to personalize their accounts.

9. Search Functionality:
   Provides a search feature to help users quickly find products or categories based on keywords.

10. Error and No Data Handling:
    Displays a message or graphic when no data is available, ensuring users are informed when searches or queries yield no results.

11. Image Viewing and Management:
    Provides the ability to view uploaded images, such as product photos or user profile pictures, in a full-size or preview mode.

12. Utility Features:
    Includes visual elements like dividers for better content organization and user menus for easier navigation and account management.