# E-COMMERCE WEBSITE DEVELOPMENT USING MERN STACK

**Smart Internz(Naan Mudhalvan)**

**PROJECT REPORT**

*Submitted by*

| | |
|---|---|
| **DHANUSH R** | **211121104012** |
| **RAMPRABAKAR J** | **211121104039** |
| **RANJITH S** | **211121104041** |
| **SARATHY A** | **211121104043** |
| **VIJAY E** | **211121104056** |

*In partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

**MADHA ENGINEERING COLLEGE**

Kundrathur, Chennai - 600069

# ABSTRACT

The E-Commerce Website using MERN Stack is a comprehensive, full-featured platform developed to provide an efficient and scalable online shopping experience. The project leverages the powerful MERN stack—MongoDB, Express.js, React, and Node.js—to create a modern and dynamic e-commerce solution. On the frontend, React offers a responsive and interactive user interface, ensuring seamless navigation and an intuitive shopping experience for customers. Features such as product search, category filters, personalized recommendations, and a smooth checkout process enhance the overall user experience, allowing customers to easily browse and purchase items. On the backend, Node.js and Express.js manage all server-side operations, such as handling requests, processing user data, managing session information, and interacting with the database. MongoDB, as the database solution, stores product information, user data, order histories, and transaction records in a flexible, scalable NoSQL format, making it easy to manage large amounts of data with high efficiency. This structure ensures that the system can handle high traffic volumes and frequent updates without performance degradation.

# TABLE OF CONTENTS

# LIST OF FIGURES

# 1.INTRODUCTION

Our e-commerce platform is built using the powerful MERN stack, which includes MongoDB, Express.js, React, and Node.js, ensuring a seamless and efficient online shopping experience. The use of **MongoDB** as the database allows us to store product catalogs, customer data, and transactions in a highly flexible and scalable NoSQL format. This enables fast data retrieval and easy expansion of the platform as the number of products and users grows. On the server side, **Express.js**, a lightweight Node.js framework, handles the API requests, routing, and middleware functionalities, ensuring smooth communication between the front-end and back-end. Express.js is designed to simplify the development of RESTful APIs, which are critical in handling user actions like product searches, adding items to the cart, and managing orders.

For the front-end, **React** is used to build a dynamic and responsive user interface that adapts to both desktop and mobile devices. React's component-based architecture allows us to create reusable UI components, enhancing development efficiency and ensuring a fast, interactive experience for users. The platform is optimized for speed and usability, providing a fluid, app-like experience that keeps customers engaged as they browse products, check out, and track their orders. On the back-end, **Node.js** provides the server-side environment, enabling quick processing of requests and high scalability. Node.js ensures that the platform remains performant even during high traffic periods, supporting rapid order processing and real-time updates.

By combining these technologies, we have created a secure, scalable, and user-friendly e-commerce solution that not only meets the demands of online shoppers but also provides flexibility for future growth and expansion. Whether you're looking for a simple product browse or a fast checkout process, the MERN stack enables an experience that is fast, reliable, and highly responsive.

The website offers secure user authentication, enabling customers to create accounts, log in, and manage their profiles. Admin users have access to additional features, including managing product listings, updating prices, processing orders, and handling customer feedback. The platform also integrates payment gateways like Stripe or PayPal for secure online transactions and order tracking in real-time.

## 1.1 Components of the MERN Stack in the Bookstore Application:

**MongoDB (Database):**

➢ MongoDB serves as the database where all book details, user information, and transactions are stored. MongoDB's document-based structure allows for flexible data storage, which is essential for storing various book attributes like titles, authors, genres, prices, and availability.

**Express.js (Backend Framework):**

➢ Express.js is used to build the backend server. It handles the communication between the front-end and the database. Express.js enables the API endpoints for user authentication, book management, and order processing.

➢ It helps route different HTTP requests (GET, POST, PUT, DELETE) and ensures data is served to the front-end efficiently.

**React.js (Frontend Library):**

➢ React.js powers the frontend of the bookstore application, offering a dynamic and responsive user interface. Users can browse books, filter results by categories or authors, add books to the shopping cart, and proceed with checkout.

➢ With state management libraries like Redux (or React's built-in context API), React ensures the state (e.g., user authentication, cart items) is consistent across different components.

**Node.js (Server-side JavaScript Runtime):**

➢ Node.js runs the backend server. As a JavaScript runtime, it allows the application to use JavaScript for both server-side and client-side code, providing consistency across the full stack.

➢ Node.js is known for its high performance and scalability, which is important when handling multiple user requests simultaneously in an e-commerce application.

# 2.PRE-REQUIREMENT

➢ **HTML, CSS, and JavaScript:** Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

➢ **Database Connectivity:** Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

➢ **Front-end Library**: Utilize React to build the user-facing part of the application, including products listings, booking forms, and user interfaces for the admin dashboard.

➢ **Version Control**: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

➢ **Git**: Download and installation instructions can be found at: https://git-scm.com/downloads

➢ **Development Environment:** Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

- Visual Studio Code: Download from https://code.visualstudio.com/download
- Sublime Text: Download from https://www.sublimetext.com/download
- WebStorm: Download from https://www.jetbrains.com/webstorm/download

**2.1 Tools and Technologies:**

1**. Node.js and npm**: Ensure Node.js and npm (Node Package Manager) are installed on your machine.

2. **MongoDB**: Install MongoDB and set up a local or cloud-based database.

3. **Text Editor/IDE**: Use a code editor like Visual Studio Code, which offers excellent support for JavaScript and React development.

4. **Version Control System**: Set up Git for version control and create a GitHub repository.

# 3.PROJECT OVERVIEW

The objective of this project is to develop a fully functional e-commerce website using the MERN stack, which includes MongoDB, Express.js, React, and Node.js. The website will serve as an online marketplace, offering a seamless and responsive shopping experience for users, from product browsing to secure checkout. The platform will be designed to scale efficiently as the number of users, products, and transactions grows.

## 3.1 Key Features:

1. **User Authentication & Authorization:**
    - Implement secure login and registration functionality using JWT (JSON Web Tokens) for authentication.
    - Users will have the ability to create accounts, log in, and maintain their sessions.
    - Admins will have special privileges for managing products, orders, and users.
2. **Product Catalog & Search**:
    - The platform will display a wide range of products, each with detailed descriptions, images, pricing, and availability.
    - Search functionality will be implemented, allowing users to filter products by category, price, ratings, and keywords. Users can view products individually, read reviews, and add them to their shopping cart.
3. **Shopping Cart & Checkout:**
    - Users can add products to their shopping cart and view/edit the cart before checkout.
    - A secure checkout process will allow users to enter shipping information, apply discount codes, and proceed with payment. Integration with payment gateways like Stripe or PayPal will ensure secure transaction handling.
4. **Order Management:**
    - Once a user completes a purchase, an order will be created and stored in the MongoDB database.
    - Admins can view and manage all orders, including changing the order status (pending, shipped, delivered).

# 4.ARCHITECTURE:

- **Technical Architecture:**

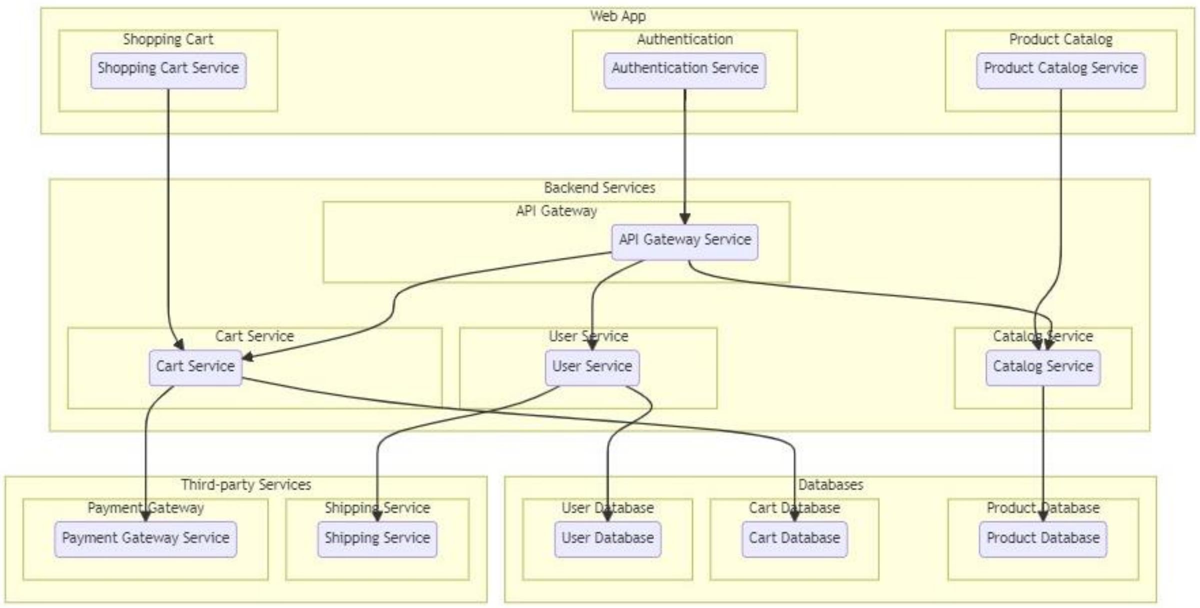


Fig 4.1 Technical Architecture

➢ **Frontend:**

The frontend is built using React.js, providing a responsive and dynamic user interface. React components manage the different pages and functions, allowing users to add Wishlist, order, and view dashboard. React, a popular front-end library, is used to build the user interface of the bookstore application.

➢ **Backend:**

Node.js and Express.js power the backend, handling user requests, authentication, Manages business logic, user authentication, and other backend services., and other business logic. The backend follows a RESTful API approach to communicate with the frontend. Express.js also seamlessly integrates with MongoDB, enabling efficient data handling and storage.

# 5.ER DIAGRAM

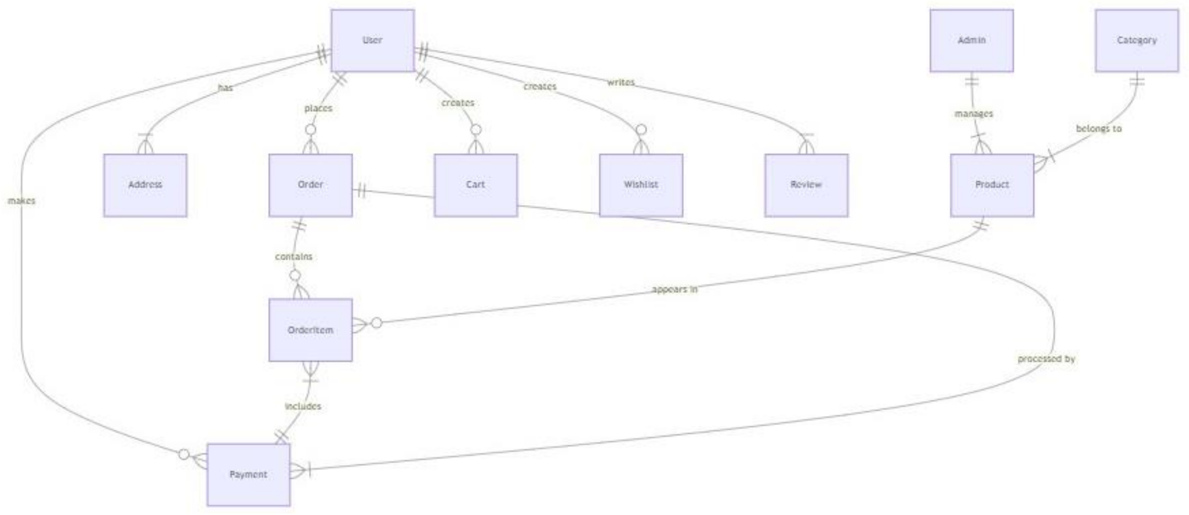


Fig 5.1 ER Diagram

The Entity-Relationship (ER) diagram for an flower and gift delivery app visually represents the relationships between different entities involved in the system, such as users, products, orders, and reviews. It illustrates how these entities are related to each other and helps in understanding the overall database structure and data flow within the application.

1. **Frontend Architecture (React.js)**

The front end is designed using React.js, a JavaScript library that allows for building reusable and responsive user interfaces. Key aspects include:

- **Component-Based Structure:**
  UI is divided into reusable components like Product Card, Navbar, and Cart. Each component handles specific tasks, improving maintainability.

- **State Management:**

  Context API or Redux is used for managing application state, such as user authentication and cart data.

- **Routing:**
  React Router manages navigation between pages, such as Home, Product Details, and Checkout, without full-page reloads.

- **Styling:**
  Styling is implemented using CSS, SCSS, or libraries like Material-UI or Bootstrap for responsiveness.

**Architecture:**

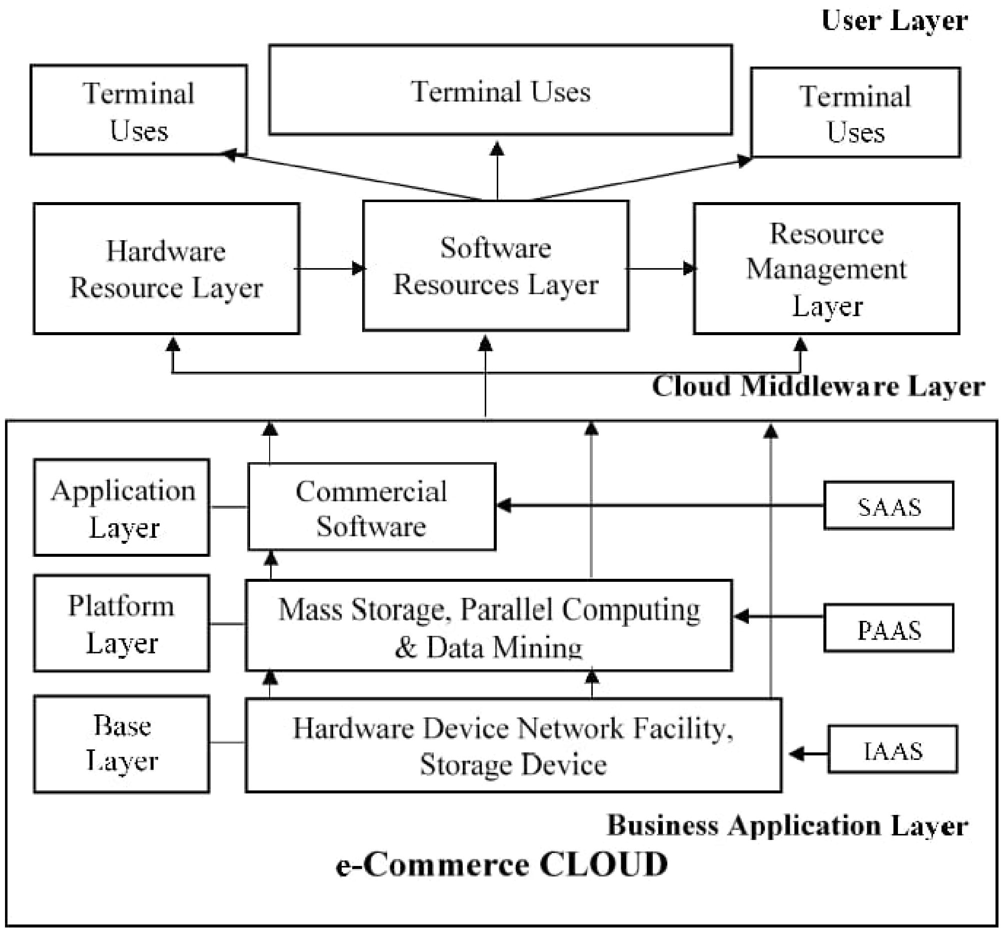


Fig 5.2 Architecture 1

2. **Backend Architecture (Node.js + Express.js)**

The backend handles business logic, API endpoints, and communication with the database. Key aspects include:

- **RESTful APIs:**

  APIs are built using Express.js to handle HTTP requests and responses.

  - **Example**
    - GET /api/products: Fetch all products.
    - POST /api/orders: Place an order.

- **Middleware:**

  Custom middleware is implemented for tasks like request validation, error handling, and authentication using JWT.

- **Scalability:**

  Node.js's asynchronous and event-driven nature ensures the backend can handle multiple concurrent requests.

**Architecture**

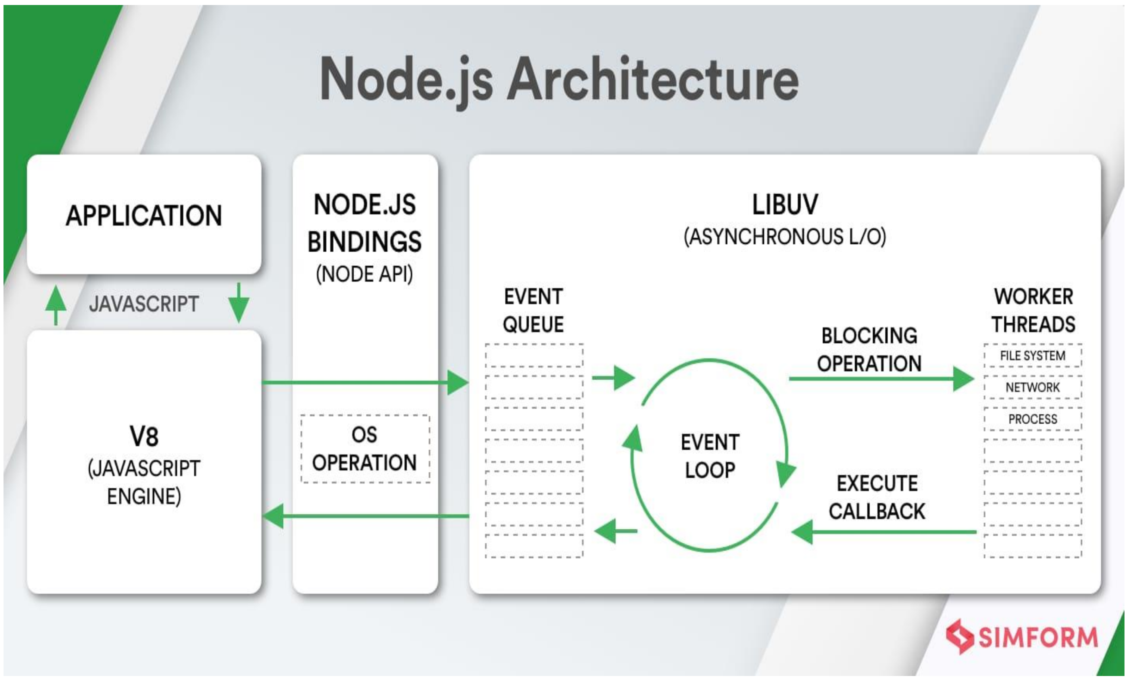


Fig 5.3 Architecture 2

3. **Database Architecture (MongoDB)**

MongoDB, a NoSQL database, is used for flexible and scalable data storage.

- **Collections:**
  - Users: Stores user credentials, profile data, and order history.
  - Products: Stores product details like name, price, category, and availability.

- Orders: Tracks orders, including user ID, product IDs, and order status.
- Cart: Temporary storage for a user's cart items.

• **Schema Design:**

Documents are structured to allow efficient querying and updates. For instance, orders store product references instead of embedding entire product details to minimize data duplication.

**Architecture**

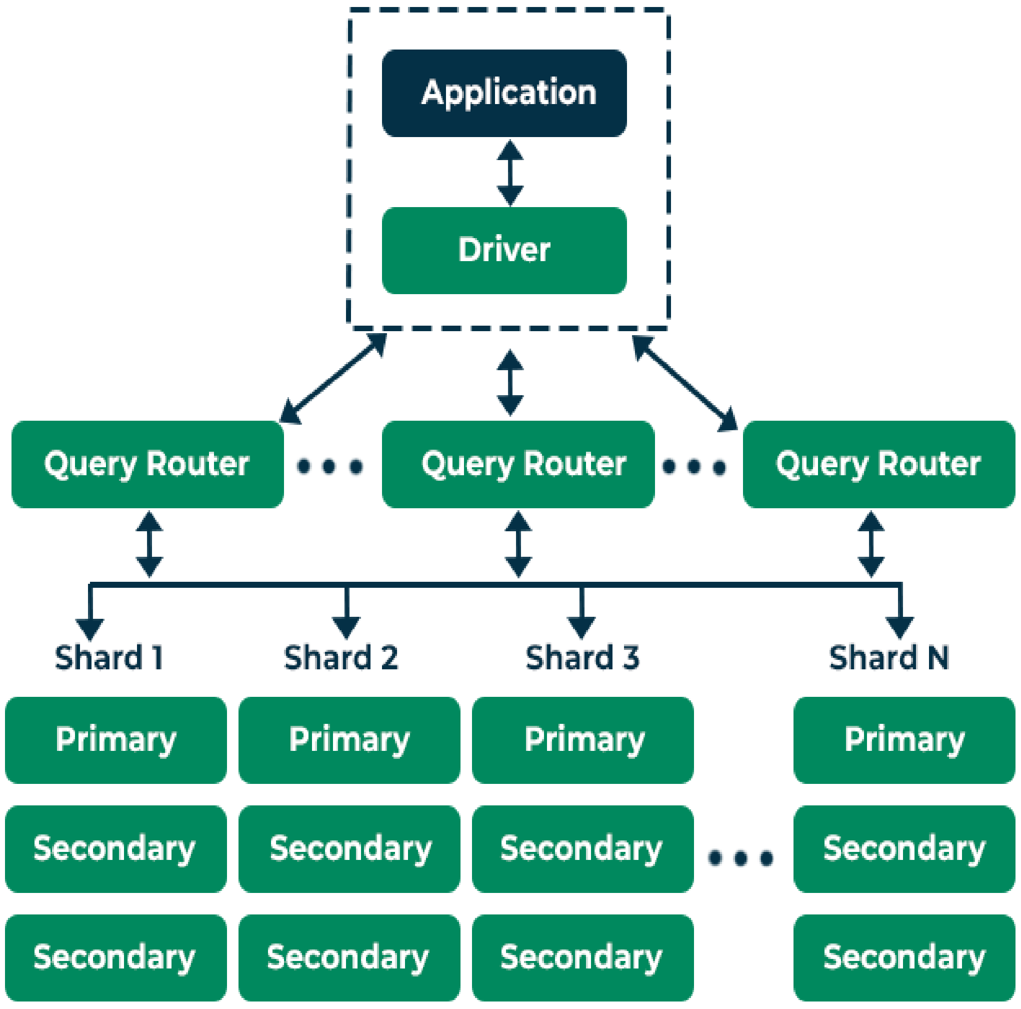


Fig 5.4 Architecture 3

# 6.SETUP INSTRUCTION

**Prerequisites:**

- Node.js (v14 or higher)
- MongoDB (locally or via a cloud service like MongoDB Atlas)
- Git
- npm or yarn

**Installation:**

**1. Clone the repository:**

```
git clone <repository-url>
cd <repository-folder>
```

**2. Install dependencies for both frontend and backend:**

```
cd client   npm install
cd ../server
npm install
```

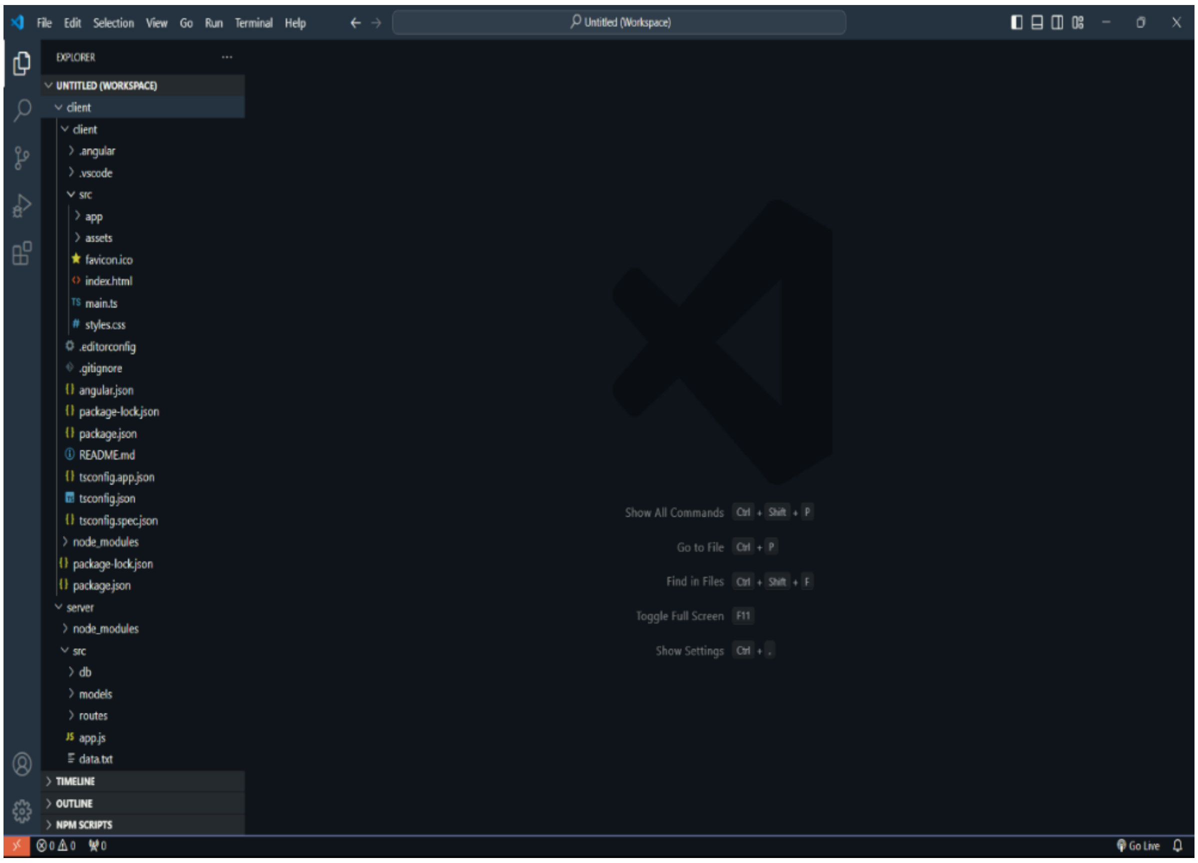
**3. Set up environment variables:**

- Create a .env file in the server directory.
- Add variables such as database connection string, JWT secret, etc.

4. **Implement API Endpoints:**

- **Create Routes Directory**: mkdir routes
- **Define API Routes**: Implement routes for user authentication, book management, order management, etc.
- **Set Up Middleware**: Implement middleware for authentication and authorization.

# 7.PROJECT STRUCTURE


File Edit Selection View Go Run Terminal Help
Untitled (Workspace)
EXPLORER
Show All Commands Ctrl + Shift + P
Go to File Ctrl + P
Find in Files Ctrl + Shift + F
Toggle Full Screen F11
Show Settings Ctrl + ,
Go Live

```
UNTITLED (WORKSPACE)
  client
    client
      .angular
      .vscode
      src
        app
        assets
        favicon.ico
        index.html
        main.ts
        styles.css
      .editorconfig
      .gitignore
      angular.json
      package-lock.json
      package.json
      README.md
      tsconfig.app.json
      tsconfig.json
      tsconfig.spec.json
    node_modules
    package-lock.json
    package.json
  server
    node_modules
    src
      db
      models
      routes
      app.js
      data.txt
TIMELINE
OUTLINE
NPM SCRIPTS
```

Fig 7.1 Foler Structure

This structure assumes an Angular app and follows a modular approach. Here's a brief explanation of the main directories and files:

- src/app/components: Contains components related to the customer app, such as register, login, home, products, my-cart, my-orders, place order, history, feedback, product-details, and more.
- src/app/modules: Contains modules for different sections of the app. In this case, the admin module is included with its own set of components like add-category, add-product, dashboard, feedback, home, orders, payment, update-product, users, and more.
- src/app/app-routing.module.ts: Defines the routing configuration for the app, specifying which components should be loaded for each route.
- src/app/app.component.ts, src/app/app.component.html, `src.

**Server:** **Client:**

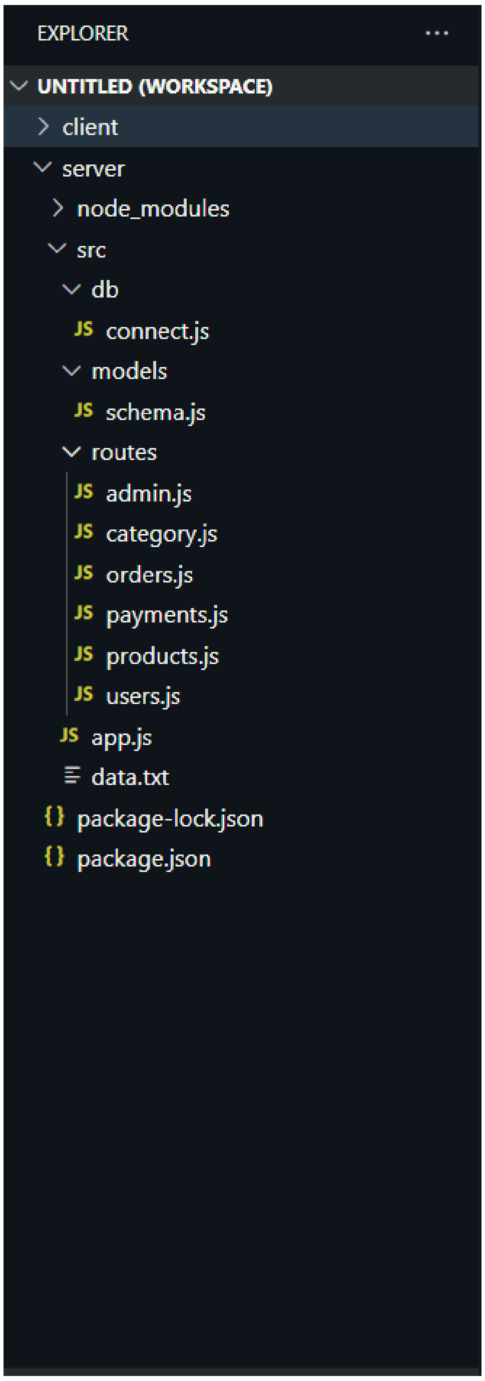

```
EXPLORER ...
UNTITLED (WORKSPACE)
  client
  server
    node_modules
    src
      db
        connect.js
      models
        schema.js
      routes
        admin.js
        category.js
        orders.js
        payments.js
        products.js
        users.js
      app.js
      data.txt
    package-lock.json
    package.json
```

Fig 7.2 Server Structure

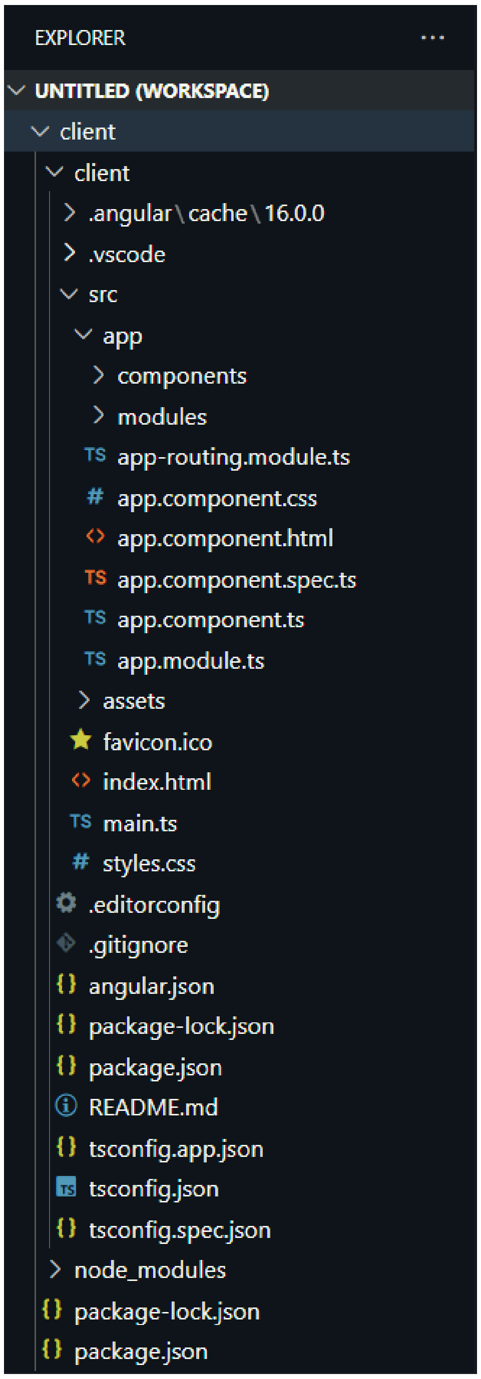

```
EXPLORER ...
UNTITLED (WORKSPACE)
  client
    client
      .angular\cache\16.0.0
      .vscode
      src
        app
          components
          modules
          app-routing.module.ts
          app.component.css
          app.component.html
          app.component.spec.ts
          app.component.ts
          app.module.ts
        assets
        favicon.ico
        index.html
        main.ts
        styles.css
      .editorconfig
      .gitignore
      angular.json
      package-lock.json
      package.json
      README.md
      tsconfig.app.json
      tsconfig.json
      tsconfig.spec.json
    node_modules
    package-lock.json
    package.json
```

Fig 7.3 Client Structure

# 8.RUNNING THE APPLICATION:

1. **Start the Backend**

    - The backend server is powered by Node.js and runs on the port specified in the .env file (default: 5000).
    - To navigate to backend use the command `cd backend` in terminal.
    - Navigate to the backend directory and run `npm start` to start the Express.js backend.

        1. Open a terminal.
        2. Navigate to the server directory:

```
cd server
```

        3. Start the backend server:

```
npm start
```

2. **Start the Frontend**

    - The frontend is built using React.js and runs on port 3000 by default.
    - To navigate to frontend use the command `cd frontend` in terminal.
    - Navigate to the frontend directory and run `npm run dev` to start the React application

        1. Open a new terminal.
        2. Navigate to the client directory:

```
cd client
```

3. **Start the frontend development server:**

```
npm start
```

4. The React application will open in your default browser. If it doesn't, you can manually open: **http://localhost:3000**

# 9.API DOCUMENTATION

## 1. User Authentication APIs

### a) Register User

- **Endpoint**: /api/users/register
- **Method**: POST
- **Description**: Registers a new user by saving their details in the database.
- **Request Body**:
    - name (string): Full name of the user.
    - email (string): Unique email address of the user.
    - password (string): Password for the account.
- **Response**:
    - Success: { message: "User registered successfully" }
    - Error: { error: "Email already exists" }

### b) Login User

- **Endpoint**: /api/users/login
- **Method**: POST
- **Description**: Authenticates a user and returns a token for session management.
- **Request Body**:
    - email (string): User’s email.
    - password (string): User’s password.
- **Response**:
    - Success: { token: "JWT_TOKEN", user: { id, name, email } }
    - Error: { error: "Invalid email or password" }

**c) Get User Profile**

- **Endpoint**: /api/users/profile
- **Method**: GET
- **Description**: Fetches the authenticated user's profile details.
- **Headers**:
    - Authorization (string): Bearer token.
- **Response**:
    - Success: { id, name, email, isAdmin }
    - Error: { error: "Unauthorized access" }

**2. Product Management APIs**

**a) Get All Products**

- **Endpoint**: /api/products
- **Method**: GET
- **Description**: Fetches a list of all products available on the platform.
- **Query Parameters**:
    - keyword (string, optional): Search keyword for filtering products.
    - category (string, optional): Filter by category.
    - priceRange (array, optional): Min and max price range.
- **Response**:
    - Success: { products: [ { id, name, description, price, category, stock, image } ] }
    - Error: { error: "Products not found" }

**b) Get Product Details**

- **Endpoint**: /api/products/:id
- **Method**: GET

- **Description**: Fetches details of a single product by its ID.
- **Response**:
    - Success: { id, name, description, price, category, stock, image }
    - Error: { error: "Product not found" }

**c) Add a Product (Admin Only)**

- **Endpoint**: /api/products
- **Method**: POST
- **Description**: Adds a new product to the catalog.
- **Headers**:
    - Authorization (string): Bearer token (Admin).
- **Request Body**:
    - name (string): Product name.
    - description (string): Product details.
    - price (number): Product price.
    - category (string): Product category.
    - stock (number): Available stock.
    - image (string): URL of the product image.
- **Response**:
    - Success: { message: "Product added successfully" }
    - Error: { error: "Unauthorized access" }

**3. Cart APIs**

**a) Add to Cart**

- **Endpoint**: /api/cart
- **Method**: POST

- **Description**: Adds a product to the user's cart.
- **Headers**:
    - Authorization (string): Bearer token.
- **Request Body**:
    - productId (string): ID of the product.
    - quantity (number): Quantity to add.
- **Response**:
    - Success: { message: "Product added to cart", cart: [...] }
    - Error: { error: "Failed to add product to cart" }

**b) Get Cart Items**

- **Endpoint**: /api/cart
- **Method**: GET
- **Description**: Fetches all products in the user's cart.
- **Headers**:
    - Authorization (string): Bearer token.
- **Response**:
    - Success: { cart: [ { productId, name, quantity, price } ] }
    - Error: { error: "Unauthorized access" }

**c) Remove from Cart**

- **Endpoint**: /api/cart/:productId
- **Method**: DELETE
- **Description**: Removes a product from the user's cart by its ID.
- **Headers**:
    - Authorization (string): Bearer token.

- **Response**:
    - Success: { message: "Product removed from cart" }
    - Error: { error: "Failed to remove product" }

**4. Order APIs**

**a) Place an Order**

- **Endpoint**: /api/orders
- **Method**: POST
- **Description**: Places an order for the items in the user's cart.
- **Headers**:
    - Authorization (string): Bearer token.
- **Request Body**:
    - cartItems (array): List of items in the cart.
    - shippingAddress (object): User’s address details.
    - paymentMethod (string): Payment method chosen (e.g., PayPal, Card).
- **Response**:
    - Success: { message: "Order placed successfully", orderId }
    - Error: { error: "Failed to place order" }

**b) Get User Orders**

- **Endpoint**: /api/orders/my-orders
- **Method**: GET
- **Description**: Fetches all orders placed by the authenticated user.

# 10.PACKAGES

**Home.jsx**

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { Router } from '@angular/router';
@Component({
  selector: 'app-landing-page',
  templateUrl: './landing-page.component.html',
  styleUrls: ['./landing-page.component.css']
})
export class LandingPageComponent {
  public data: any[] = [];
  public searchInput: string = '';
  public itemId: string;
  public product: any = {};
  public isLoading = false;
  electronicsSelected = '';
  clothesSelected = '';
  gadgetsSelected = false;
  regForm: FormGroup;
  constructor(private http: HttpClient, private route: Router) {
    this.isLoading = true;
    this.itemId = ''
    this.product = {}
    this.regForm = new FormGroup({
      user: new FormControl(null, Validators.required),
      phone: new FormControl(null, Validators.required),
      quantity: new FormControl(null, Validators.required),
      address: new FormControl(null, Validators.required),
      paymentMethod: new FormControl(null, Validators.required)
    })
    this.searchInput = '';
    this.http.get<any[]>('http://localhost:5100/products').subscribe(data => {
      this.data = data;
      this.isLoading = false
  });
  const jwtToken = localStorage.getItem('adminJwtToken')
  if (jwtToken) {
    window.alert("You can't Access this!")
    this.route.navigate(['/admin/home'])
  }
}

filterItems() {
  return this.data.filter(product => {
    const searchMatch =
      !this.searchInput ||
      product.productname.toLowerCase().includes(this.searchInput.toLowerCase());

    const categoryMatch =
      ((!this.electronicsSelected && !this.clothesSelected && !this.gadgetsSelected) ||
        (this.electronicsSelected && product.category === 'Mobile') ||
        (this.clothesSelected && product.category === 'Clothing') ||
        (this.gadgetsSelected && product.category === 'Shoes'));

    return searchMatch && categoryMatch;
  });
}

onAddToCart(productId: string): void {
  const token = localStorage.getItem("jwtToken")
  const jwtToken = localStorage.getItem('adminJwtToken')
  const userId = localStorage.getItem('userId')
  if (jwtToken) {
    this.route.navigate(['/admin/home'])
  } if (!token) {
    window.alert("You can't Access this! because your not an loggedin user!")
    this.route.navigate(['/login'])
  } else {
    this.http.post('http://localhost:5100/add-to-cart', {userId, productId}).subscribe(
      (response) => {
        window.alert('Product added to cart!');
      },
      (error) => {
        console.error(error);
        window.alert('Error occurred while adding the product to cart!');
      }
    );
  }
 }

 onBuyNow(orderDetails = { user: String, phone: String, productId: this.itemId,
address1: String, address2: String }): void {
   const token = localStorage.getItem("jwtToken")
   if (!token) {
     this.route.navigate(['/login'])
   } else {
     const order = {
       user: orderDetails.user,
       phone: orderDetails.phone,
       productId: this.itemId,
       address1: orderDetails.address1,
       address2: orderDetails.address2
     };
     this.http.post(`http://localhost:5100/order`, order).subscribe((res) => {
       if (res) {
         window.alert("Product Placed Successfully!");
         this.http.get<any[]>('http://localhost:5100/products').subscribe(data => {
           this.data = data;
         });
       }
     });
   }
 }
}
```

**Product Listing.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { ProductDetailsComponent } from './product-details.component';

describe('ProductDetailsComponent', () => {
  let component: ProductDetailsComponent;
  let fixture: ComponentFixture<ProductDetailsComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ProductDetailsComponent]
    });
    fixture = TestBed.createComponent(ProductDetailsComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

**Product Details.jsx**

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent {
  constructor(private router:Router){

  }

  onShop(){
    const token = localStorage.getItem('jwtToken')
    if(!token){
      this.router.navigate(['/login'])
    }else{
      this.router.navigate(['/shopping'])
    }
  }
}
```

**Cart Page.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { MyCartComponent } from './my-cart.component';

describe('MyCartComponent', () => {
  let component: MyCartComponent;
  let fixture: ComponentFixture<MyCartComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [MyCartComponent]
    });
    fixture = TestBed.createComponent(MyCartComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

**Login.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { LoginComponent } from './login.component';

describe('LoginComponent', () => {
 let component: LoginComponent;
 let fixture: ComponentFixture<LoginComponent>;

 beforeEach(() => {
  TestBed.configureTestingModule({
   declarations: [LoginComponent]
  });
  fixture = TestBed.createComponent(LoginComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
 });

 it('should create', () => {
  expect(component).toBeTruthy();
 });
});
```

**Signup.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { RegisterComponent } from './register.component';

describe('RegisterComponent', () => {
 let component: RegisterComponent;
 let fixture: ComponentFixture<RegisterComponent>;

 beforeEach(() => {
  TestBed.configureTestingModule({
   declarations: [RegisterComponent]
  });
```

```
    fixture = TestBed.createComponent(RegisterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

**Checkout.jsx**

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router'

@Component({
  selector: 'app-place-order',
  templateUrl: './place-order.component.html',
  styleUrls: ['./place-order.component.css']
})
export class PlaceOrderComponent {
  public data: any[] = [];
  public searchText: string;
  public routerId: string;
  public product: any = {};
  public isLoading = false;
  public isSuccess = false;

  regForm: FormGroup;

  constructor(private http: HttpClient, private route: Router, private activatedRoute: ActivatedRoute) {
    const jwtToken = localStorage.getItem('adminJwtToken')
    if (jwtToken) {
      this.route.navigate(['/admin/home'])
    }
    const token = localStorage.getItem("jwtToken")
    if (!token) {
      window.alert("You can't Access this! because your not an loggedin user!")
      this.route.navigate(['/login'])
    }
    this.routerId = '';
    const idParam = this.activatedRoute.snapshot.paramMap.get('id');
    if (idParam) {
      this.routerId = idParam;
    }

    this.product = {}
    this.regForm = new FormGroup({
      firstname: new FormControl(null, Validators.required),
      lastname: new FormControl(null, Validators.required),
      phone: new FormControl(null, Validators.required),
      quantity: new FormControl(null, Validators.required),
      address: new FormControl(null, Validators.required),
      paymentMethod: new FormControl(null, Validators.required)
    })
    this.searchText = '';
    this.http.get<any[]>('http://localhost:5100/products').subscribe(data => {
      this.data = data;
    });

  }

  filterData() {
    this.isLoading = true;
    if (this.searchText) {
      this.isLoading = false;
      return this.data.filter((product) =>
        product.productname.toLowerCase().includes(this.searchText.toLowerCase())
      );
    } else {
      this.isLoading = false;
      return this.data;
    }
  }

  onAddToCart(productId: string): void {
    this.isLoading = true;
    this.http.post('http://localhost:5100/add-to-cart', { "productId": productId }).subscribe((res) => {
      if (res) {
        window.alert("Product Added to cart!")
        this.isLoading = false;
      }else{
        this.isLoading = false;
      }
    })
  }

  createOrder(orderDetails = {firstname:String,lastname:String, phone: String, productId: this.routerId,
address: String, quantity: String, paymentMethod: String }): void {
    this.isLoading = true;
    const userId = localStorage.getItem('userId')
    const order = {
      firstname:orderDetails.firstname,
      lastname:orderDetails.lastname,
      user: userId,
      phone: orderDetails.phone,
      productId: this.routerId,
      quantity: orderDetails.quantity,
      address: orderDetails.address,
      paymentMethod: orderDetails.paymentMethod
    };
    console.log(order)
    this.http.post('http://localhost:5100/orders',order).subscribe((response) => {
      window.alert("Order Created Successfully!")
      this.isLoading = false;
      this.isSuccess = true
      this.regForm.reset()
    }, (error) => {
      window.alert("Failed to Create Order!")
      console.log(error);
      this.isLoading = false;
    });
  }

  onContinue(){
    this.isSuccess = false
  }
}
```

**Confirmation.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';


import { MyOrdersComponent } from './my-orders.component';


describe('MyOrdersComponent', () => {
 let component: MyOrdersComponent;
 let fixture: ComponentFixture<MyOrdersComponent>;


 beforeEach(() => {
  TestBed.configureTestingModule({
   declarations: [MyOrdersComponent]
  });
  fixture = TestBed.createComponent(MyOrdersComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
 });


 it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

**Admin Dashboard.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { AdminDashboardComponent } from './admin-dashboard.component';
describe('AdminDashboardComponent', () => {
  let component: AdminDashboardComponent;
  let fixture: ComponentFixture<AdminDashboardComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [AdminDashboardComponent]
    });
    fixture = TestBed.createComponent(AdminDashboardComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

**Admin Product List.jsx**

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { Router } from '@angular/router';

@Component({
  selector: 'app-add-products',
  templateUrl: './add-products.component.html',
  styleUrls: ['./add-products.component.css']
})
export class AddProductsComponent {
  regForm: FormGroup;
  public isLoading = false;

  constructor(private http: HttpClient, private route: Router) {
    this.regForm = new FormGroup({
      productname: new FormControl(null, Validators.required),
      description: new FormControl(null, Validators.required),
      price: new FormControl(null, Validators.required),
      brand: new FormControl(null, Validators.required),
      image: new FormControl(null, Validators.required),
      quantity: new FormControl(null, Validators.required),
      category: new FormControl(null, Validators.required),
      countInStock: new FormControl(null, Validators.required),
      rating: new FormControl(null, Validators.required),
    })
    const jwtToken = localStorage.getItem('adminJwtToken')
    if (!jwtToken){
      window.alert("You can't Access this!")
      this.route.navigate(['/login'])
    }

  }

  onSubmit(details = { productname: String, description: String, price: String, brand: String, image:
String, category: String, countInStock: String, rating: String }): void {
    this.isLoading = true;
    this.http.post('http://localhost:5100/add-products', details).subscribe((response) => {
      window.alert("Product Added Successfully!");
      this.regForm.reset();
      this.isLoading = false;
    });
  }


}
```

## Admin Category Management .jsx

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { FormGroup, FormControl,Validators } from '@angular/forms';
import { Router } from '@angular/router';

@Component({
  selector: 'app-add-categories',
  templateUrl: './add-categories.component.html',
  styleUrls: ['./add-categories.component.css']
})
export class AddCategoriesComponent {
  regForm: FormGroup;
  public isLoading = false;

  constructor(private http:HttpClient, private route:Router) {
    this.regForm  = new FormGroup({
      category:new FormControl(null,Validators.required),
    })
    const jwtToken = localStorage.getItem('adminJwtToken')
    if (!jwtToken){
     window.alert("You can't Access this!")
     this.route.navigate(['/login'])
    }
   }

   onSubmit(details={category:String}): void {
    this.isLoading = true;
    this.http.post('http://localhost:5100/add-category',details).subscribe((response) => {
     window.alert("Category Added Successfully!")
     this.regForm.reset();
     this.isLoading = false;
    })
  }
}
```

## Admin Order Management.jsx

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { Router } from '@angular/router';
import { NgbModal } from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'app-orders',
  templateUrl: './orders.component.html',
  styleUrls: ['./orders.component.css']
})
export class OrdersComponent {
  public data: any[] = [];
  public isLoading = false;
  public orderId: string = '';
  public isUpdate = false;

  statusForm: FormGroup;

  constructor(private http: HttpClient, private route: Router,private modalService: NgbModal) {

    // Email Message to Customer //



    // Email Message to Customer //
    this.isLoading = true;
    this.http.get<any[]>('http://localhost:5100/orders').subscribe(data => {
      this.data = data;
      this.isLoading = false;
    });
    const jwtToken = localStorage.getItem('adminJwtToken')
    if (!jwtToken){
```

```
    window.alert("You can't Access this!")
    this.route.navigate(['/login'])
   }
   this.statusForm  = new FormGroup({
    status:new FormControl('pending')
   })
  }

  onChangeStatus(id: string): void {
    this.orderId = id
    this.isUpdate = true
  }


  onSubmit(status:String):void{
   this.http.put(`http://localhost:5100/orders/${this.orderId}`,status).subscribe((res)=>{
    window.alert('Order Status Updated!')
    this.isUpdate = false
    this.http.get<any[]>('http://localhost:5100/orders').subscribe(data => {
    this.data = data;
    this.isLoading = false;
   });
   })
  }
}
```

## User Management .jsx

```
import { HttpClient } from '@angular/common/http';
import { Component , OnInit} from '@angular/core';

@Component({
  selector: 'app-users',
  templateUrl: './users.component.html',
  styleUrls: ['./users.component.css']
})
export class UsersComponent implements OnInit{
  users: any[] = [];
  public isLoading = false

  constructor(private http:HttpClient) { }

  ngOnInit() {
    this.fetchUsers();
  }

  fetchUsers() {
    this.isLoading = true
    this.http.get<any[]>('http://localhost:5100/users').subscribe(
      users => {
        this.users = users;
        this.isLoading = false
      },
      error => {
        console.error(error);
      }
    );
  }
}
```

**Update Product.jsx**

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { HttpClient } from '@angular/common/http';
@Component({
  selector: 'app-update-product',
  templateUrl: './update-product.component.html',
  styleUrls: ['./update-product.component.css']
})
export class UpdateProductComponent {
  public data: any[] = [];
  public isLoading = false;
  regForm: FormGroup;

  constructor(private router: Router, private http: HttpClient, private route: ActivatedRoute) {
    this.regForm = new FormGroup({
      productname: new FormControl(null, Validators.required),
      description: new FormControl(null, Validators.required),
      price: new FormControl(null, Validators.required),
      brand: new FormControl(null, Validators.required),
      image: new FormControl(null, Validators.required),
      quantity: new FormControl(null, Validators.required),
      category: new FormControl(null, Validators.required),
      countInStock: new FormControl(null, Validators.required),
      rating: new FormControl(null, Validators.required),
    });
    const jwtToken = localStorage.getItem('adminJwtToken');
    if (!jwtToken) {
      window.alert("You can't access this!");
      this.router.navigate(['/login']);
    }
  }

  onUpdate(productDetails: {
    productname: string;
    description: string;
    price: string;
    brand: string;
    image: string;
    category: string;
    countInStock: string;
    rating: string;
  }): void {
    this.isLoading = true;
    const productId = this.route.snapshot.paramMap.get('id');
    this.http
      .put(`http://localhost:5100/products/${productId}`, productDetails)
      .subscribe((res) => {
        if (res) {
          window.alert('Product Updated Successfully!');
          this.router.navigate(['/admin/home'])
          this.http
            .get<any[]>('http://localhost:5100/products')
            .subscribe((data) => {
              this.data = data;
              this.isLoading = false;
            });
        }
      });
  }
```

**Payment.jsx**

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-payment',
  templateUrl: './payment.component.html',
  styleUrls: ['./payment.component.css']
})
export class PaymentComponent {
  public data: any[] = [];
  public isUpdate = false;
  public isManage = false;
  public paymentId: String = '';

  statusForm: FormGroup;

  constructor(private http:HttpClient){
    this.http.get<any[]>('http://localhost:5100/payments').subscribe((response) => {
```

```
    this.data = response
  })
  this.statusForm = new FormGroup({
    status: new FormControl('pending'),
    amount: new FormControl(null, Validators.required)
  });

}

onChangeStatus(id:string):void{
  this.paymentId = id
  this.isUpdate = true;
}

onSubmit(paymentDetails={status:String,amount:String}): void {
  this.http.put(`http://localhost:5100/payment/${this.paymentId}`, paymentDetails).subscribe(
    (res) => {
      window.alert('Payment Status Updated!');
      this.isUpdate = false;
      this.http.get<any[]>('http://localhost:5100/payments').subscribe((response) => {
        this.data = response
      })
    },
    (error) => {
      console.error(error);
      window.alert('Failed to update Payment status');
    }
  );
}
}
```

**Feedback.jsx**

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';

@Component({
  selector: 'app-feedback',
  templateUrl: './feedback.component.html',
  styleUrls: ['./feedback.component.css']
})
export class FeedbackComponent {
 public data: any[] = [];
 public isLoading = false;

 constructor(private http:HttpClient){
   this.isLoading = true;
   this.http.get<any[]>('http://localhost:5100/feedback').subscribe(data => {
      this.data = data;
      this.isLoading = false;
    });
  }
}
```

**Footer.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { FooterComponent } from './footer.component';

describe('FooterComponent', () => {
  let component: FooterComponent;
  let fixture: ComponentFixture<FooterComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [FooterComponent]
    });
    fixture = TestBed.createComponent(FooterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

**Header.jsx**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { HeaderComponent } from './header.component';

describe('HeaderComponent', () => {
  let component: HeaderComponent;
  let fixture: ComponentFixture<HeaderComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [HeaderComponent]
    });
    fixture = TestBed.createComponent(HeaderComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

## Side bar.jsx

```
import { Component } from '@angular/core';
import { faBars, faTachometerAlt, faShoppingBag, faShoppingCart, faTags,
faMoneyBill,faComment,faUsers } from '@fortawesome/free-solid-svg-icons';

@Component({
  selector: 'app-sidebar',
  templateUrl: './sidebar.component.html',
  styleUrls: ['./sidebar.component.css']
})
export class SidebarComponent {
 public isSidebarHidden = false;
 public barIcon = faBars

 public data = [
    {
     path: '/admin/dashboard',
     icon: faTachometerAlt,
     name: 'Dashboard'
    },
    {
     path: '/admin/users',
     icon: faUsers,
     name: 'Users'
    },
    {
     path: '/admin/products',
     icon: faShoppingBag,
     name: 'Products'
    },
    {
     path: '/admin/add-products',
     icon: faShoppingCart,
     name: 'Add Products'
    },
    {
     path: '/admin/add-categories',
     icon: faTags,
     name: 'Add Category'
    },{
     path: '/admin/orders',
     icon: faShoppingCart,
     name: 'Orders'
    },
    {
     path: '/admin/payment',
     icon: faMoneyBill,
     name: 'Payment'
    },
    {
     path: '/admin/feedback',
     icon: faComment,
     name: 'Feedback'
    }
  ];

  constructor() {
  }

  toggleSidebar(): void {
    this.isSidebarHidden = !this.isSidebarHidden;
  }

}
```

**Admin Home.jsx**

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent {
 public data: any[] = [];
 public searchText: string;
 public isUpdate = false;
 public isLoading = false;


  constructor(private http: HttpClient, private route: Router) {
    this.isLoading = true;

    this.searchText = '';
    this.http.get<any[]>('http://localhost:5100/products').subscribe(data => {
      this.data = data;
      this.isLoading = false;
    });
    const jwtToken = localStorage.getItem('adminJwtToken')
    if (!jwtToken){
      window.alert("You can't Access this!")
      this.route.navigate(['/login'])
    }
  }
  filterData() {
    if (this.searchText) {
      return this.data.filter((product) =>
        product.productname.toLowerCase().includes(this.searchText.toLowerCase())
      );
    } else {
      return this.data;
    }
  }


  onDelete(productId: string): void {
    // this.isLoading = true;
    this.http.delete(`http://localhost:5100/products/${productId}`).subscribe((res) => {
      if (res) {
        window.alert("Product Deleted Successfully!")
        this.http.get<any[]>('http://localhost:5100/products').subscribe(data => {
          this.data = data;
          // this.isLoading = false;
        });
      }
    })
  }
}
```

## Backend Connection

### Payment.js

```
const app = require("../app");
const models = require("../models/schema");
// Create a payment
app.post('/api/admin/create-payment', async (req, res) => {
   try {
       const { userId, orderId, amount, paymentMethod, deliveryStatus } = req.body;

       // Check for missing required fields
       if (!userId || !orderId || !amount || !paymentMethod || !deliveryStatus) {
           return res.status(400).send({ message: 'Missing required fields' });
       }

       // Check if the order exists
       const foundOrder = await models.Order.findById(orderId);
       if (!foundOrder) {
           return res.status(404).send({ message: 'Order not found' });
       }
```

```
    // Create a new payment record
    const payment = new models.Payment({
      user: userId,
      order: orderId,
      amount,
      paymentMethod,
      deliveryStatus,
      status: 'Pending', // Default status is 'Pending'
    });

    // Save the payment to the database
    await payment.save();

    // Update the order status if payment is successful
    foundOrder.status = 'Confirmed'; // Optionally change order status after successful payment
    await foundOrder.save();

    // Return success response with the created payment
    res.status(201).send({ message: 'Payment created successfully', payment });
  } catch (error) {
    console.log(error);
    res.status(500).send({ message: 'Internal server error' });
  }
});

// Get all payments
app.get('/api/admin/get-payments', async (req, res) => {
  try {
    // Get all payments from the database
    const payments = await models.Payment.find().populate('user', 'firstname lastname
email').populate('order', 'productName price');
    res.status(200).send(payments);
  } catch (error) {
    console.log(error);
    res.status(500).send({ message: 'Internal server error' });
  }
});

// Get payment by ID
app.get('/api/admin/get-payment/:id', async (req, res) => {
  try {
    const { id } = req.params;

    // Find payment by ID
    const payment = await models.Payment.findById(id)
      .populate('user', 'firstname lastname email')
      .populate('order', 'productName price');
    if (!payment) {
      return res.status(404).send({ message: 'Payment not found' });
    }

    res.status(200).send(payment);
  } catch (error) {
    console.log(error);
    res.status(500).send({ message: 'Internal server error' });
  }
});

// Update payment status
app.put('/api/admin/update-payment/:id', async (req, res) => {
  try {
    const { id } = req.params;
    const { status, deliveryStatus } = req.body;

    // Find the payment to update
    const payment = await models.Payment.findById(id);
    if (!payment) {
      return res.status(404).send({ message: 'Payment not found' });
    }

    // Update payment fields
    if (status) payment.status = status; // 'Pending', 'Success', or 'Failed'
    if (deliveryStatus) payment.deliveryStatus = deliveryStatus; // 'Pending', 'Shipped', 'Delivered', etc.

    // Save the updated payment
    await payment.save();

    // Return success response
    res.status(200).send({ message: 'Payment updated successfully', payment });
  } catch (error) {
    console.log(error);
    res.status(500).send({ message: 'Internal server error' });
  }
});

// Delete payment
app.delete('/api/admin/delete-payment/:id', async (req, res) => {
  try {
    const { id } = req.params;

    // Find the payment to delete
    const payment = await models.Payment.findById(id);
    if (!payment) {
      return res.status(404).send({ message: 'Payment not found' });
    }

    // Delete the payment record
    await payment.remove();

    // Return success response
    res.status(200).send({ message: 'Payment deleted successfully' });
  } catch (error) {
    console.log(error);
    res.status(500).send({ message: 'Internal server error' });
  }
});

// Get payments by status
app.get('/api/admin/get-payments-by-status/:status', async (req, res) => {
  try {
    const { status } = req.params;

    // Find payments with a specific status
    const payments = await models.Payment.find({ status }).populate('user', 'firstname lastname
email').populate('order', 'productName price');
    res.status(200).send(payments);
  } catch (error) {
    console.log(error);
    res.status(500).send({ message: 'Internal server error' });
  }
});
```

# 11.AUTHENTICATION

Authentication is a critical component of an e-commerce website as it ensures secure user access to personalized features like account management, order history, and shopping cart persistence. Below is an explanation of how authentication can be implemented in a MERN stack application

**1. Authentication Flow**

The authentication process involves **user registration**, **login**, **protected routes**, and **logout**. It ensures that only authorized users can perform specific actions, such as viewing their orders or making purchases.

**2. Components of Authentication**

1. **User Registration**:
    - A user provides their name, email, and password.
    - The password is hashed using a library like **bcrypt** for secure storage in the database.
    - The hashed password and user details are saved in **MongoDB**.
2. **User Login**:
    - The user submits their email and password.
    - The server verifies the credentials by comparing the entered password with the stored hashed password.
    - Upon successful login, the server generates a **JSON Web Token (JWT)**, which serves as a session identifier.
3. **Token-Based Authentication**:
    - After login, the server sends the JWT to the client.
    - The client stores the token (usually in **localStorage** or **cookies**).
    - The token is sent with every protected request in the **Authorization header** as Bearer <token>.
4. **Protected Routes**:
    - Certain API routes, like fetching user profile, cart, or orders, require authentication.

- The server verifies the token using a library like **jsonwebtoken**. If valid, it grants access to the requested resource.

5. **User Logout**:
    - The client deletes the token from storage to log out the user.
    - Optionally, tokens can be invalidated server-side using blacklists.

**3. Security Best Practices**

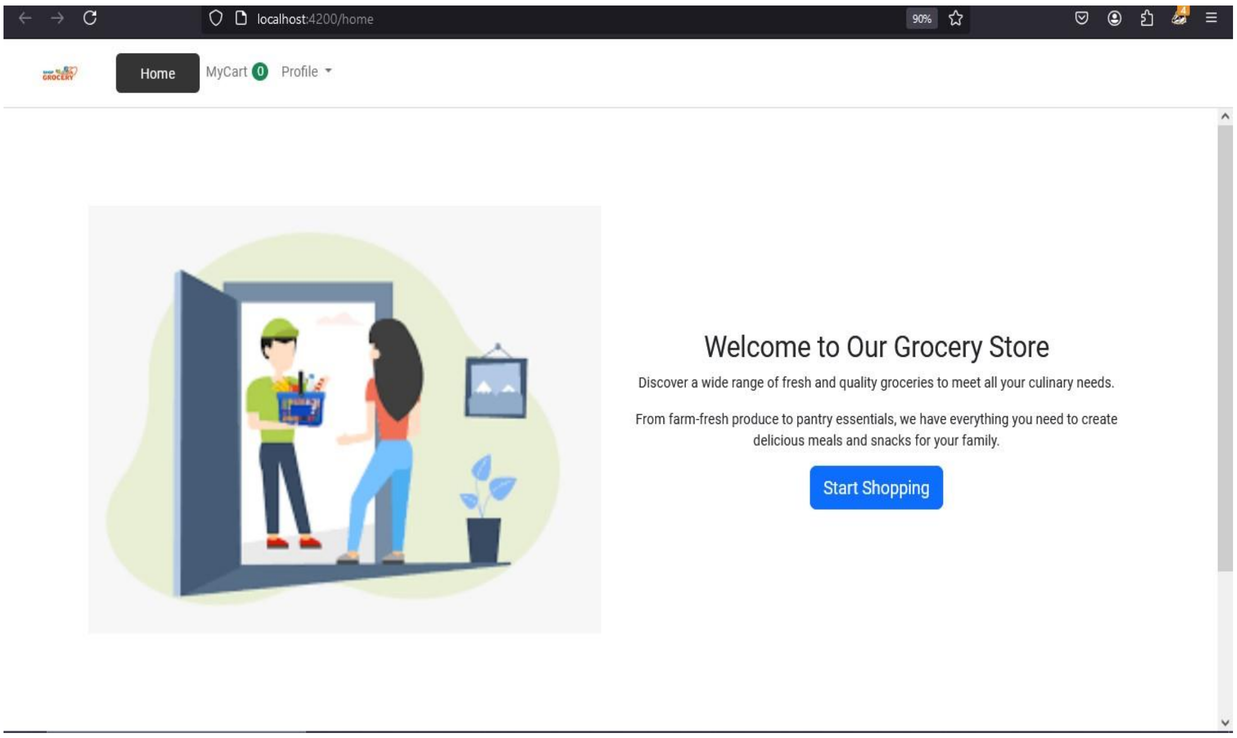
1. **Password Hashing**:
    - Use **bcrypt** to hash passwords before storing them in the database.
    - Avoid storing plain-text passwords.
2. **JWT Security**:
    - Use a strong secret key for signing JWTs.
    - Set an expiration time for tokens (e.g., 1 hour) to limit misuse.
3. **HTTPS**:
    - All data exchange between the client and server should use **HTTPS** to encrypt communication.
4. **Validation**:
    - Validate user inputs (email format, password length, etc.) to prevent injection attacks.
5. **Environment Variables**:
    - Store sensitive data like the JWT secret key and database credentials in environment variables.
6. **Rate Limiting**:
    - Implement rate-limiting to prevent brute-force attacks on login endpoints.

**4. User Roles and Authorization**

1. **Role-Based Access Control (RBAC)**:
    - Users can have roles like "customer" or "admin".
    - Admins can access additional functionalities, such as managing products and orders.
    - Assign roles during user registration or later using an admin panel.
2. **Middleware for Role Checking**:
    - Implement middleware to verify a user’s role before granting access to admin-only routes.

# 12.USER INTERFACE & SCREENSHOTS

## Home Page

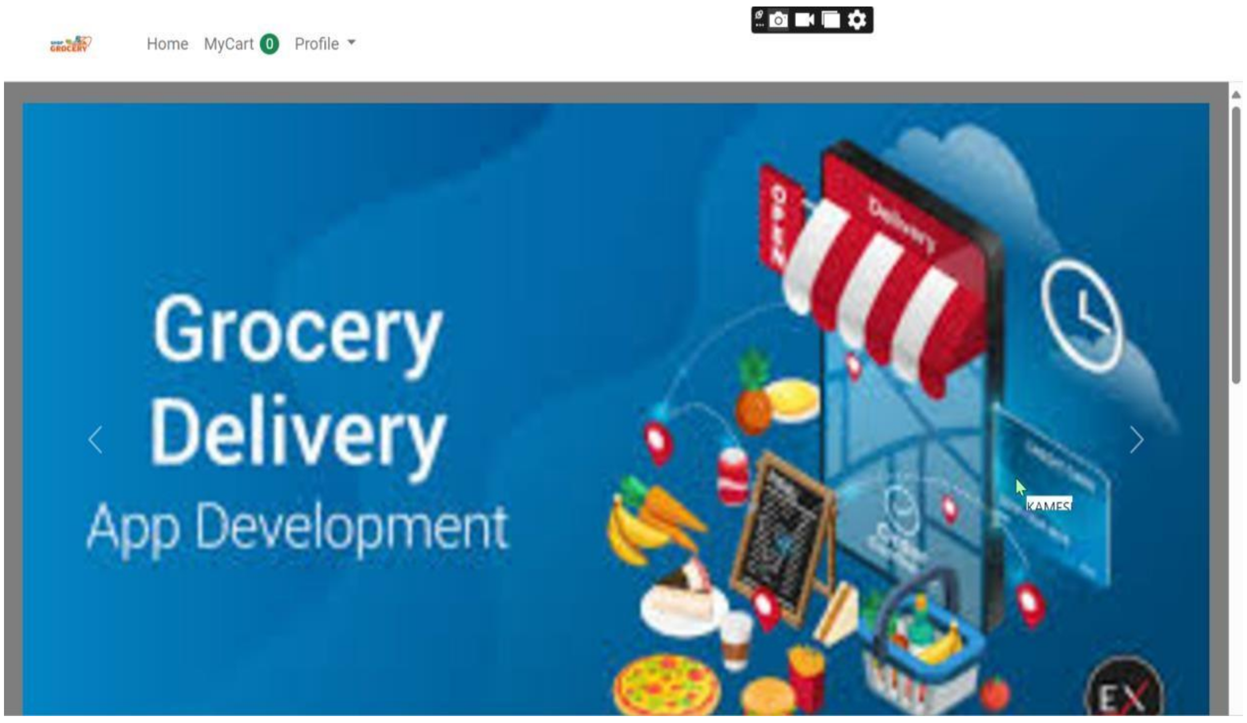


Welcome to Our Grocery Store

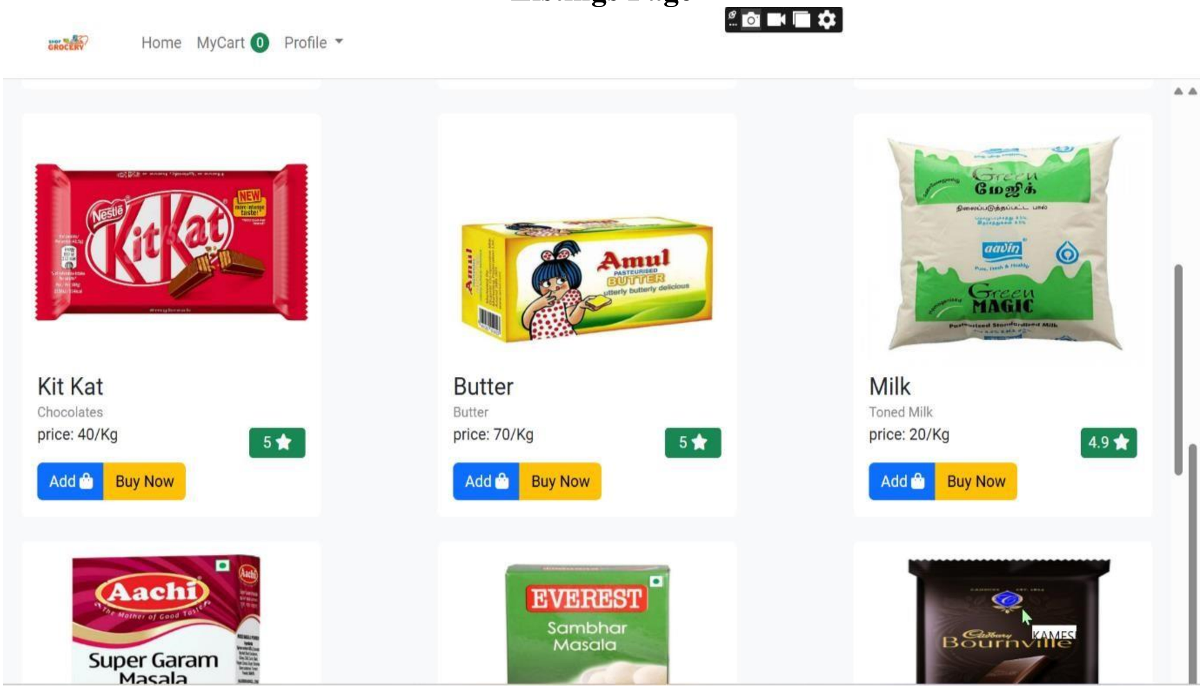
Discover a wide range of fresh and quality groceries to meet all your culinary needs.

From farm-fresh produce to pantry essentials, we have everything you need to create delicious meals and snacks for your family.

## Landing Page



## Listings Page

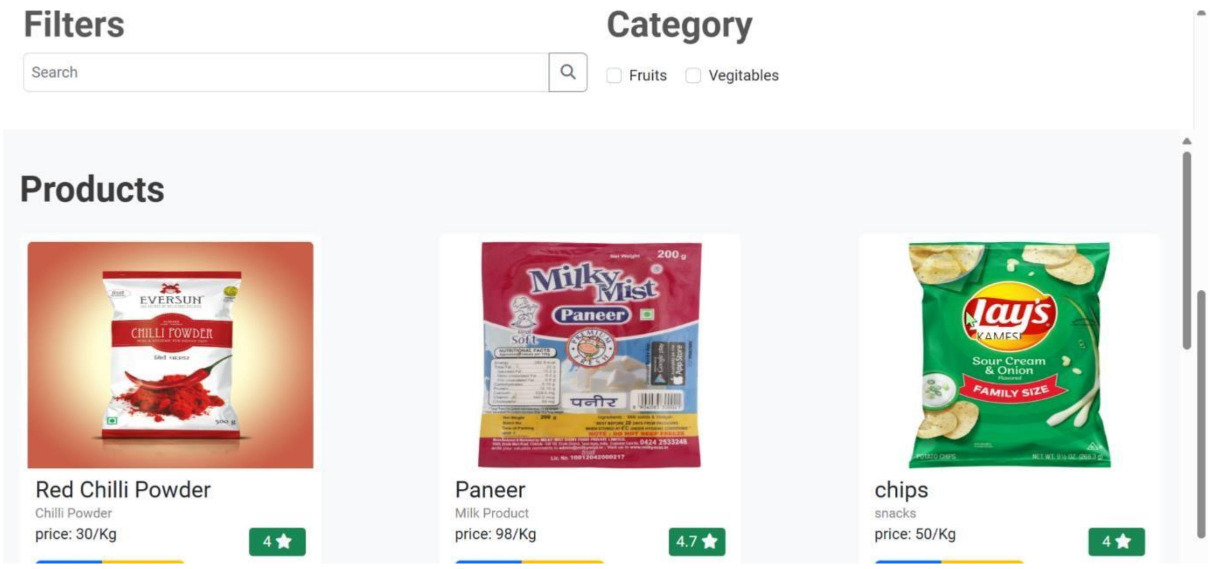


Kit Kat
Chocolates
price: 40/Kg
5 ★
Add | Buy Now

Butter
Butter
price: 70/Kg
5 ★
Add | Buy Now

Milk
Toned Milk
price: 20/Kg
4.9 ★
Add | Buy Now

## Category Page


Filters

Category

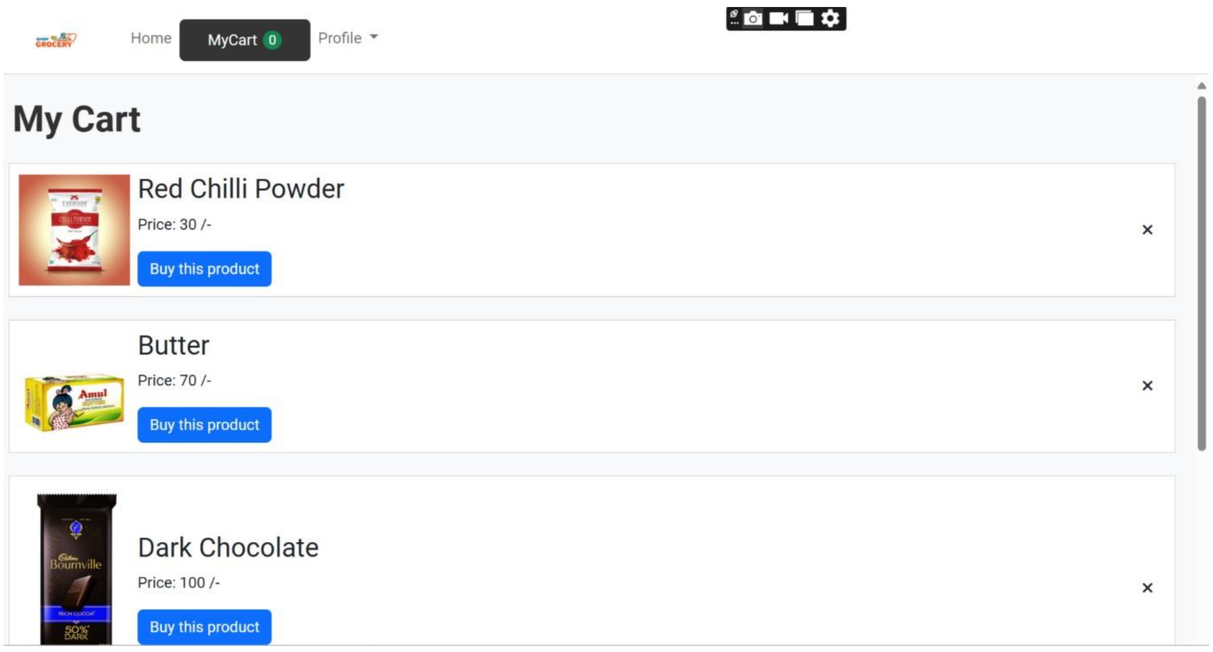
Products

Red Chilli Powder
Chilli Powder
price: 30/Kg
4 ★

Paneer
Milk Product
price: 98/Kg
4.7 ★

chips
snacks
price: 50/Kg
4 ★

## Cart pages



Home MyCart 0 Profile

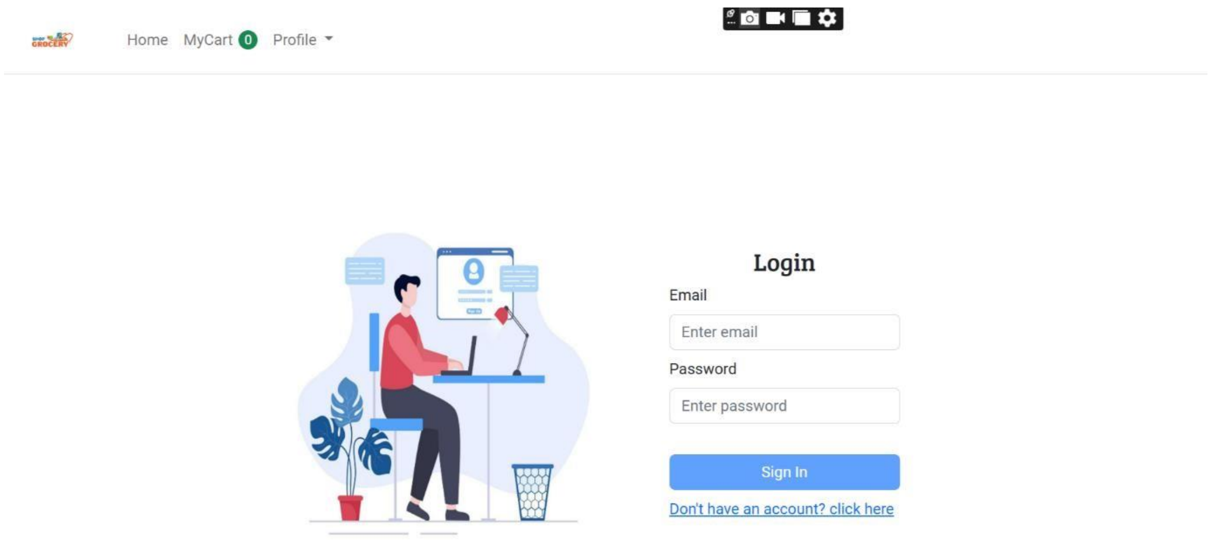
# My Cart

Red Chilli Powder
Price: 30 /-
Buy this product

Butter
Price: 70 /-
Buy this product

Dark Chocolate
Price: 100 /-
Buy this product

## Login page:



Home MyCart 0 Profile

Login

Email

Enter email

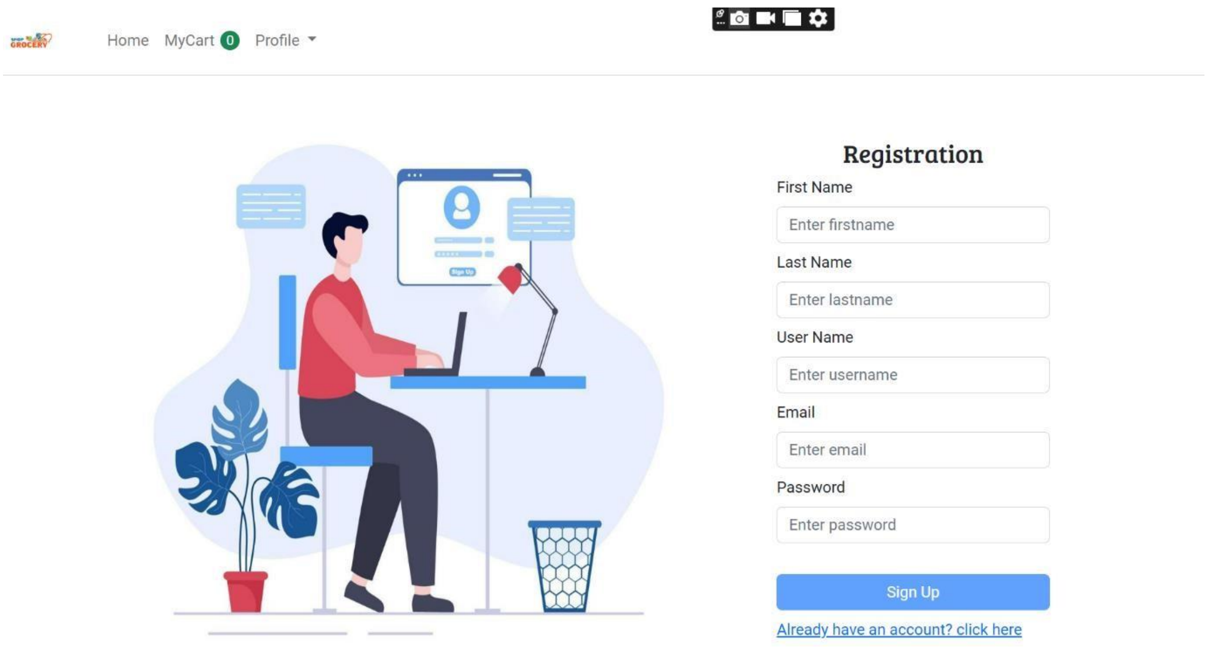
Password

Enter password

Sign In

Don't have an account? click here

## Register Page:



Registration

First Name

Enter firstname

Last Name

Enter lastname

User Name

Enter username

Email

Enter email

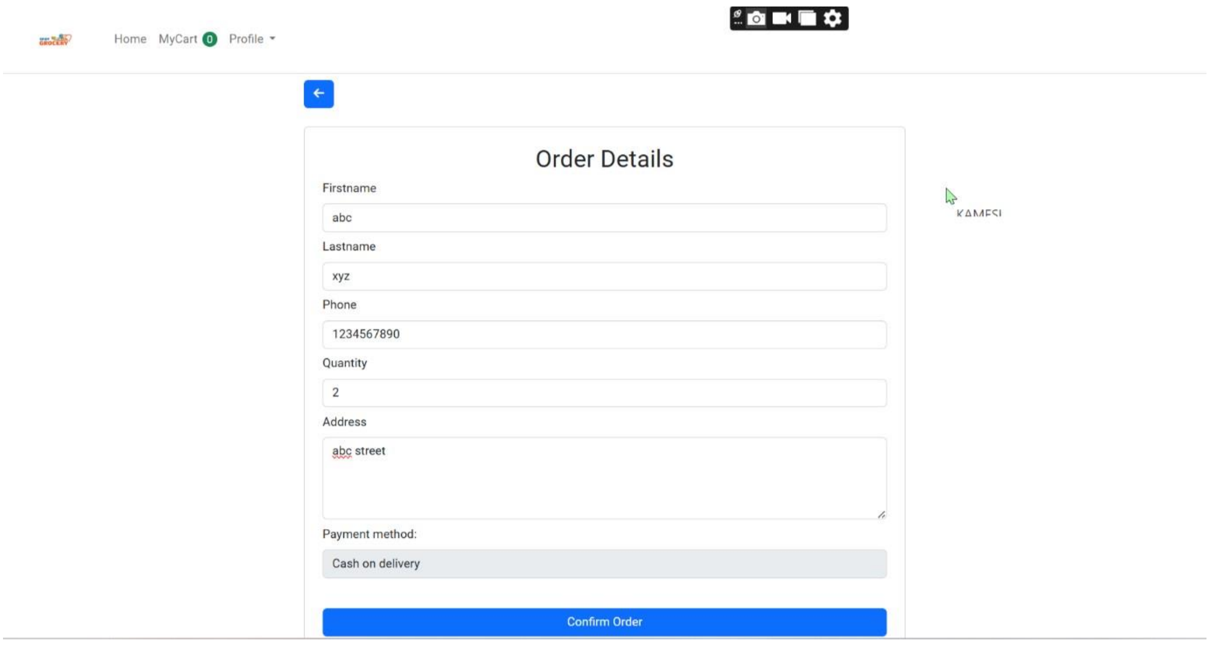
Password

Enter password

Sign Up

Already have an account? click here

## Order Page



Order Details

Firstname

abc

Lastname

xyz

Phone

1234567890

Quantity

2

Address

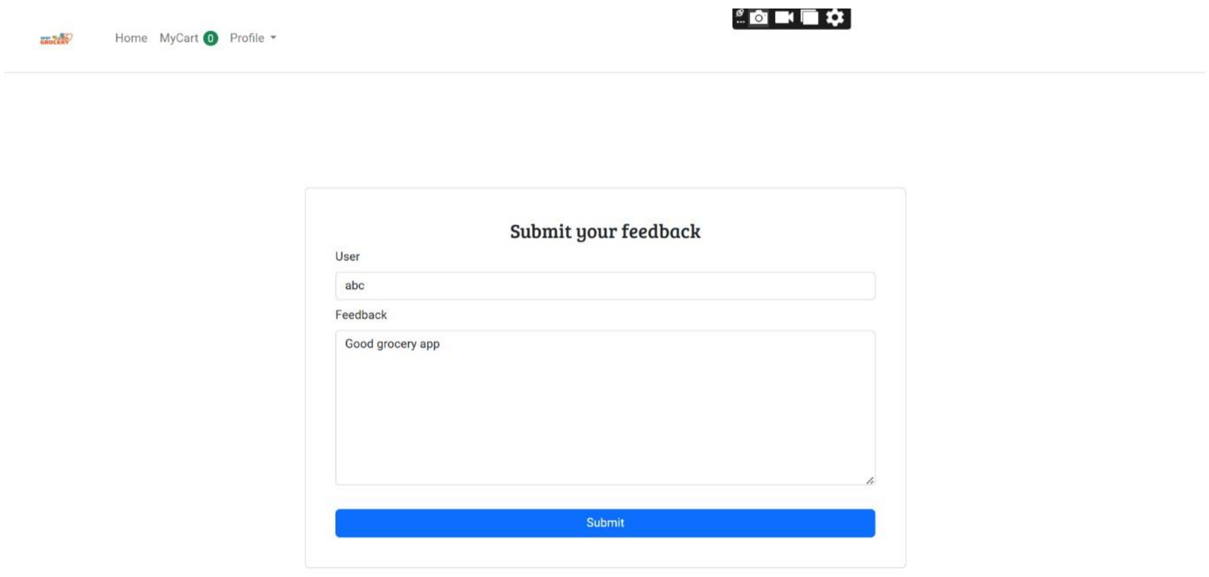
abc street

Payment method:

Cash on delivery

Confirm Order

## Feedback Page



Submit your feedback

User

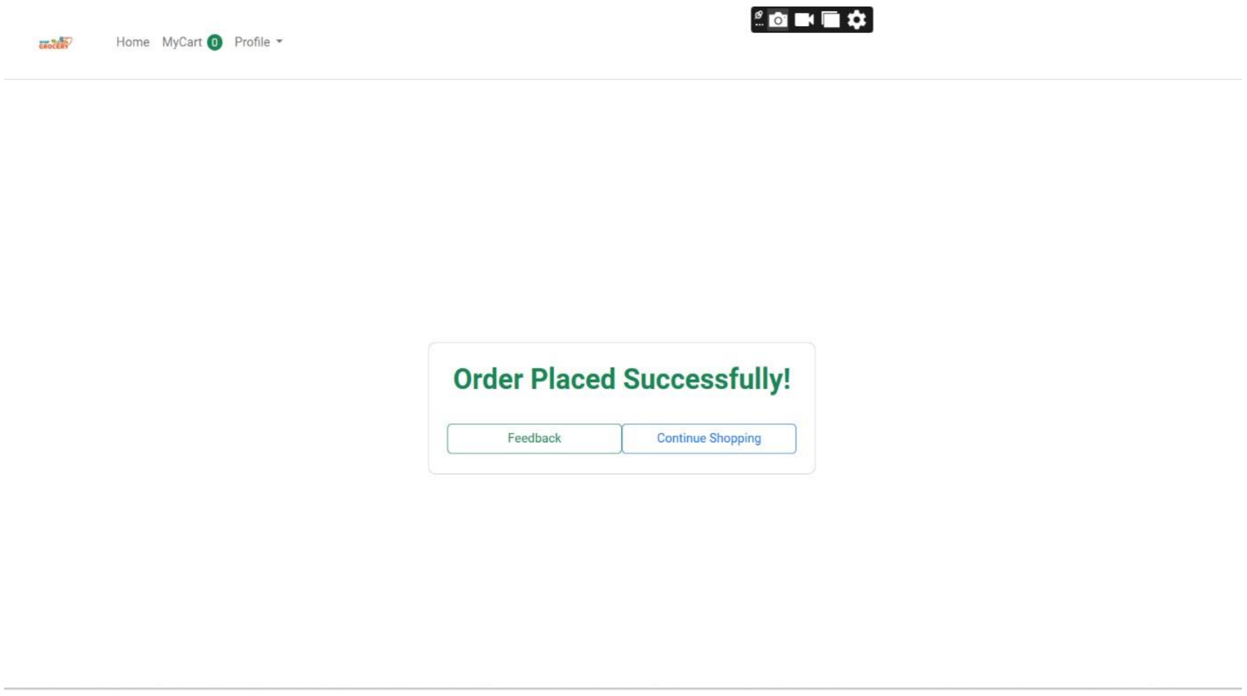
abc

Feedback

Good grocery app

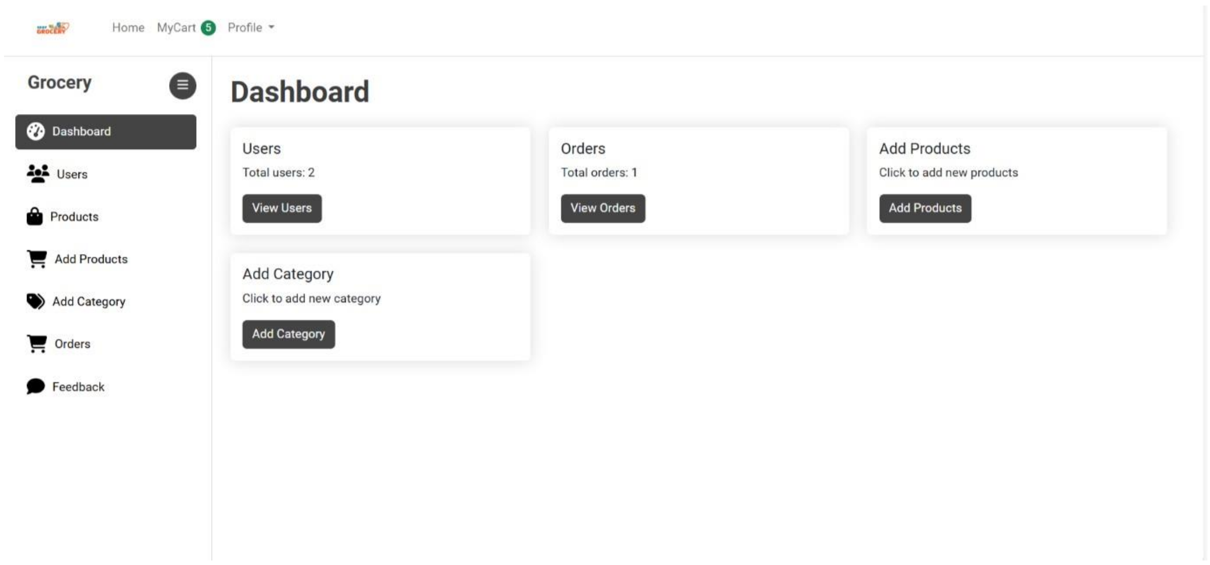
Submit

## Admin View



Order Placed Successfully!

## Dashboard Pages



Dashboard

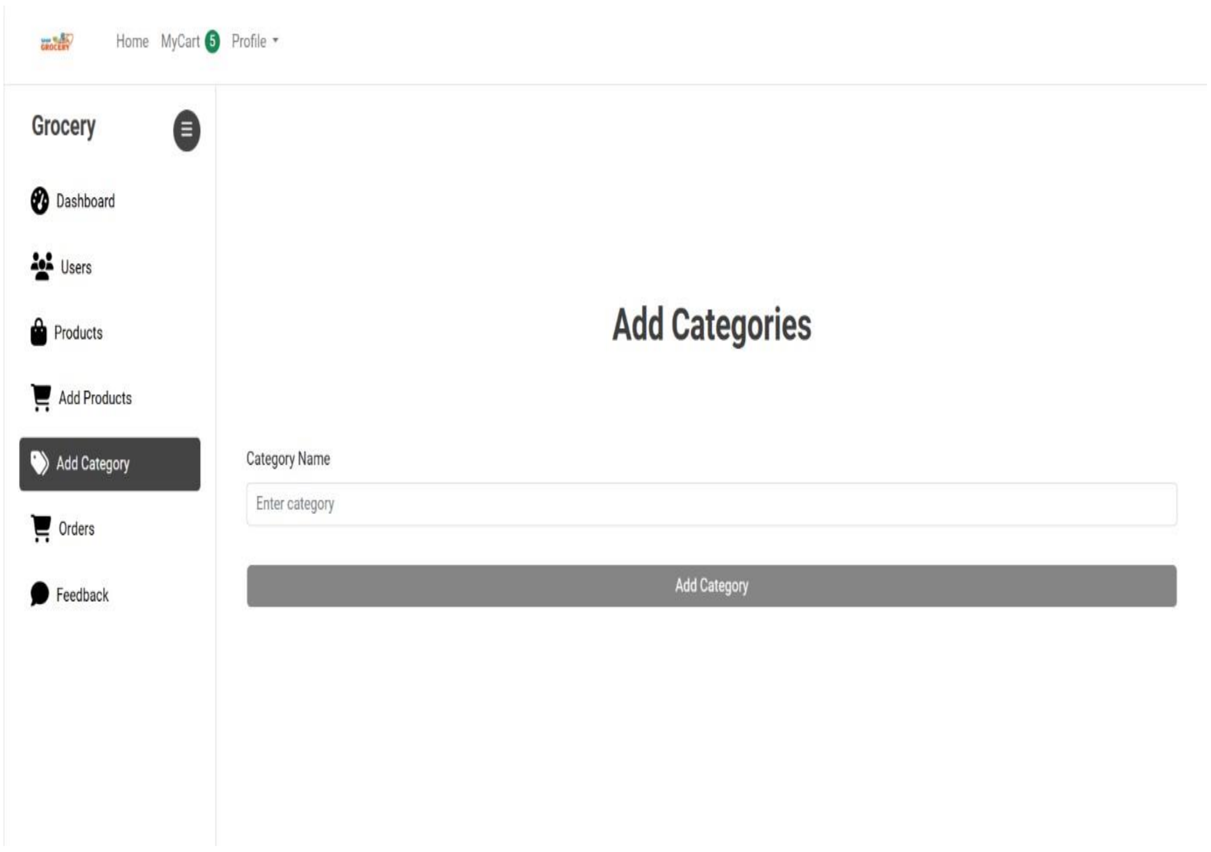
Users
Total users: 2
View Users

Orders
Total orders: 1
View Orders

Add Products
Click to add new products
Add Products

Add Category
Click to add new category
Add Category

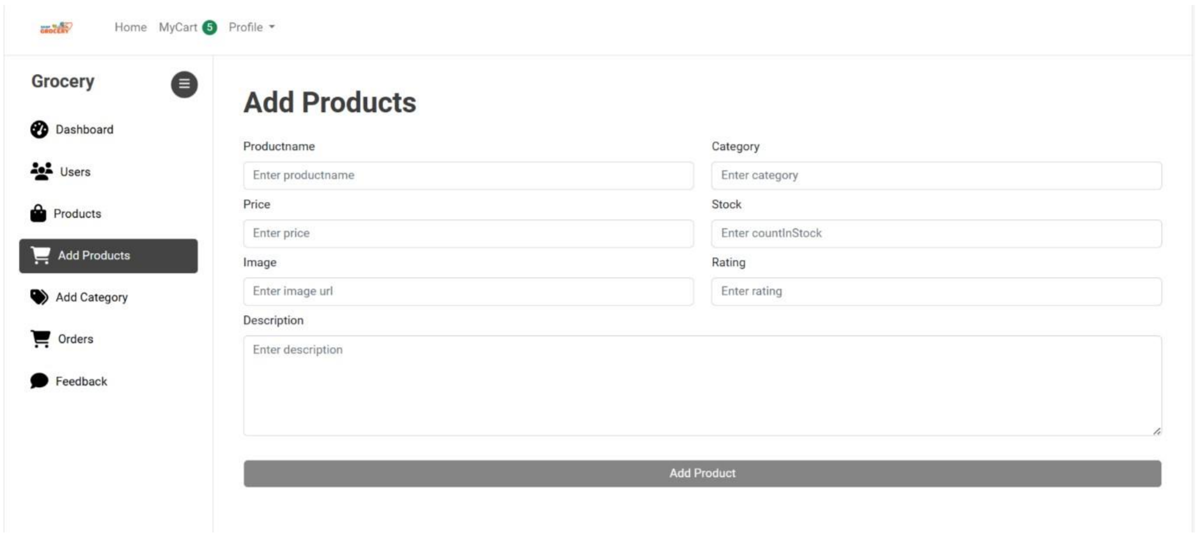
## Add Category



Add Categories

Category Name
Enter category
Add Category

## Add Products



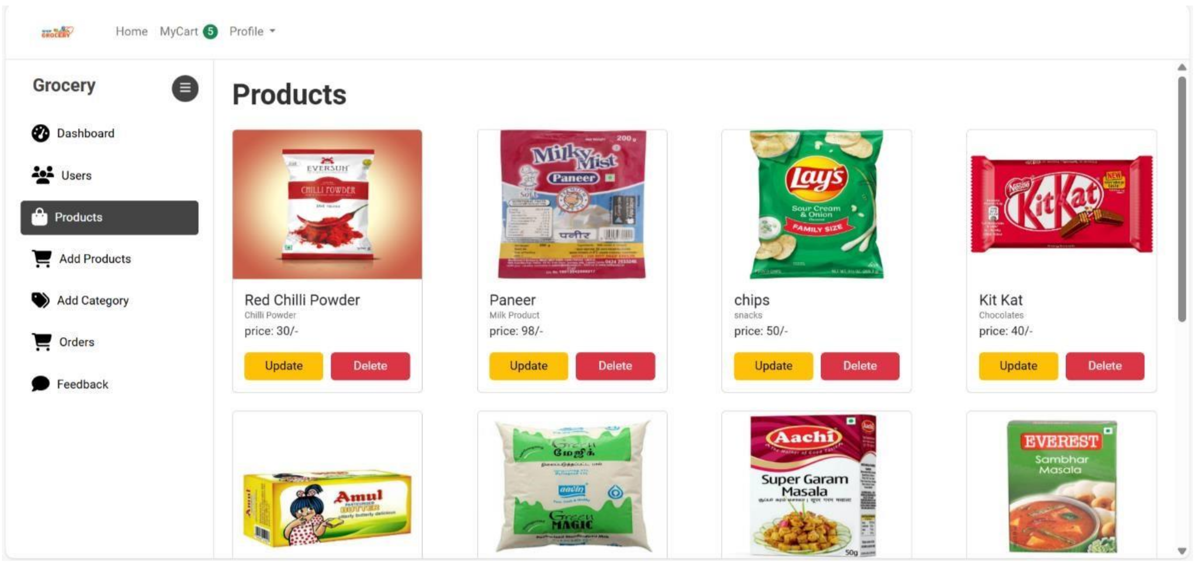
Add Products

| | |
|---|---|
| Productname | Category |
| Enter productname | Enter category |
| Price | Stock |
| Enter price | Enter countInStock |
| Image | Rating |
| Enter image url | Enter rating |

Description

Enter description

Add Product



Products

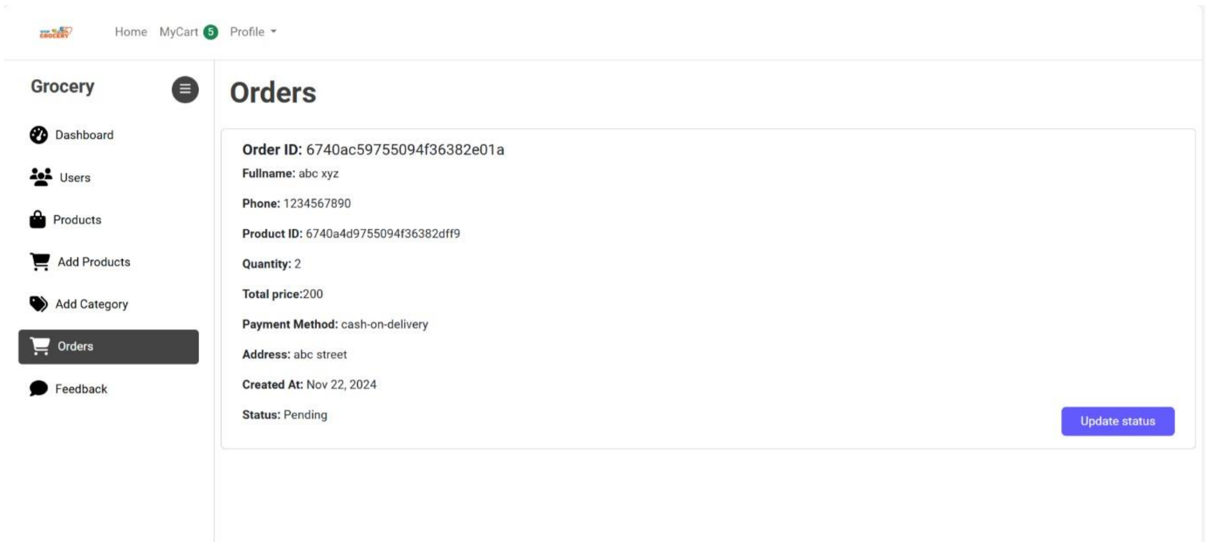
Red Chilli Powder
Chilli Powder
price: 30/-
Update Delete

Paneer
Milk Product
price: 98/-
Update Delete

chips
snacks
price: 50/-
Update Delete

Kit Kat
Chocolates
price: 40/-
Update Delete

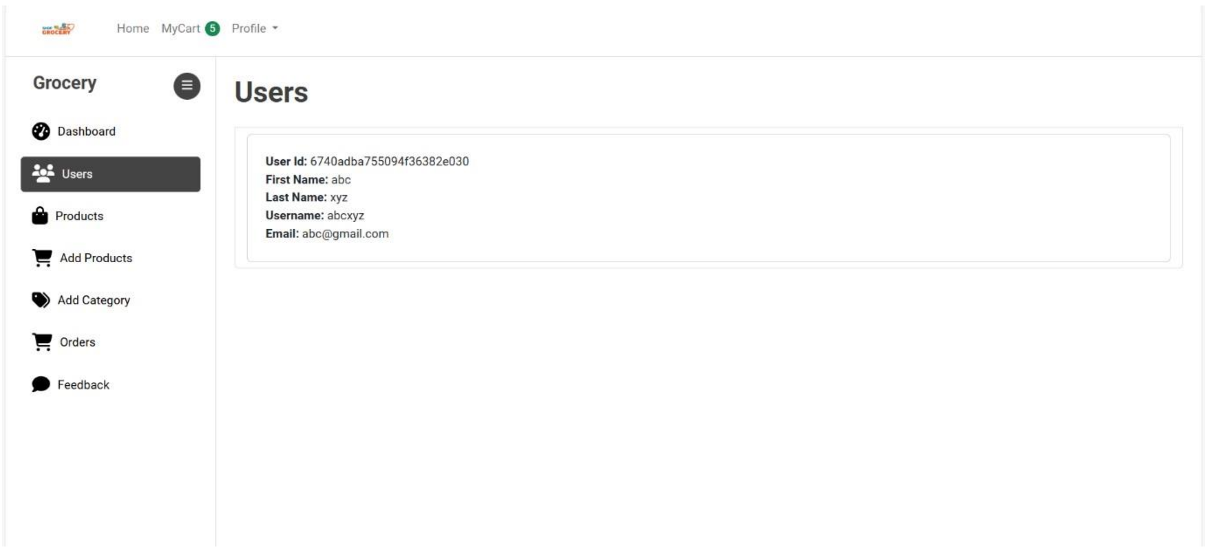
## Orders



Orders

**Order ID:** 6740ac59755094f36382e01a
**Fullname:** abc xyz
**Phone:** 1234567890
**Product ID:** 6740a4d9755094f36382dff9
**Quantity:** 2
**Total price:**200
**Payment Method:** cash-on-delivery
**Address:** abc street
**Created At:** Nov 22, 2024
**Status:** Pending

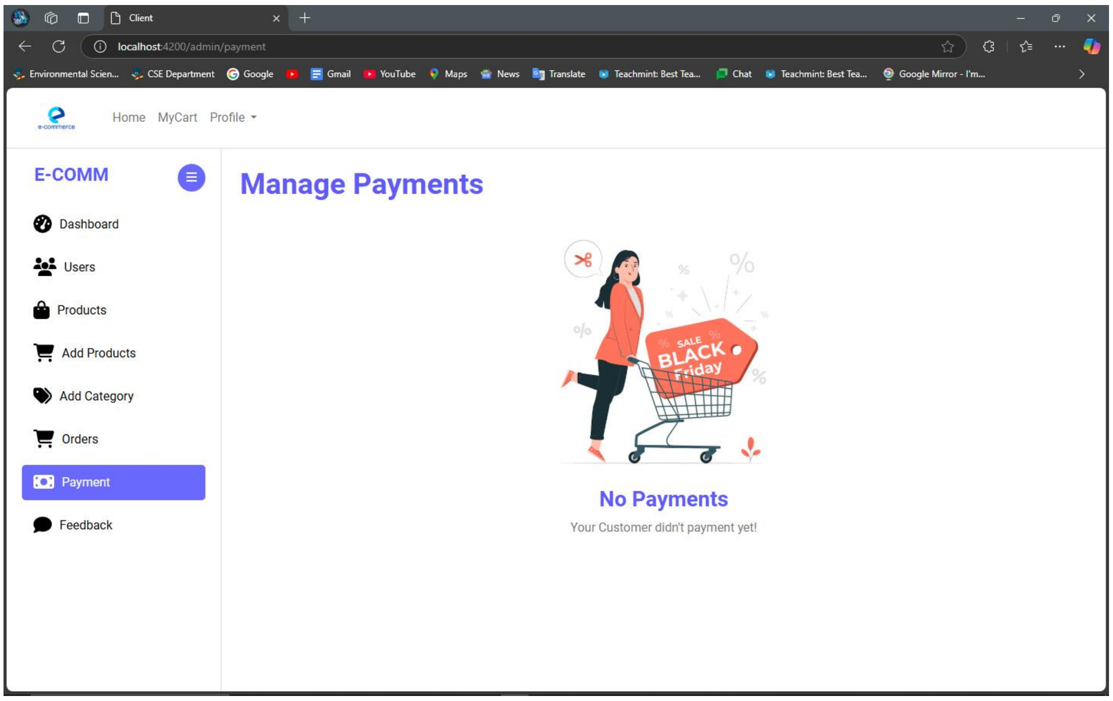
## User



Users

**User Id:** 6740adba755094f36382e030
**First Name:** abc
**Last Name:** xyz
**Username:** abcxyz
**Email:** abc@gmail.com

## Manage Payments


Client
localhost:4200/admin/payment
Environmental Scien... CSE Department Google Gmail YouTube Maps News Translate Teachmint: Best Tea... Chat Teachmint: Best Tea... Google Mirror - I'm...
Home MyCart Profile
E-COMM
Dashboard
Users
Products
Add Products
Add Category
Orders
Payment
Feedback

Manage Payments

No Payments

Your Customer didn't payment yet!

# 13.CONCLUSION

The **E-commerce WebApp** is a full-featured platform designed to streamline online economy for users and empower administrators to manage the system efficiently.

Node.js and Express.js for a robust backend, and MongoDB for scalable data storage.

Key features like user authentication, cart management, and seamless checkout ensure an excellent user experience, while the admin panel offers tools for effective product, order, and user management.

Through rigorous testing and thoughtful design, the application demonstrates reliability, usability, and scalability, making it suitable for real-world deployment. The project showcases technical expertise in full-stack development and emphasizes modern development practices, including modular architecture, RESTful API design, and responsive UI development.

Future enhancements, such as integrating AI-powered recommendations or advanced analytics, can further enrich the app’s functionality and provide added value to users.

# 14.REFERENCES

1. **React Documentation** - https://reactjs.org/docs
2. **Node.js Documentation** - https://nodejs.org/en/docs
3. **Express.js Guide** - https://expressjs.com/en/guide
4. **MongoDB Documentation** - https://www.mongodb.com/docs
5. **JWT Authentication** - https://jwt.io/introduction
6. **Cypress Testing** - https://www.cypress.io
7. **Bootstrap (for responsive design)** - https://getbootstrap.com/docs
8. **Postman (for API testing)** - https://www.postman.com
9. **MERN Stack Tutorials** - https://www.freecodecamp.org/news/mern-stack-tutorial