SMARTBRIDGE
Let's Bridge the Gap

ShopEZ: One-Stop Shop for Online Purchases - Project Documentation

This document provides a comprehensive overview of the ShopEZ project, outlining its purpose, architecture, setup instructions, and key technical details as a Full Stack MERN (MongoDB, Express.js, React, Node.js) application.

1. Introduction

- **Project Title:** ShopEZ: One-Stop Shop for Online Purchases
- **Team Members:**
    - [J.Eesha] – Frontend Lead
    - [S.Arshiya Taj] - Backend Developer
    - [K.Keerthi] - Database Administrator
    - [C.Anjali] - QA Engineer
    - [T.Yaswanth] - Testing

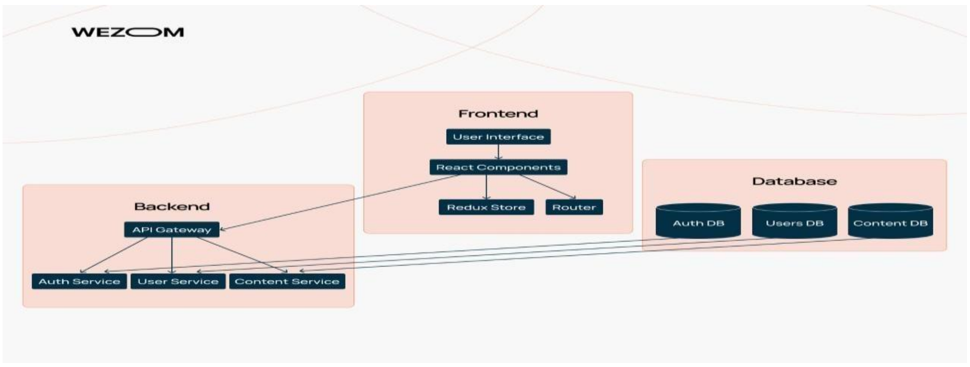
2. Project Overview

- **Purpose:** ShopEZ is designed to be a comprehensive online shopping platform that simplifies the purchasing process for customers and provides efficient management tools for sellers. Its primary goal is to offer a seamless, secure, and personalized ecommerce experience.
- **Features:**
    - **Effortless Product Discovery:** Intuitive categories, robust search, and advanced filtering options.
    - **Personalized Shopping Experience:** (Future) AI-powered product recommendations.
    - **Seamless Checkout Process:** Secure and efficient multi-step checkout.
    - **Order Confirmation & Tracking:** Instant notifications and real-time status updates for customers.
    - **Efficient Order Management for Sellers:** A dedicated dashboard for order processing, inventory updates, and product listing.
    - **Insightful Analytics for Business Growth:** (Future) Data visualizations for seller performance.
    - **User Management:** Registration, login, profile management.

3. Architecture

ShopEZ employs a modern MERN stack architecture, ensuring a scalable, high-performance, and responsive application.

- **Frontend (Client-side):**
    - **Framework:** React.js, a declarative, component-based JavaScript library for building user interfaces.
    - **Styling:** Tailwind CSS for a utility-first approach to quickly build responsive designs.
    - **State Management:** React Context API or Zustand for efficient state handling across components.
    - **UI Components:** Utilizes shadcn/ui for high-quality, accessible UI elements and lucide-react for icons.
    - **Interaction:** Communicates with the backend via RESTful API calls using fetch or Axios.
    - **Responsiveness:** Designed to be fully responsive, adapting to various screen sizes (mobile, tablet, desktop).
- **Backend (Server-side):**
    - **Language/Runtime:** Node.js, a JavaScript runtime built on Chrome's V8 JavaScript engine.
    - **Framework:** Express.js, a fast, unopinionated, minimalist web framework for Node.js, used to build RESTful APIs.
    - **Middleware:** Leverages various Express middleware for parsing requests, handling authentication, and error management.
    - **API Endpoints:** Exposes structured RESTful API endpoints for all core functionalities (users, products, carts, orders, etc.). o **Authentication:** Handles user authentication and authorization using JW



- **Database:**
  - **Type:** MongoDB, a NoSQL, document-oriented database. It provides flexibility and scalability, ideal for handling the diverse data requirements of an ecommerce platform.
  - **ORM/ODM:** Mongoose.js, an object data modeling (ODM) library for MongoDB and Node.js, used for schema definition, data validation, and simplified interaction with the database.
  - **Collections (Schema Overview):**
    - users: Stores user details (username, email, password hash, roles (buyer/seller/admin)).
    - // Example User Schema
    - {
    - "username": "String",
    - "email": "String (unique)",
    - "password": "String (hashed)",
    - "role": "String (enum: 'buyer', 'seller', 'admin')",
    - "createdAt": "Date",
    - "updatedAt": "Date"
    - }

    - products: Stores product information (name, description, price, category, images, stock quantity, seller ID).
    - // Example Product Schema
    - {
    - "name": "String",
    - "description": "String",
    - "price": "Number",
    - "category": "String",
    - "imageUrl": "String",
    - "stock": "Number",
    - "seller": "ObjectId (ref to User)",

- "createdAt": "Date",
- "updatedAt": "Date"
- }

- carts: Stores items currently in a user's shopping cart (user ID, array of product IDs and quantities).
- // Example Cart Schema
- {
- "user": "ObjectId (ref to User)",
- "items": [
- {
- "product": "ObjectId (ref to Product)",
- "quantity": "Number"
- }
-],
- "createdAt": "Date",
- "updatedAt": "Date"
- }

- orders: Stores details of completed orders (user ID, ordered items, total amount, shipping address, status, payment details).
- // Example Order Schema
- {
- "user": "ObjectId (ref to User)",
- "items": [
- {
- "product": "ObjectId (ref to Product)",
- "quantity": "Number",
- "price": "Number" // Price at the time of order
- }

-],
- "totalAmount": "Number",
- "shippingAddress": "Object",
- "status": "String (enum: 'pending', 'processing', 'shipped', 'delivered', 'cancelled')",
- "paymentDetails": "Object",
- "createdAt": "Date",
- "updatedAt": "Date"
- }

4. Setup Instructions

To get the ShopEZ application up and running on your local machine, follow these steps:

- **Prerequisites:**
  - Node.js (LTS version recommended, e.g., 18.x or 20.x) ○ npm (Node Package Manager, comes with Node.js)
  - MongoDB Community Server (Installed and running locally, or access to a cloud MongoDB Atlas instance)
  - Git
- **Installation:**

1. **Clone the repository:**
2. git clone https://github.com/your-username/shopez-mern.git
3. cd shopez-mern

4. **Install backend dependencies:**
5. cd server
6. npm install

7. **Install frontend dependencies:**
8. cd ../client
9. npm install

10. **Set up environment variables:**

- **Backend (server/.env):** Create a .env file in the server directory.
- PORT=5000
- MONGO_URI=mongodb://localhost:27017/shopezdb # Or your MongoDB Atlas connection string
- JWT_SECRET=your_jwt_secret_key_here # Use a strong, random key
- # Add any other sensitive keys like payment gateway API keys

- **Frontend (client/.env):** Create a .env file in the client directory.
- REACT_APP_API_URL=http://localhost:5000/api # Match your backend URL

5. Folder Structure

The project is divided into two main parts: client for the React frontend and server for the Node.js/Express backend.

- **Client (client/):**
- client/
- ├── public/   # Public assets (index.html)
- ├── src/
- │   ├── assets/   # Images, icons, etc.
- │   ├── components/   # Reusable React components (e.g., Button, Modal)
- │   ├── pages/   # Top-level components for different views (e.g., HomePage, ProductPage, CheckoutPage)
- │   ├── context/   # React Context for global state management (e.g., AuthContext, CartContext)
- │   ├── hooks/   # Custom React Hooks
- │   ├── services/   # API interaction functions (e.g., authService.js, productService.js)
- │   ├── utils/   # Utility functions (e.g., helpers, formatters)
- │   ├── App.js   # Main application component
- │   ├── index.js   # React entry point
- │   ├── tailwind.config.js   # Tailwind CSS configuration

- │ └── ...
- ├── package.json
- └── .env

- **Server (server/):**
- server/
- ├── config/ # Database connection setup
- ├── controllers/ # Logic for handling API requests (e.g., userController.js, productController.js)
- ├── middlewares/ # Custom Express middleware (e.g., authMiddleware.js)
- ├── models/ # Mongoose schemas/models (e.g., User.js, Product.js, Order.js)
- ├── routes/ # API routes definitions (e.g., userRoutes.js, productRoutes.js)
- ├── utils/ # Server-side utility functions (e.g., jwtUtils.js)
- ├── server.js # Main Express application entry point
- ├── package.json
- └── .env

6. Running the Application

To start the frontend and backend servers locally:

- **Frontend:** Navigate to the client directory and run:
- cd client
- npm start

This will typically open the application in your browser at http://localhost:3000.

- **Backend:** Navigate to the server directory and run:
- cd server
- npm start

The backend server will typically run on http://localhost:5000 (or the port specified in your .env file).

7. API Documentation

All API endpoints are designed to be RESTful, using standard HTTP methods (GET, POST, PUT, DELETE).

- **Base URL:** http://localhost:8080/api
- **Users & Authentication:**
  - POST /api/auth/register
    - **Description:** Register a new user.
    - **Request Body:** { username, email, password, role }
    - **Response:** { token, user: { id, username, email, role } }
  - POST /api/auth/login
    - **Description:** Authenticate a user.
    - **Request Body:** { email, password }
    - **Response:** { token, user: { id, username, email, role } }
  - GET /api/users/profile (Protected)
    - **Description:** Get authenticated user's profile.
    - **Headers:** Authorization: Bearer <token>
    - **Response:** { user: { id, username, email, role, ... } }
- **Products:**
  - GET /api/products
    - **Description:** Get all products. Supports query parameters for search and filtering.
    - **Query Params:** ?search=<keyword>, ?category=<categoryName>, ?minPrice=<val>&maxPrice=<val>
    - **Response:** [{ productId, name, price, ... }]
  - GET /api/products/:id
    - **Description:** Get product by ID.
    - **Response:** { productId, name, description, price, ... }
  - POST /api/products (Protected - Admin/Seller)
    - **Description:** Add a new product.
    - **Headers:** Authorization: Bearer <token>
    - **Request Body:** { name, description, price, category, imageUrl, stock }

- **Response:** { productId, name, ... } ○ PUT /api/products/:id (Protected - Admin/Seller) **Description:** Update product details.
- **Headers:** Authorization: Bearer <token>
- **Request Body:** { name?, description?, price?, ... }
- **Response:** { message: 'Product updated successfully', product: { ... } }

○ DELETE /api/products/:id (Protected - Admin/Seller) **Description:** Delete a product.

- **Headers:** Authorization: Bearer <token>
- **Response:** { message: 'Product deleted successfully' }

- **Cart:**
  - GET /api/cart
    - **Description:** Get authenticated user's cart.
    - **Headers:** Authorization: Bearer <token>
    - **Response:** { cart: [{ product: { id, name, ... }, quantity }] } ○ POST /api/cart/add (Protected)
    - **Description:** Add item to cart.
    - **Headers:** Authorization: Bearer <token>
    - **Request Body:** { productId, quantity }
    - **Response:** { message: 'Item added to cart', cart: { ... } } ○ PUT /api/cart/update (Protected)
    - **Description:** Update item quantity in cart.
    - **Headers:** Authorization: Bearer <token>
    - **Request Body:** { productId, quantity }
    - **Response:** { message: 'Cart updated', cart: { ... } } ○ DELETE /api/cart/remove/:productId (Protected) **Description:** Remove item from cart.
    - **Headers:** Authorization: Bearer <token>
    - **Response:** { message: 'Item removed from cart', cart: { ... } }
- **Orders:**
  - POST /api/orders (Protected)

 Description: Place a new order from the cart.

 Headers: Authorization: Bearer <token>

 Request Body: { shippingAddress: { street, city, ... }, paymentMethod: 'credit_card' }

 Response: { message: 'Order placed successfully', order: { orderId, ...

} } ○ GET /api/orders/my-orders (Protected)

 Description: Get orders for the authenticated user.

 Headers: Authorization: Bearer <token>

 Response: [{ orderId, status, totalAmount, ... }] ○ GET /api/orders/:id (Protected - User's own order or Admin/Seller) **Description:** Get order details by ID.

 Headers: Authorization: Bearer <token>

 Response: { orderId, items: [...], status, ... } ○ PUT /api/orders/:id/status (Protected - Admin/Seller) **Description:** Update order status.

 Headers: Authorization: Bearer <token>

 Request Body: { status: 'shipped' }

 Response: { message: 'Order status updated', order: { ... } }

8. Authentication

Authentication and authorization in ShopEZ are handled using JSON Web Tokens (JWTs).

- **Process:**
  1. **Login/Register:** When a user logs in or registers, the backend authenticates their credentials.
  2. **Token Generation:** Upon successful authentication, the server generates a JWT containing the user's ID and role, then signs it with a secret key.
  3. **Token Issuance:** The JWT is sent back to the client.
  4. **Client Storage:** The client stores the JWT (e.g., in localStorage or HttpOnly cookies).
  5. **Protected Routes:** For subsequent requests to protected routes, the client includes the JWT in the Authorization header as a Bearer token (Authorization: Bearer <token>).
  6. **Server Verification:** Backend middleware intercepts these requests, verifies the JWT's signature and expiration, and extracts the user ID and role.

7. **Authorization:** Based on the user's role (buyer, seller, admin), the backend authorizes access to specific resources or actions.

- **Security:**
  - Password hashing (e.g., using bcrypt.js) for storing user passwords securely. o JWTs provide stateless authentication, allowing for scalability.
  - Role-based access control (RBAC) ensures users only access resources they are authorized for.

9. User Interface

The User Interface (UI) is designed to be intuitive, responsive, and visually appealing, consistent with the "Effortless Product Discovery" and "Seamless Checkout" goals.

(This section would typically include visual aids. As this is a text-based documentation, placeholders are used.)

- **Homepage/Product Listing:** (Screenshot/GIF of the main product grid with search and filter options, similar to the React code provided earlier)
- **Product Details Page:** (Screenshot/GIF of a page showing full product description, images, price, "Add to Cart" button, and customer reviews (if implemented))
- **Shopping Cart View:** (Screenshot/GIF of the cart contents, quantities, subtotal, and options to proceed to checkout)
- **Checkout Flow:** (Sequence of screenshots/GIFs showing address input, payment selection, and order review steps)
- **User Profile & Order History:** (Screenshot/GIF of a user's dashboard showing their past orders and personal information)
- **Seller Dashboard (Key Views):** (Screenshot/GIFs of order list, product management, and simple analytics charts)

10. Testing

The testing strategy for ShopEZ focuses on ensuring the reliability, functionality, and performance of both the frontend and backend.

- **Unit Testing:**
  - **Tools:** Jest (for React components and Node.js functions), React Testing Library (for React components).
  - **Scope:** Individual functions, components, and utility modules to ensure they work as expected in isolation.
- **Integration Testing:**
  - **Tools:** Supertest (for Node.js API endpoints), Jest.

- **Scope:** Verifying the interaction between different modules (e.g., a controller interacting with a Mongoose model), and API endpoint functionality.

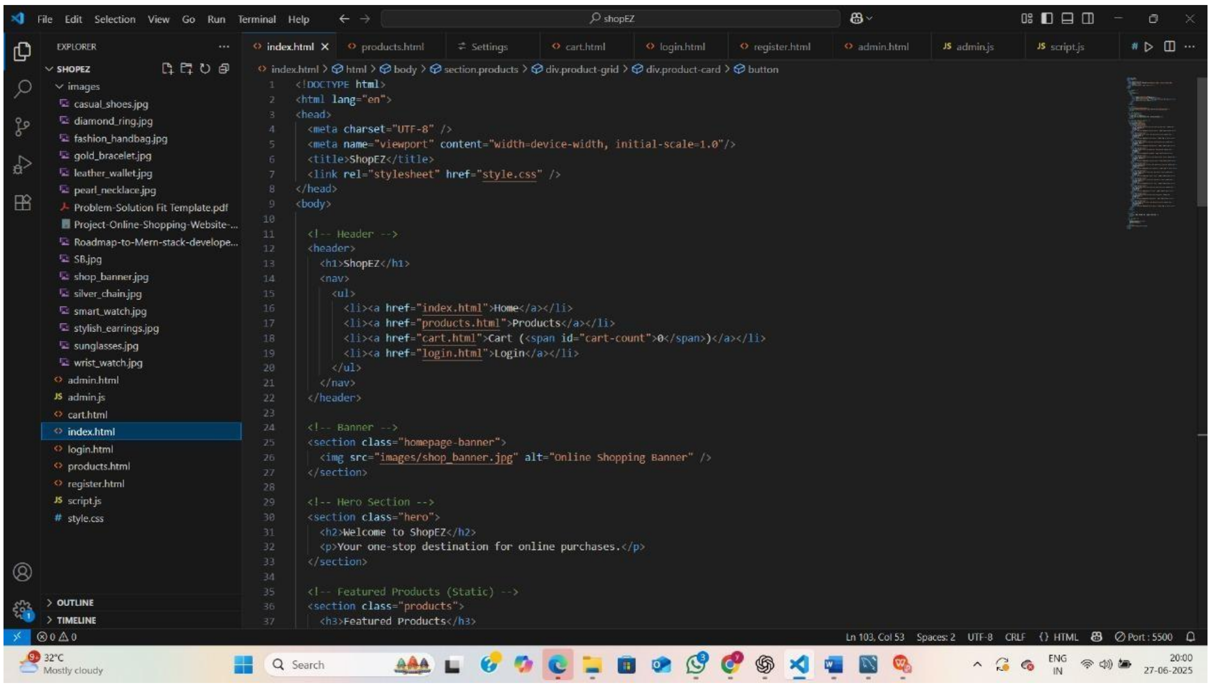
- **End-to-End (E2E) Testing:**
  - **Tools:** Cypress or Playwright.
  - **Scope:** Simulating full user journeys through the application (e.g., registering, logging in, adding to cart, checking out) to ensure all components work together seamlessly.
- **Performance Testing:** (As described in section 6.1 of the previous report) ○ **Tools:**

  Apache JMeter or K6.
  - **Scope:** Load testing, stress testing, and response time benchmarking for critical API endpoints.

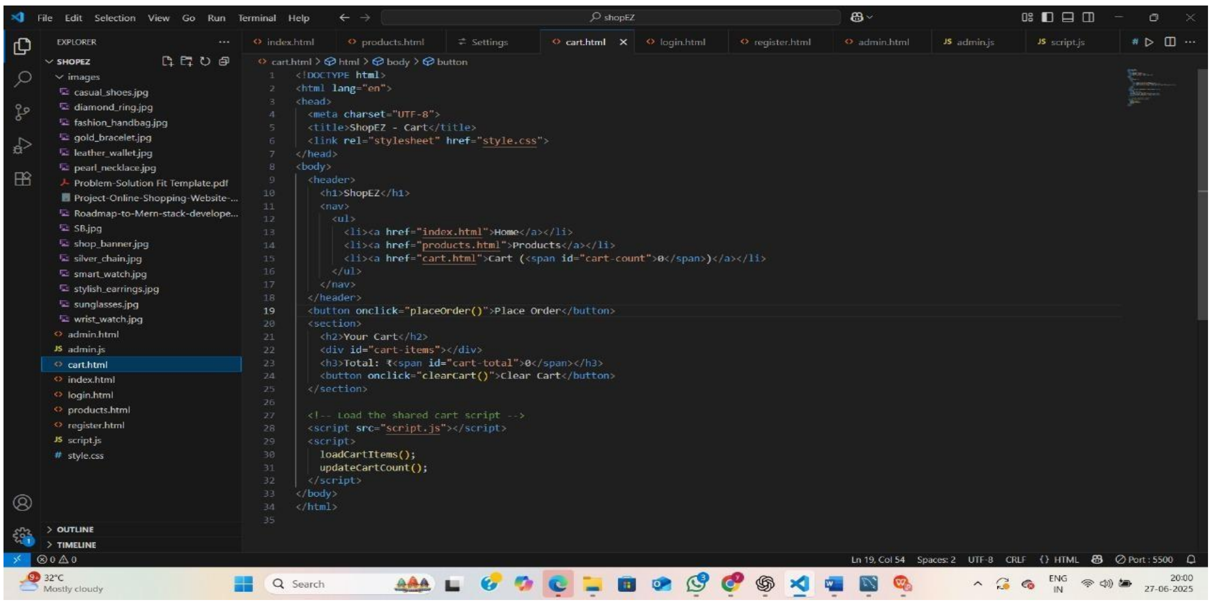
11. Screenshots or Demo

• **Live demo Video: :https://drive.google.com/file/d/12RVF9gA0GGzxsvDuTFEmYtx09sPQh44q/view?usp=sharing**

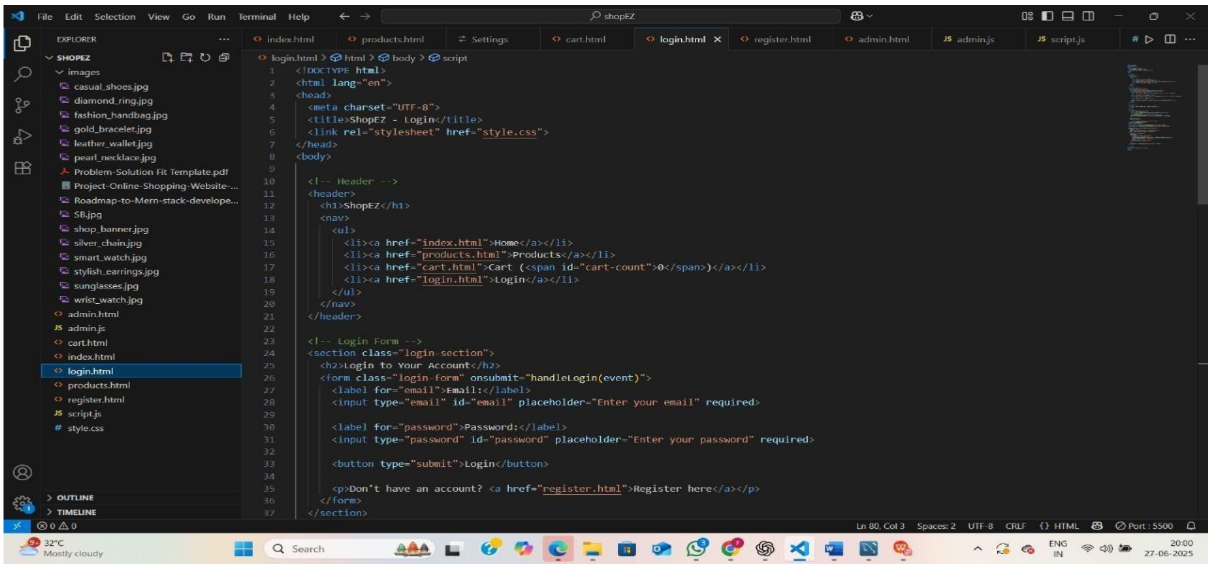
- **Static Screenshots:**



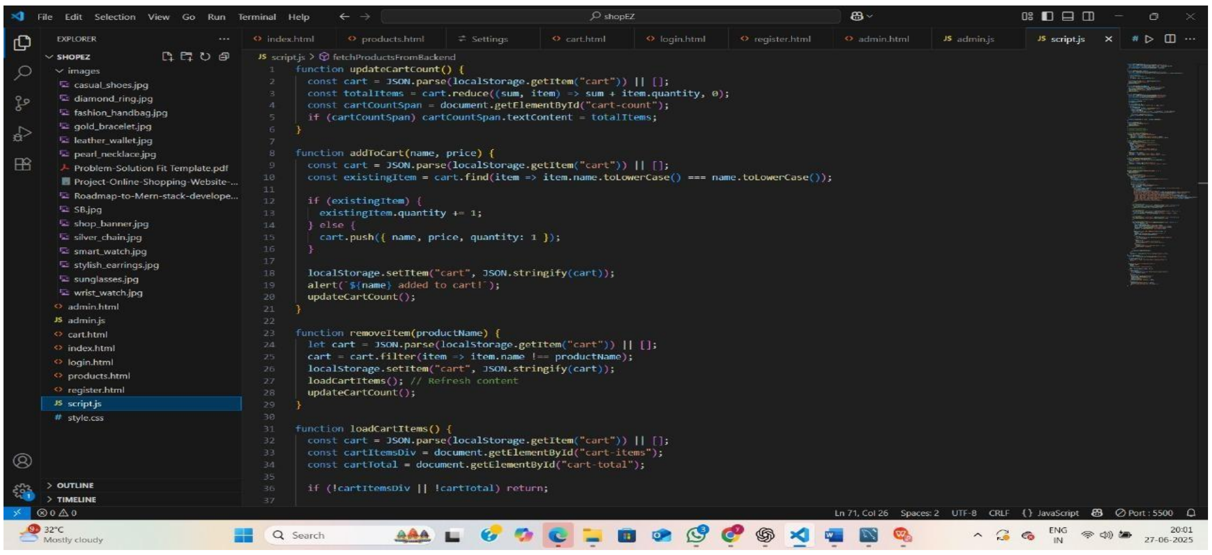
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>ShopEZ</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>

  <!-- Header -->
  <header>
    <h1>ShopEZ</h1>
    <nav>
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="products.html">Products</a></li>
        <li><a href="cart.html">Cart (<span id="cart-count">0</span>)</a></li>
        <li><a href="login.html">Login</a></li>
      </ul>
    </nav>
  </header>

  <!-- Banner -->
  <section class="homepage-banner">
    <img src="images/shop_banner.jpg" alt="Online Shopping Banner" />
  </section>

  <!-- Hero Section -->
  <section class="hero">
    <h2>Welcome to ShopEZ</h2>
    <p>Your one-stop destination for online purchases.</p>
  </section>

  <!-- Featured Products (Static) -->
  <section class="products">
    <h3>Featured Products</h3>
```



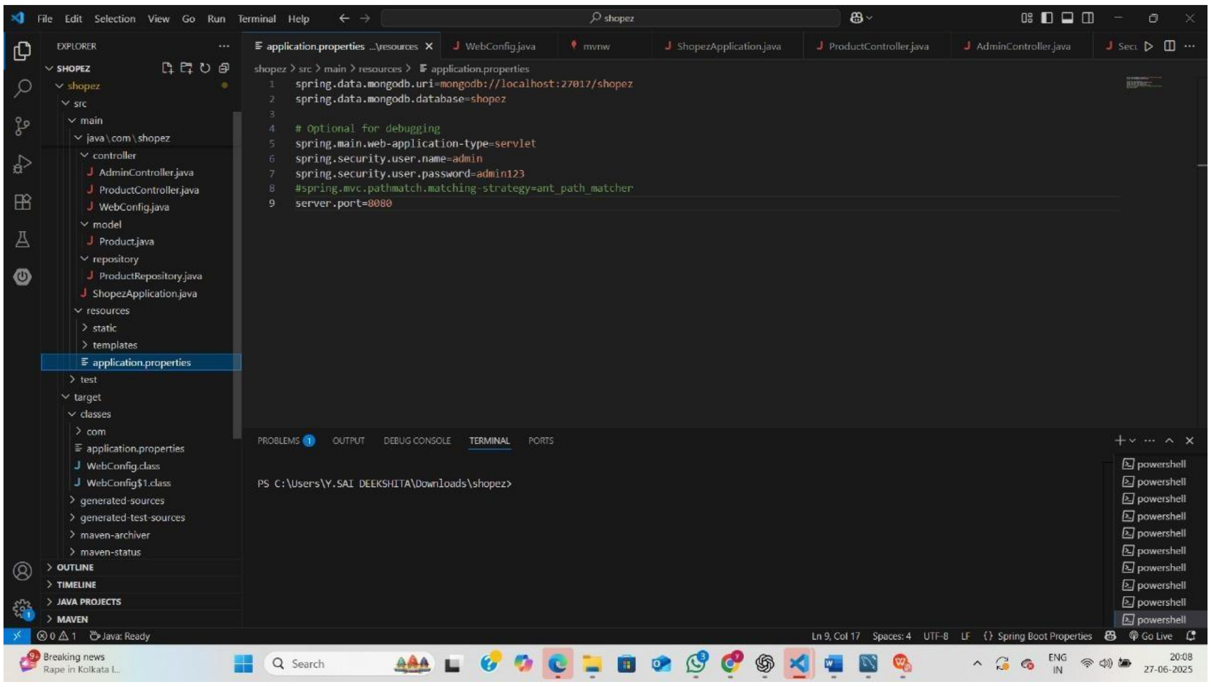
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ShopEZ - Cart</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <header>
    <h1>ShopEZ</h1>
    <nav>
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="products.html">Products</a></li>
        <li><a href="cart.html">Cart (<span id="cart-count">0</span>)</a></li>
      </ul>
    </nav>
  </header>
  <button onclick="placeOrder()">Place Order</button>
  <section>
    <h2>Your Cart</h2>
    <div id="cart-items"></div>
    <h3>Total: ₹<span id="cart-total">0</span></h3>
    <button onclick="clearCart()">Clear Cart</button>
  </section>

  <!-- Load the shared cart script -->
  <script src="script.js"></script>
  <script>
    loadCartItems();
    updateCartCount();
  </script>
</body>
</html>
```



```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ShopEZ - Login</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

  <!-- Header -->
  <header>
    <h1>ShopEZ</h1>
    <nav>
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="products.html">Products</a></li>
        <li><a href="cart.html">Cart (<span id="cart-count">0</span>)</a></li>
        <li><a href="login.html">Login</a></li>
      </ul>
    </nav>
  </header>

  <!-- Login Form -->
  <section class="login-section">
    <h2>Login to Your Account</h2>
    <form class="login-form" onsubmit="handleLogin(event)">
      <label for="email">Email:</label>
      <input type="email" id="email" placeholder="Enter your email" required>

      <label for="password">Password:</label>
      <input type="password" id="password" placeholder="Enter your password" required>

      <button type="submit">Login</button>

      <p>Don't have an account? <a href="register.html">Register here</a></p>
    </form>
  </section>
```



```javascript
function updateCartCount() {
  const cart = JSON.parse(localStorage.getItem("cart")) || [];
  const totalItems = cart.reduce((sum, item) => sum + item.quantity, 0);
  const cartCountSpan = document.getElementById("cart-count");
  if (cartCountSpan) cartCountSpan.textContent = totalItems;
}

function addToCart(name, price) {
  const cart = JSON.parse(localStorage.getItem("cart")) || [];
  const existingItem = cart.find(item => item.name.toLowerCase() === name.toLowerCase());

  if (existingItem) {
    existingItem.quantity += 1;
  } else {
    cart.push({ name, price, quantity: 1 });
  }

  localStorage.setItem("cart", JSON.stringify(cart));
  alert(`${name} added to cart!`);
  updateCartCount();
}

function removeItem(productName) {
  let cart = JSON.parse(localStorage.getItem("cart")) || [];
  cart = cart.filter(item => item.name !== productName);
  localStorage.setItem("cart", JSON.stringify(cart));
  loadCartItems(); // Refresh content
  updateCartCount();
}

function loadCartItems() {
  const cart = JSON.parse(localStorage.getItem("cart")) || [];
  const cartItemsDiv = document.getElementById("cart-items");
  const cartTotal = document.getElementById("cart-total");

  if (!cartItemsDiv || !cartTotal) return;

```



```
spring.data.mongodb.uri=mongodb://localhost:27017/shopez
spring.data.mongodb.database=shopez

# Optional for debugging
spring.main.web-application-type=servlet
spring.security.user.name=admin
spring.security.user.password=admin123
#spring.mvc.pathmatch.matching-strategy=ant_path_matcher
server.port=8080
```

```
PS C:\Users\Y.SAI DEEKSHITA\Downloads\shopez>
```

12. Known Issues

- **Payment Gateway Integration (Sandbox Only):** The current payment integration is set up with a sandbox/test environment. Live transactions require full production API keys and adherence to PCI compliance.
- **Basic Search Functionality:** The current search is a simple keyword match. Full-text search with fuzzy matching and relevancy ranking is a future enhancement.
- **No Real-time Chat Support:** Customer service is currently via email/ticket system; live chat is not yet implemented.
- **Limited Seller Analytics:** The seller dashboard currently offers basic sales reports. Advanced analytics and customizable dashboards are planned.
- **Mobile Responsiveness Glitches:** While designed to be responsive, minor layout or interaction issues might occur on less common device sizes or orientations.

13. Future Enhancements

- **Advanced Recommendation Engine:** Implement AI/ML-driven algorithms for hyper personalized product suggestions based on browsing history, purchase patterns, and collaborative filtering.
- **Customer Reviews & Ratings:** Develop a robust system for users to submit and view product reviews and ratings, including photo/video uploads.
- **Wishlist Functionality:** Allow users to create and manage wishlists for products they are interested in.
- **Multi-Vendor Support:** Expand the platform to enable multiple independent sellers to register, list their products, and manage their storefronts within ShopEZ.

- **Loyalty Programs & Rewards:** Introduce a points-based loyalty program or discount tiers to incentivize repeat purchases.
- **Live Chat Support:** Integrate a real-time customer support chat feature using a service like Socket.io or a third-party chat widget.
- **Internationalization & Localization:** Support multiple languages and currencies to expand the platform's global reach.
- **Native Mobile Applications:** Develop dedicated iOS and Android applications using React Native for an optimized mobile user experience.
- **Integration with Shipping Carriers:** Automate shipping label generation, real-time tracking updates, and shipping cost calculations by integrating with popular shipping APIs (e.g., FedEx, UPS, USPS).
- **User Roles and Permissions:** Granular control over user roles (e.g., different levels of admin, specific seller permissions).
- **Payment Gateway Expansion:** Integrate with more diverse payment methods (e.g., Apple Pay, Google Pay, specific local payment options).