



MERN STACK POWERED BY MONGODB

SHOPEZ:E-COMMERCE APPLICATION

A PROJECT REPORT

Submitted by

DEVANAND C

113321104012

MAFIN SHARO J

113321104053

NISHAN K M

113321104068

PARAMESWARAN J

113321104069

BACHELOR OF ENGINEERING

COMPUTER SCIENCE AND ENGINEERING

VELAMMAL INSTITUTE OF TECHNOLOGY CHENNAI 601204

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report **“SHOPEZ: E-COMMERCE APPLICATION”** is the Bonafide work of **“DEVANAND C-113321104012, MAFIN SHARO J-113321104053, NISHAN K M-113321104068, PARAMESWARAN J-113321104069”** who carried out the Project work under my supervision.

SIGNATURE

Dr.V.P.GLADIS PUSHPARATHI
PROFESSOR,
HEAD OF THE DEPARTMENT,
Computer Science & Engineering,
Velammal Institute of Technology,
Velammal Gardens, Panchetti,
Chennai-601 204.

SIGNATURE

MRS.J.JOICE RUBY
ASSISSTANT PROFESSOR,
NM COORDINATOR ,
Computer Science & Engineering,
Velammal Institute of Technology,
Velammal Gardens, Panchetti,
Chennai-601 204

ACKNOWLEDGEMENT

We are personally indebted to many who had helped us during the course of this project work. Our deepest gratitude to the **God Almighty**.

We are greatly and profoundly thankful to our beloved Chairman **Thiru.M.V.Muthuramalingam** for facilitating us with this opportunity. Our sincere thanks to our respected Director **Thiru.M.V.M Sasi Kumar** for his consent to take up this project work and make it a great success.

We are also thankful to our Advisors **Shri.K.Razak, Shri.M.Vaasu**, our Principal **Dr.N.Balaji** and our Vice Principal **Dr.S.Soundararajan** for their never ending encouragement that drives us towards innovation.

We are extremely thankful to our Head of the Department **Dr.V.P.Gladis Pushaparathi** and Naan Mudhalvan Coordinator **Mrs.J.Joice Ruby**, for their valuable teachings and suggestions.

The Acknowledgment would be incomplete if we would not mention word of thanks to our Parents, Teaching and Non-Teaching Staffs, Administrative Staffs and Friends for their motivation and support throughout the project.

Finally, we thank all those who directly or indirectly contributed to the successful completion of this project. Your contributions have been a vital part of our success.

TABLE OF CONTENTS

S.NO	TITLE	PAGE NO
1	INTRODUCTION	5
2	PROJECT OVERVIEW	6
3	ARCHITECTURE	8
4	SETUP INSTRUCTIONS	9
5	RUNNING THE APPLICATION	16
6	API DOCUMENTATION	18
7	AUTHENTICATION	25
8	TESTING	31
9	SCREENSHOTS	33
10	KNOWN ISSUES	35
11	FUTURE ENHANCEMENTS	39

1.Introduction:

The **Shopsphere E-Commerce Application** is a modern and scalable web-based platform designed to facilitate the buying and selling of products online. Built with the aim of providing an exceptional shopping experience for users, the application integrates both the frontend (client-facing interface) and backend (server-side logic) to offer a smooth, secure, and feature-rich e-commerce environment.

In today's digital age, e-commerce has become a key component of retail, enabling businesses to reach customers worldwide and allowing users to shop from the comfort of their homes. The **Shopsphere E-Commerce Application** leverages the power of the internet to connect buyers with sellers in a secure, efficient, and scalable environment.

Key Objectives

The primary goal of the **Shopsphere E-Commerce Application** is to create an intuitive platform where users can easily discover, purchase, and manage products while offering robust tools for administrators to efficiently manage the product catalog, orders, and customer accounts. The application's core objectives include:

- **Providing a seamless shopping experience** for customers by offering a user-friendly interface, easy navigation, secure payments, and reliable order tracking.
- **Empowering administrators** with a comprehensive backend that allows them to manage users, products, categories, orders, and payments in an efficient and streamlined manner.
- **Ensuring the security of all transactions**, user data, and personal information with industry-standard protocols and security measures.
- **Scaling the platform** to handle increasing traffic, product listings, and order volumes, while maintaining high performance and availability.
- **Enhancing the user experience** with features like personalized recommendations, advanced product search, and mobile compatibility to cater to users across various devices.

Scope of the Application

The **Shopsphere E-Commerce Application** is designed to serve as a fully functional e-commerce platform capable of supporting diverse product categories, secure user authentication, and a seamless shopping process. It allows both users and administrators to perform the following functions:

- **User Account Management:** Users can sign up, log in, and manage their personal profiles, including shipping details and payment methods.
- **Product Browsing:** Customers can browse through a wide range of products, view details, and search by categories, price range, or other attributes.
- **Shopping Cart and Checkout:** Customers can add products to their cart, review their selections, and proceed to checkout, where they can enter payment information and finalize the purchase.

- **Order Tracking:** After completing a purchase, customers can track their order status, from pending to shipped and delivered.
- **Payment Gateway Integration:** Secure payment methods are integrated into the application, enabling customers to pay via various methods like credit cards, debit cards, or wallets.
- **Admin Dashboard:** Administrators can manage all aspects of the platform, including adding or editing products, processing orders, and managing user accounts.

2. Project Overview:

The **ShopEZ E-Commerce Application** is a comprehensive, scalable, and user-friendly online shopping platform designed to enhance the digital retail experience for both customers and merchants. It integrates advanced technologies to ensure seamless browsing, purchasing, and order management, catering to a wide range of user preferences and business needs.

Purpose:

The primary goal of ShopEZ is to provide an efficient, secure, and accessible platform for e-commerce operations. The platform enables customers to explore a diverse range of products, customize their shopping experience, and enjoy a secure and hassle-free checkout process. Simultaneously, it empowers merchants with tools for inventory management, order tracking, and sales analytics, helping them optimize operations and improve customer satisfaction.

Objectives:

1. Customer-Centric Experience:

- Offer an intuitive interface for users to browse products, view details, and make purchases effortlessly.
- Provide features like personalized recommendations, product reviews, and real-time order tracking.

2. Merchant Empowerment:

- Equip businesses with tools to manage inventory, monitor sales trends, and handle orders efficiently.
- Enable merchants to expand their digital presence and reach a broader audience.

3. Scalability and Performance:

- Develop a system capable of handling high traffic and large datasets.
- Ensure the platform is responsive and performs well on both web and mobile devices.

4. Security and Reliability:

- Implement robust authentication and data encryption mechanisms to safeguard user and business information.
- Provide real-time notifications and fail-safe systems for uninterrupted operations.

Key Features:

1. User Registration and Authentication:

- Secure signup and login processes with token-based authentication (JWT).

2. Product Browsing:

- Organized product catalog with categories, descriptions, images, pricing, and stock availability.

3. Cart Management and Checkout:

- Dynamic cart system allowing users to add, remove, or modify items before purchase.
- Seamless integration with payment gateways for secure transactions.

4. Order Management:

- Real-time order tracking for customers from placement to delivery.
- Merchants can monitor and update order statuses through a dedicated admin panel.

5. Admin Dashboard:

- Tools for managing product inventory, viewing sales data, and analyzing customer trends.

6. Notifications System:

- Real-time email and push notifications for order updates and promotional offers.

7. Personalized Recommendations:

- AI-powered suggestion system to recommend products based on user behavior and purchase history.

8. Feedback and Reviews:

- Customers can leave reviews and rate products, providing valuable insights to other users and merchants.

Benefits:

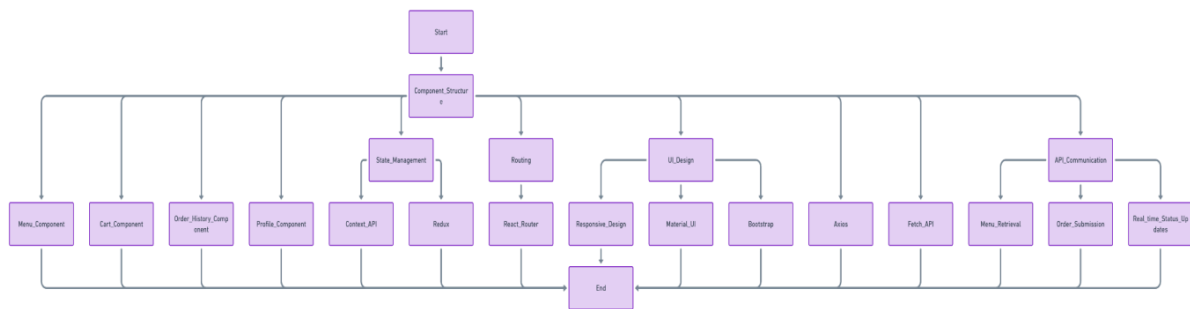
1. For Customers:

- Easy-to-navigate interface for discovering and purchasing products.
- Secure payment options and real-time updates on order status.
- Personalized recommendations and seamless mobile experience.

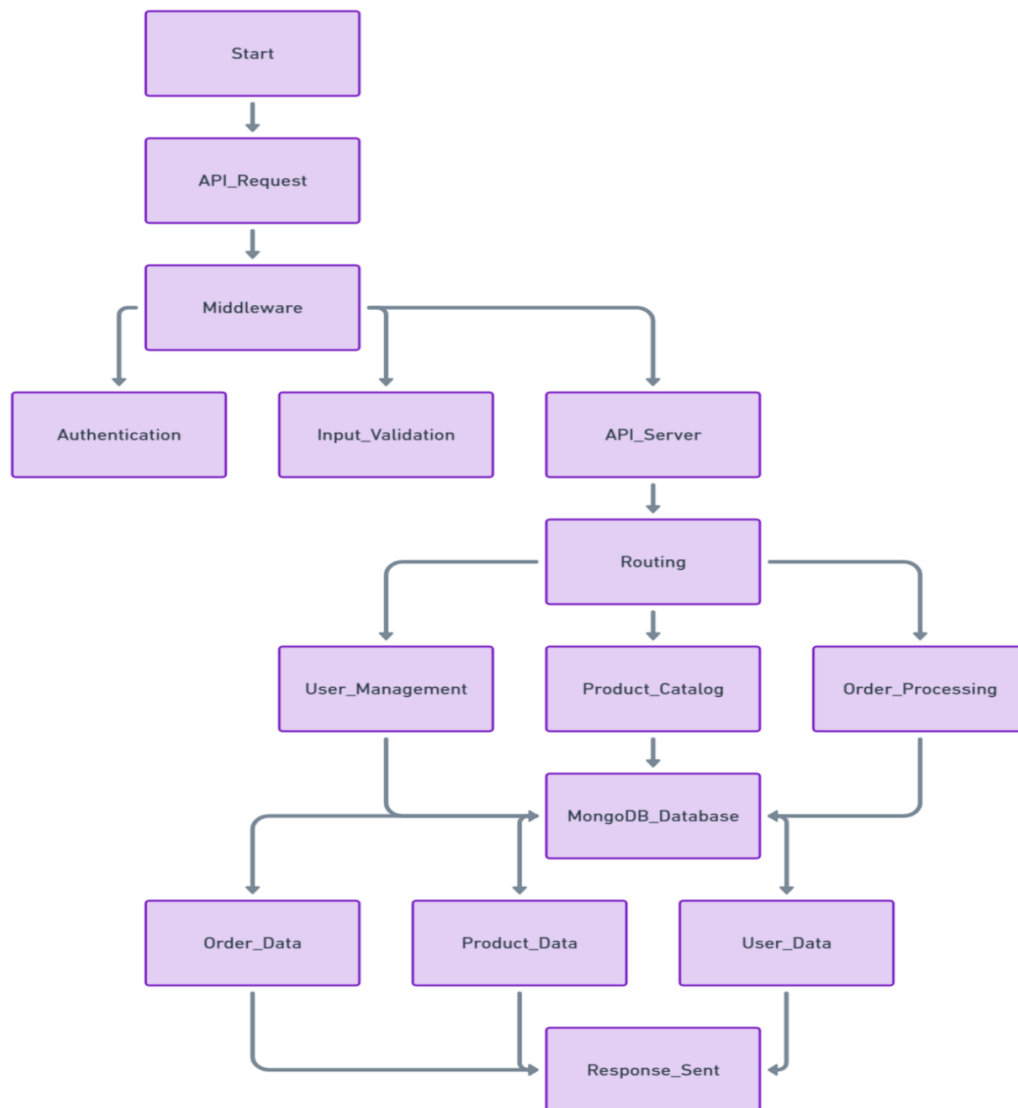
2. For Merchants:

- Simplified inventory and sales management.
- Data-driven insights to enhance business strategies.
- Increased customer engagement through tailored promotions and notifications.

3. Architecture Diagram :



Backend Architecture :



Technology Stack

- Frontend: React.js, Redux, Bootstrap/CSS
- Backend: Node.js, Express.js
- Database: MongoDB
- Authentication: JWT for secure login and API access.
- DevOps: Docker, CI/CD for deployment.

4. Setup Instructions:

Prerequisites

Before setting up the application, ensure your system has the following installed and configured:

Node.js

- **Purpose:** Node.js is a runtime environment that allows you to execute JavaScript code on the server-side.
- **Installation:**
 - Download and install Node.js from [Node.js Official Website](#).
 - Select the **LTS (Long-Term Support)** version for stability.

Verify Installation: After installation, check the version using the terminal:

```
node -v
npm -v
```

- **Output:** This will display the version numbers of Node.js and npm (Node Package Manager), confirming successful installation.

MongoDB

- **Purpose:** MongoDB is a NoSQL database used to store structured data for users, products, orders, and more.
- **Installation:**
 - Install MongoDB from [MongoDB Official Website](#).
 - After installation, ensure MongoDB service is running.

Verify Installation: Use the following command in your terminal:

```
mongo --version
```

- **Output:** Displays the installed version of MongoDB.

Git

- **Purpose:** Git is a version control system that allows you to clone, manage, and update the project repository.
- **Installation:**
 - Download and install Git from [Git Official Website](#).

Verify Installation: Check the Git version using the command:

```
git --version
```

Code Editor (*Optional but Recommended*)

- **Purpose:** A code editor such as **Visual Studio Code** simplifies development with built-in debugging, syntax highlighting, and Git integration.
- **Installation:**
 - Download and install Visual Studio Code from [VS Code Official Website](#).

Cloning the Repository

1. Open a terminal or command prompt on your system.

Navigate to the directory where you want to store the project. For example:

```
cd ~/projects
```

2. Clone the repository using Git:

```
git clone  
https://github.com/your-username/shopez.git
```

3. Navigate into the project folder:

```
cd shopez
```

Backend Setup

Step 1: Navigate to the Backend Directory

Move to the backend folder where server-side logic is located:

```
cd backend
```

Step 2: Install Dependencies

Install all required backend dependencies by running:

```
npm install
```

This command reads the `package.json` file and installs all necessary packages, such as:

- **Express.js:** For building server-side APIs.
- **Mongoose:** For MongoDB database interactions.
- **JWT:** For secure user authentication.

Step 3: Set Up Environment Variables

Environment variables are used to store sensitive information, such as database URIs and JWT secrets.

Create a `.env` file in the `backend` folder by copying the example file:

```
cp .env.example .env
```

1. Open the `.env` file and configure it as follows:

```
PORT=5000
```

```
MONGODB_URI=mongodb://localhost:27017/shopez
```

```
JWT_SECRET=your_jwt_secret_key
```

2. **PORT:** The port on which the backend server will run (default: `5000`).
 - **MONGODB_URI:** MongoDB connection string. If using a local database, use `mongodb://localhost:27017/shopez`.
 - **JWT_SECRET:** A secure string used for signing JWT tokens. Generate a random string for this.

Step 4: Start the Backend Server

Start the backend server by running:

```
npm start
```

- **Default Behavior:** The server starts on `http://localhost:5000`.
- **Testing the API:** Use tools like **Postman** or **Insomnia** to test backend endpoints, such as `/api/auth/login`.

Frontend Setup

Step 1: Navigate to the Frontend Directory

Switch to the frontend folder containing the React.js code for the user interface:

```
cd ../frontend
```

Step 2: Install Dependencies

Install all required frontend dependencies by running:

```
npm install
```

This installs packages like:

- **React.js:** For building the user interface.
- **Axios:** For making HTTP requests.
- **Redux:** For managing application state.

Step 3: Set Up Environment Variables

(Optional) If your frontend requires backend API URLs or other configurations, create a `.env` file in the `frontend` directory:

Create the file:

```
touch .env
```

3. Add the backend API URL:

```
REACT_APP_API_URL=http://localhost:5000
```

Step 4: Start the Frontend Server

Start the React development server by running:

```
npm start
```

- **Default Behavior:** The frontend runs on `http://localhost:3000`.
- Open this URL in your browser to interact with the application.

Testing the Application

4. **Access the Frontend:**

Open a web browser and navigate to `http://localhost:3000`. Verify key pages, such as:

- Homepage
- Product Listing
- Cart and Checkout

5. **Access the Backend:**

Test backend API endpoints with tools like **Postman**. For example:

- `/api/products`: Retrieves the product catalog.

```
0 /api/orders: Processes customer orders.
```

6. Frontend and Backend Communication:

Ensure the frontend communicates with the backend by verifying:

- 0 Product data is loaded from the database.
- 0 Orders are successfully placed and reflected in the backend.

Stopping the Servers

To stop the servers:

- **Frontend:** Press `Ctrl + C` in the terminal running the frontend.
- **Backend:** Press `Ctrl + C` in the terminal running the backend.

Troubleshooting Common Issues

Issue 1: CORS Errors

- **Problem:** The frontend cannot make requests to the backend due to Cross-Origin Resource Sharing (CORS) restrictions.

Solution: Ensure CORS is configured in the backend:

javascript

```
const cors = require('cors');  
app.use(cors());
```

Issue 2: MongoDB Connection Issues

- **Problem:** The backend cannot connect to MongoDB.

Solution: Verify MongoDB is running locally:

```
mongo --eval "db.runCommand({ connectionStatus: 1 })"
```

Issue 3: Missing Environment Variables

- **Problem:** The server fails to start due to missing `.env` configurations.
- **Solution:** Double-check the `.env` file in both `backend` and `frontend` directories.

Issue 4: Port Conflicts

- **Problem:** Another application is using the same port.
- **Solution:** Change the `PORT` in the `.env` file to an available one (e.g., `5001`).

Accessing the Fully Functional Application

Once both the backend (`http://localhost:5000`) and frontend (`http://localhost:3000`) servers are running:

7. Open the browser and navigate to `http://localhost:3000`.
8. Perform end-to-end tests:
 - Register a user.
 - Browse products.
 - Add items to the cart.
 - Place an order.
 - Verify the order in the admin dashboard.

Folder Structure:

Root Directory

This is the base of the project, containing:

- Folders:
 - `backend`: Manages server-side logic and data processing.
 - `frontend`: Contains all the UI components and client-side code.
- Files:
 - `Readme.md`: Project documentation for understanding the setup and usage.

Backend Folder

Path: `/backend`

Purpose: Contains the server-side logic, API routes, controllers, and models.

Key Subfolders:

1. `public/uploads`:
Stores media files, such as product images, uploaded by users or administrators.
 - Contains `.gitkeep` to ensure the folder is kept in version control.
2. `src`: Contains the core application logic for backend operations.
 - Subfolders:
 - `config`:
 - Holds database connection settings like `db.js` to connect to MongoDB.
 - `controllers`:
 - Manages business logic for handling requests (e.g., `category.controller.js`, `order.controller.js`).
 - `middlewares`:

- Reusable functions like `auth.middleware.js` (for JWT authentication).
- `models:`
 - Defines Mongoose schemas for MongoDB collections (e.g., `user.model.js`, `product.model.js`).
- `routes:`
 - Defines RESTful API routes for handling requests (e.g., `user.routes.js`, `product.routes.js`).
- `utils:`
 - Helper functions like `createToken.js` (for generating JWT tokens).

Important Files:

- `index.js`: Entry point for starting the backend server.
- `package.json`: Defines backend dependencies (e.g., Express.js, Mongoose).
- `vercel.json`: Configuration for deployment (e.g., on Vercel).

Frontend Folder

Path: `/frontend`

Purpose: Manages the user interface and client-side logic built with React.

Key Subfolders:

1. `public`: Contains static assets, such as images (e.g., `shopsphere.png`).
2. `src`: The main folder for React components, page layouts, and Redux state management.
 - 0 Subfolders:
 - `Icons`:
Reusable SVG icons (e.g., `EyeFilledIcon.jsx`).
 - `components`:
Shared UI components (e.g., `Header.jsx`, `Modal.jsx`).
 - `pages`:
Components for specific pages (e.g., `Login.jsx`, `Cart.jsx`).
 - `redux`:
Manages the application state using Redux.
 - `api`: Contains Redux slices for API communication (e.g., `orderApiSlice.js`).
 - `features`: Contains slices for specific app features (e.g., `cartSlice.js`).
 - `utils`:
Helper functions for tasks like cart management (`cartUtils.js`).

Important Files:

- `App.jsx`: Main React component that handles routing and layout.
- `package.json`: Manages frontend dependencies (e.g., React, Redux).
- `tailwind.config.js`: Configuration for styling using TailwindCSS.

Purpose of the Structure:

1. Separation of Concerns: Frontend and backend are separated, each focusing on their respective tasks (UI and API).
2. Modularity: Each feature (authentication, products, cart) has its own folder, allowing easy maintenance and future expansion.
3. Reusability: Common components like icons, headers, and buttons are reusable throughout the app.

5. Running the Application:

1. Starting the Backend (Server)

Navigate to the Backend Directory: Open a terminal/command prompt and navigate to the **backend** folder where the server-side code resides:

```
cd backend
```

1. **Start the Backend Server:** In the terminal, run the following command to start the backend server:

```
npm start
```

2. This will start the backend on <http://localhost:5000> (or the port specified in the `.env` file).
 - The server will listen for incoming API requests from the client (frontend).
3. **Backend Operations:**
 - Handles user authentication (login/registration).
 - Manages product, order, and payment-related operations.
 - Connects to the database (MongoDB).
 - Exposes RESTful APIs for frontend interactions.
4. **Verify Backend is Running:** Open a browser or use a tool like **Postman** to check if the backend is running:
 - **Test API Endpoint:** Open <http://localhost:5000/api/products> (or your backend API URL) to check if the products data is being returned correctly.
 - If you see a list of products (or an appropriate API response), the backend is functioning properly.

2. Starting the Frontend (Client)

Navigate to the Frontend Directory: Open a new terminal window and navigate to the **frontend** folder:

```
cd frontend
```

1. **Start the Frontend Server:** Run the following command to start the frontend development server:

```
npm start
```

2. This will start the frontend on <http://localhost:3000> by default.
 - The frontend application will use the APIs provided by the backend to display data like products, orders, and user information.
3. **Verify Frontend is Running:**
 - Open a browser and navigate to <http://localhost:3000>.
 - You should see the homepage of the **Shopsphere E-Commerce Application**.
 - If the application loads correctly, it means the frontend is functioning properly.

3. Interacting with the Application

Once both the backend and frontend servers are running, you can interact with the application by navigating through various features:

1. **Homepage:**
 - Displays featured products and allows browsing through categories.
2. **Product Pages:**
 - View individual product details, including images, descriptions, and pricing.
 - Add products to the cart.
3. **User Authentication:**
 - Register a new user or log in with an existing account.
 - Once logged in, access personalized features like order history and profile settings.
4. **Shopping Cart:**
 - Add products to the cart and proceed to checkout.
 - You can view, modify, and remove items from the cart.
5. **Checkout and Payment:**
 - Complete the order by entering shipping details and making a payment via the integrated payment system (e.g., Razorpay).

4. Stopping the Servers

To stop the backend and frontend servers:

- For the **backend** server: In the terminal, press **Ctrl + C** where the server is running.
- For the **frontend** server: In the terminal, press **Ctrl + C** where the React app is running.

5. Real-time Changes

Both the backend and frontend use **hot-reloading** to reflect changes in real-time:

- When you modify the frontend code (e.g., React components), the changes will automatically be reflected in the browser without needing to restart the server.
- Similarly, if you make changes to the backend (e.g., update API logic), restart the backend server by stopping and starting it again with `npm start`.

6. API Documentation:

Base URL

The base URL for all API requests is: `http://localhost:5000/api/`

Authentication API

The authentication API manages user registration and login, providing JWT tokens for secure access.

1. User Registration

- **Endpoint:** `POST /users/register`
- **Description:** Registers a new user by providing their details (name, email, password).

Request Body:

```
{
  "name": "John Doe",
  "email": "johndoe@example.com",
  "password": "password123"
}
```

- **Response:**
0 **Status:** 201 (Created)

Body:

```
{
  "message": "User registered successfully",
  "user": {
    "id": "user_id",
    "name": "John Doe",
    "email": "johndoe@example.com"
  }
}
```

- **Errors:**

- **400 Bad Request:** Missing fields or invalid data (e.g., weak password).

2. User Login

- **Endpoint:** `POST /users/login`
- **Description:** Authenticates a user and returns a JWT token for future requests.

Request Body:

```
{
  "email": "johndoe@example.com",
  "password": "password123"
}
```

- **Response:**

- **Status:** 200 (OK)

Body:

```
{
  "message": "Login successful",
  "token": "jwt_token_here"
}
```

- **Errors:**

- **400 Bad Request:** Missing or incorrect credentials.

Product API

The product API allows you to view, add, update, and delete products.

3. Get All Products

- **Endpoint:** `GET /products`

- **Description:** Retrieves a list of all available products.
- **Response:**
 - **Status:** 200 (OK)

Body:

```
[
  {
    "id": "product_id",
    "name": "Product Name",
    "price": 99.99,
    "description": "Product description",
    "image": "image_url",
    "category": "Category Name"
  },
  ...
]
```

4. Get Product by ID

- **Endpoint:** `GET /products/:id`
- **Description:** Retrieves details of a specific product by its ID.
- **URL Params:**
 - `id`: The ID of the product.
- **Response:**
 - **Status:** 200 (OK)

Body:

```
{
  "id": "product_id",
  "name": "Product Name",
  "price": 99.99,
  "description": "Product description",
  "image": "image_url",
  "category": "Category Name"
}
```

5. Add a New Product

- **Endpoint:** `POST /products`
- **Description:** Adds a new product to the catalog (Admin only).

Request Body:

```
{
  "name": "New Product",
  "price": 49.99,
  "description": "Product description",
  "image": "image_url",
  "category": "Category Name"
}
```

- **Response:**

```
0   Status: 201 (Created)
```

Body:

```
{
  "message": "Product added successfully",
  "product": {
    "id": "product_id",
    "name": "New Product",
    "price": 49.99,
    "description": "Product description",
    "image": "image_url",
    "category": "Category Name"
  }
}
```

6. Update Product

- **Endpoint:** `PUT /products/:id`
- **Description:** Updates an existing product by its ID (Admin only).
- **URL Params:**
 - 0 `id`: The ID of the product.

Request Body:

```
{
  "name": "Updated Product",
  "price": 59.99,
  "description": "Updated description",
  "image": "updated_image_url",
  "category": "Updated Category"
}
```

- **Response:**
 - **Status:** 200 (OK)

Body:

```
{
  "message": "Product updated successfully",
  "product": {
    "id": "product_id",
    "name": "Updated Product",
    "price": 59.99,
    "description": "Updated description",
    "image": "updated_image_url",
    "category": "Updated Category"
  }
}
```

7. Delete Product

- **Endpoint:** `DELETE /products/:id`
- **Description:** Deletes a product from the catalog by its ID (Admin only).
- **URL Params:**
 - `id`: The ID of the product to delete.
- **Response:**
 - **Status:** 200 (OK)

Body:

```
{
  "message": "Product deleted successfully"
}
```

- **Errors:**
 - `404 Not Found`: Product with the specified ID does not exist.

Order API

The order API handles the creation and management of customer orders.

8. Create Order

- **Endpoint:** `POST /orders`
- **Description:** Creates a new order for a logged-in user.

Request Body:

```
{
  "user_id": "user_id",
  "products": [
    {
      "product_id": "product_id",
      "quantity": 2
    }
  ],
  "total_amount": 199.98,
  "shipping_address": "123 Main St, City, Country"
}
```

- **Response:**

- **Status:** 201 (Created)

Body:

```
{
  "message": "Order created successfully",
  "order": {
    "id": "order_id",
    "user_id": "user_id",
    "products": [
      {
        "product_id": "product_id",
        "quantity": 2
      }
    ],
    "total_amount": 199.98,
    "status": "Pending",
    "shipping_address": "123 Main St, City, Country"
  }
}
```

9. Get Order by ID

- **Endpoint:** `GET /orders/:id`
- **Description:** Retrieves the details of a specific order by its ID.
- **URL Params:**
 - `id`: The ID of the order.
- **Response:**

0 **Status:** 200 (OK)

Body:

```
{
  "id": "order_id",
  "user_id": "user_id",
  "products": [
    {
      "product_id": "product_id",
      "quantity": 2
    }
  ],
  "total_amount": 199.98,
  "status": "Pending",
  "shipping_address": "123 Main St, City, Country"
}
```

10. Get All Orders (Admin Only)

- **Endpoint:** `GET /orders`
- **Description:** Retrieves a list of all orders for the admin.
- **Response:**
 - 0 **Status:** 200 (OK)

Body:

```
[
  {
    "id": "order_id",
    "user_id": "user_id",
    "products": [
      {
        "product_id": "product_id",
        "quantity": 2
      }
    ],
    "total_amount": 199.98,
    "status": "Pending",
    "shipping_address": "123 Main St, City, Country"
  },
  ...
]
```


Payment API

The payment API manages order payments.

11. Create Payment

- **Endpoint:** `POST /payments`
- **Description:** Initiates a payment for an order (integrated with payment gateways like Razorpay).

Request Body:

```
{  
  "order_id": "order_id",  
  "amount": 199.98,  
  "payment_method": "razorpay"  
}
```

- **Response:**
 - **Status:** 200 (OK)

Body:

```
{  
  "message": "Payment initiated successfully",  
  "payment_url": "razorpay_payment_url"  
}
```

7. Authentication and Authorization:

1. Authentication

Authentication is the process of verifying the identity of a user. In simpler terms, it ensures that the user is who they say they are.

Authentication Flow in Shoppere E-Commerce Application

1. User Registration:

A new user registers with their credentials (e.g., name, email, password).

- The password is hashed using **bcrypt** (a hashing algorithm) before being stored in the database for security reasons.

2. User Login:

When a user logs in, their credentials (email and password) are sent to the backend.

- The password is compared with the stored hash in the database.
 - If the credentials are valid, a **JWT (JSON Web Token)** is generated and sent back to the client. This token is used to authenticate the user on future requests.
3. **JWT (JSON Web Token):**
- JWT is a compact, URL-safe token that is used for securely transmitting information between the client and server.
- **JWT Structure:** It consists of three parts:
 - **Header:** Contains metadata about the token (algorithm, token type).
 - **Payload:** Contains claims, or the data that is transmitted (e.g., user ID, email).
 - **Signature:** Ensures that the token has not been tampered with. It is created using a secret key.
 - The client stores the JWT token (usually in **localStorage** or **cookies**) and includes it in the **Authorization header** of subsequent API requests to authenticate themselves.
4. **Token Expiry and Refresh:**
- JWTs usually have an expiry time (e.g., 1 hour). When the token expires, the client will need to reauthenticate.
 - Some systems implement a **refresh token** system, where a secondary token (with a longer lifespan) is used to obtain a new access token.

Authentication Process:

- **Registration API (POST /users/register):**
 - Input: User's details (name, email, password).
 - Server validates the data, hashes the password, and stores it in the database.
 - A success message is sent back to the user.
- **Login API (POST /users/login):**
 - Input: User's email and password.
 - The server checks if the user exists and verifies the password. If valid, a JWT is issued.
 - Output: A JWT is returned to the client to be used in future requests.

2. Authorization

Authorization is the process of determining what an authenticated user is allowed to do. In simple terms, it defines what resources a user can access and what actions they can perform.

Role-Based Access Control (RBAC)

In **Shopsphere E-Commerce Application**, **role-based access control (RBAC)** is used for managing what actions users can perform based on their roles (e.g., customer, admin).

- **User Roles:**

- **Admin:** Can perform all actions, such as managing products, categories, orders, and users.
- **Customer:** Can only view products, add items to the cart, and place orders.

Implementing Authorization

1. Protected Routes:

Authorization is implemented by protecting certain API routes that require the user to have a specific role. For example:

- Only admins can access product management routes, such as **POST** `/products` (add new products).
- Customers cannot access admin-specific pages, like viewing other users' information or managing products.

2. Middleware for Role Check: The backend uses middleware functions to verify whether a user has the appropriate role to access certain routes.

- **Example:** The `auth.middleware.js` checks if a user has a valid JWT, and the `role.middleware.js` checks if the user's role is **admin**.

Role Check Example:

```
const authorizeRole = (role) => {

  return (req, res, next) => {

    if (req.user.role !== role) {

      return res.status(403).json({ message: 'Access Denied:
Insufficient permissions' });

    }

    next();

  };

};

// Usage in route

app.post('/products', authMiddleware, authorizeRole('admin'),
productController.addProduct);
```

3. **Accessing Resources:** When a user makes a request to a protected API route, the backend checks the JWT token, extracts the user's role from it, and compares it against the required role for that endpoint.

3. JWT Authentication Middleware

To ensure that only authenticated users can access protected resources, the backend uses JWT middleware to validate incoming requests.

JWT Validation Middleware (`auth.middleware.js`)

The middleware intercepts API requests and verifies if the request contains a valid JWT token in the **Authorization header**.

- **Process:**

- The JWT token is passed in the HTTP request header:
`Authorization: Bearer <JWT_Token>`
- The middleware verifies the token's signature using the secret key.
- If the token is valid, the user's details (e.g., user ID, role) are decoded and attached to the request object (`req.user`).
- If the token is invalid or missing, a 401 Unauthorized response is sent.

Example of JWT Middleware:

```
const jwt = require('jsonwebtoken');

const authMiddleware = (req, res, next) => {

  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {

    return res.status(401).json({ message: 'Access Denied: No token provided' });

  }

  try {
```

```

    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    req.user = decoded; // Attach user info to the request object

    next();

  } catch (err) {

    res.status(401).json({ message: 'Access Denied: Invalid token'
  });

  }

};

module.exports = authMiddleware;

```

Authorization Middleware

After authenticating the user, the application checks if they have the necessary role to access a particular route.

- **Example:** If the route is restricted to **admin** users, the middleware checks the role stored in `req.user.role` to ensure it matches the required role.

Role Authorization Middleware:

```

const authorizeRole = (role) => {

  return (req, res, next) => {

    if (req.user.role !== role) {

      return res.status(403).json({ message: 'Access Denied: You do
not have the required role' });

    }

    next();

  };

};

```

```
module.exports = authorizeRole;
```

4. Authorization in Action

Let's see how **authentication** and **authorization** work together:

Example: Admin-Only Endpoint

In this scenario, the endpoint is **POST** `/products`, which allows an admin to add a product. This route should be accessible only by users with the **admin** role.

- **Steps:**
 1. The user sends a **POST** request to `/products` with a valid JWT in the **Authorization header**.
 2. The backend first checks if the JWT is valid (authentication).
 3. Then, the backend checks if the decoded user role is **admin** (authorization).
 4. If both checks pass, the user is allowed to add a new product.

Example Route:

```
app.post('/products', authMiddleware, authorizeRole('admin'),  
productController.addProduct);
```

Example: Customer-Only Endpoint

On the other hand, endpoints such as **GET** `/orders` should be restricted to customers. Only the user who created the order should be able to access it.

Steps:

1. The user sends a **GET** request to `/orders/:orderId`.
2. The backend first checks if the JWT is valid (authentication).
3. The backend checks if the user is the owner of the order (authorization).
4. If both checks pass, the order is returned.

5. Summary of Authentication and Authorization

- **Authentication** ensures that the user is who they claim to be. This is typically done using **JWT tokens**.
- **Authorization** ensures that an authenticated user has the right permissions to access specific resources or perform certain actions. This is usually done using **role-based access control** (RBAC).

- The **JWT Middleware** verifies the token, while **Authorization Middleware** checks the user's role to grant access to protected routes.

8. Testing:

1. Unit Testing

Unit testing focuses on testing individual components or functions in isolation. It ensures that each unit of code performs its intended function correctly.

Backend Unit Tests

Backend testing will verify the core business logic, such as user authentication, JWT token generation, and product management.

- **Tools: Mocha, Chai, Jest**

Example Unit Tests:

1. **User Registration:**
 - Verify that the server correctly hashes the password and stores the user.
2. **JWT Token Generation:**
 - Test if the token is correctly signed and has the right payload.
3. **Product Management:**
 - Verify that products are correctly added, updated, and deleted.

2. Integration Testing

Integration tests check the interaction between multiple components, such as the backend API, database, and the frontend.

Backend Integration Tests

- **Tools: Mocha, Supertest, Chai**

Example: Testing Product Creation

1. Test the API endpoint to ensure a product is correctly added to the database and returned in the response.

Frontend Integration Tests

- **Tools: Jest, React Testing Library**

3. End-to-End (E2E) Testing

End-to-end tests simulate real-world user behavior and test the entire flow of the application from the frontend to the backend.

- **Tools: Cypress, Selenium, Puppeteer**

Frontend E2E Testing

- Test the entire flow of actions a user might take on the website:
 - **Login, Add to Cart, Checkout, Place Order.**

4. Performance Testing

Performance testing verifies that the application can handle high traffic, heavy loads, and large amounts of data without compromising on performance.

- **Tools: Artillery, JMeter**

Example: Load Testing the API

1. Test how the backend performs under heavy load when many users try to access product data at the same time.

5. Security Testing

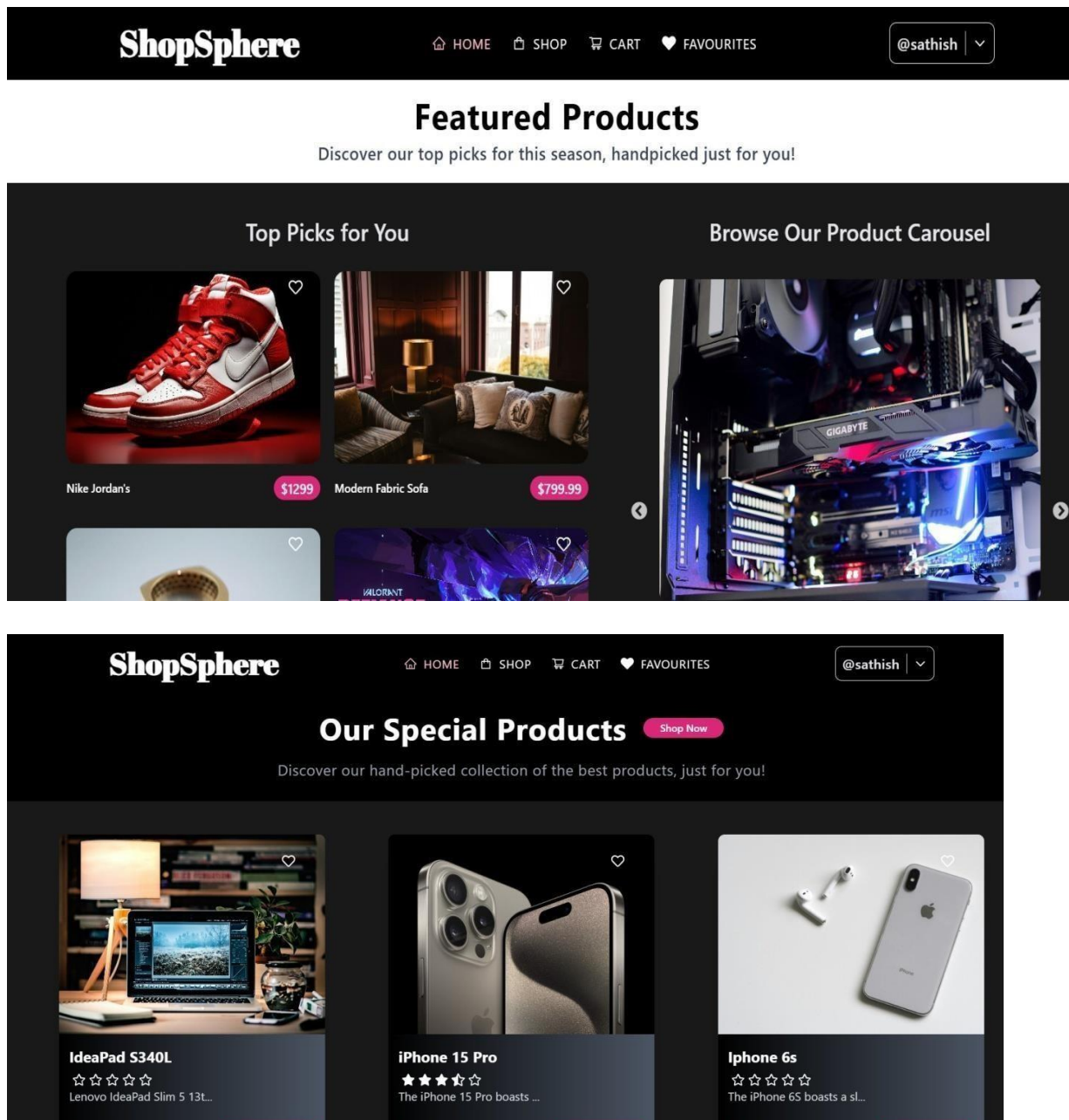
Security testing verifies that the application is secure from vulnerabilities such as **SQL injection**, **XSS**, and **CSRF** attacks.

- **Tools: OWASP ZAP, Snyk**

Focus Areas:

1. **JWT Validation:**
 - Ensure that JWT tokens are validated correctly and that unauthorized users cannot access protected routes.
2. **Cross-Site Scripting (XSS):**
 - Test input fields for user data to ensure that HTML or JavaScript cannot be injected into the application.
3. **SQL Injection:**
 - Ensure that the application does not directly interact with raw SQL queries or is vulnerable to SQL injection.

9. Screenshots:



ShopSphere

[HOME](#) [SHOP](#) [CART](#) [FAVOURITES](#)

@sathish

Filter by Categories

☐ Clothing

☐ Electronics

☐ Furniture

☐ Accessories

☐ Automobile

☐ Sports

☐ Beverages

☐ Instruments

☐ Fragrances

☐ Food

☐ Skincare

☐ Lighting

☐ Luggage

PRODUCTS (34)

IdeaPad S340L

\$549.99

Lenovo IdeaPad Slim 5 13th Gen ...

Read More →

iPhone 15 Pro

\$999.99

The iPhone 15 Pro boasts cutti ...

Read More →

iPhone 6s

\$199.99

The iPhone 6S boasts a sleek d ...

Read More →

Lamborghini Aventador

\$517,770.00

The Lamborghini Aventador is a ...

Read More →

DEFIANCE

ShopSphere

[HOME](#) [SHOP](#) [CART](#) [FAVOURITES](#)

@sathish

SHOPPING CART

iPhone 15 Pro

Apple Phone

\$ 999.99

1

Nike Jordan's

Nike

\$ 1299

1

IdeaPad S340L

Lenovo

\$ 549.99

1





AirPods Max

Apple

\$ 699

1

34

ShopSphere				
HOME SHOP CART FAVOURITES @sathish				
IMAGE	PRODUCT	QUANTITY	PRICE	TOTAL
	iPhone 15 Pro	1	\$ 999.99	\$ 999.99
	Nike Jordan's	1	\$ 1299.00	\$ 1299.00
	IdeaPad S340L	1	\$ 549.99	\$ 549.99
	AirPods Max	1	\$ 699.00	\$ 699.00

10. Known Issues:

1. Authentication and Authorization Issues

JWT Token Expiry

- **Description:** Users may experience issues when the JWT token expires. After the token expires (usually after 1 hour), users may find themselves logged out unexpectedly and unable to access protected routes until they reauthenticate.
- **Cause:** JWT tokens have a built-in expiration time for security purposes.
- **Solution:** Implement **refresh tokens** in the system to allow users to automatically refresh their JWT token without having to log in again. This can be done by storing a **longer-lasting refresh token** in the client and issuing a new access token when the original expires.

Incorrect Role Handling

- **Description:** Users with the correct role (e.g., admin) may experience difficulties accessing protected routes or actions (e.g., managing products or orders).

- **Cause:** There may be issues with the role checking logic in the backend, especially if the role is not properly encoded in the JWT or if the authorization middleware is not correctly applied.
- **Solution:** Verify that the role is properly assigned to the user during login and that the authorization middleware checks the role correctly before allowing access to certain routes. Ensure the token is correctly validated and decoded in all relevant places.

Session Persistence

- **Description:** Users may be logged out unexpectedly when navigating between pages or after a page refresh, especially if JWT tokens are not stored securely.
- **Cause:** Insecure handling of tokens in **localStorage** or **cookies** could result in loss of session data.
- **Solution:** Store JWT tokens in **HTTP-only cookies** to improve security and persistence. HTTP-only cookies cannot be accessed by JavaScript, which reduces the risk of cross-site scripting (XSS) attacks.

2. Product and Cart Management Issues

Cart Not Updating Properly

- **Description:** The cart does not reflect changes correctly (e.g., adding/removing products, changing quantities).
- **Cause:** This can occur due to state management issues or incorrect API calls. For instance, the cart state in **Redux** may not be updated after an item is added or removed.
- **Solution:** Ensure that the **Redux store** is properly updated after each cart action, and that the frontend is synchronized with the backend. Make sure **cart APIs** handle the logic of adding/removing items and updating quantities correctly.

Product Stock Not Decreasing After Checkout

- **Description:** The stock for a product is not correctly updated after a successful checkout, leading to over-selling.
- **Cause:** There might be an issue with the API that updates the product stock in the database after an order is placed.
- **Solution:** Ensure that the order API updates the **product stock** correctly after a successful purchase. This should be done within a **transaction** to prevent race conditions where two users might buy the last item at the same time.

3. Payment Issues

Payment Gateway Integration Failures

- **Description:** Payments may fail to process due to errors in integration with external payment gateways (e.g., Razorpay, Stripe).
- **Cause:** Issues could arise from incorrect API keys, invalid data passed to the payment gateway, or network issues.
- **Solution:** Double-check the payment gateway API integration, including the correct API keys, payload formatting, and error handling. Ensure proper validation of the payment response before confirming the order.

Payment Confirmation Delay

- **Description:** There is a delay in receiving payment confirmation, which leads to orders being marked as "pending" or "unpaid."
- **Cause:** This might be due to network delays or asynchronous API calls not being properly handled.
- **Solution:** Implement **asynchronous processing** of payment confirmation in the backend. This can be done by handling the payment confirmation in a background job or through webhooks provided by the payment gateway.

4. Performance and Load Issues

Slow Page Load Times

- **Description:** Some pages, especially those with many product listings or large images, may load slowly.
- **Cause:** This could be due to unoptimized images, excessive data being loaded, or inefficient API queries.
- **Solution:**
 - Use **image compression** to reduce the size of product images without sacrificing quality.
 - Implement **pagination** or **lazy loading** for product listings to reduce the initial data load.
 - Optimize backend queries by indexing relevant database fields (e.g., product names, categories).

API Endpoint Performance

- **Description:** Some backend API endpoints may take longer than expected to respond, especially when retrieving large datasets or performing complex calculations.
- **Cause:** Inefficient database queries, unoptimized code, or lack of caching.
- **Solution:**
 - Use **database indexing** to optimize query performance.
 - Implement **caching** (e.g., Redis) for frequently requested data (e.g., product listings, categories).
 - Profile and optimize backend code for bottlenecks.

5. Security Issues

Cross-Site Scripting (XSS)

- **Description:** User input may not be properly sanitized, allowing for **cross-site scripting** attacks where malicious scripts are injected into the web page.
- **Cause:** Insufficient input sanitization and validation, especially in forms such as user registration and product reviews.
- **Solution:** Sanitize all user input on both the frontend and backend using libraries such as **DOMPurify** to prevent malicious scripts from being executed in the browser.

SQL Injection

- **Description:** The application may be vulnerable to **SQL injection** attacks, allowing attackers to execute arbitrary SQL queries by injecting malicious input.
- **Cause:** Direct usage of user input in SQL queries without proper validation or sanitization.
- **Solution:** Use **parameterized queries** or **ORMs** (e.g., **Mongoose** for MongoDB) to prevent SQL injection vulnerabilities. Never concatenate user input directly into SQL queries.

Insecure API Endpoints

- **Description:** Some API endpoints may be left exposed, allowing unauthorized users to access sensitive data (e.g., user profiles, order details).
- **Cause:** Lack of proper authentication or authorization checks on certain API routes.
- **Solution:** Ensure that all **protected routes** are behind authentication and authorization checks. Use **JWT** for securing API endpoints, and implement **role-based access control** (RBAC) for access control.

6. User Interface (UI) and User Experience (UX) Issues

Layout and Responsiveness Issues

- **Description:** The application layout may break or not scale properly on different devices, especially on mobile.
- **Cause:** CSS styles may not be responsive or tested across different screen sizes.
- **Solution:** Use **responsive design principles** (e.g., CSS Grid, Flexbox) and test on various screen sizes. Implement a mobile-first approach, using **media queries** to adjust the layout based on the screen size.

Accessibility Issues

- **Description:** Some users may experience difficulties using the application due to accessibility barriers.
- **Cause:** Lack of proper **ARIA** (Accessible Rich Internet Applications) tags, non-semantic HTML, or color contrast issues.

- **Solution:** Ensure the app is **WCAG (Web Content Accessibility Guidelines)** compliant. Use tools like **Lighthouse** or **Axe** to identify and fix accessibility issues.

7. Data Synchronization Issues

Data Mismatch Between Frontend and Backend

- **Description:** The data displayed on the frontend may not match the data in the backend due to synchronization issues.
- **Cause:** This can occur when the frontend and backend are not properly synced, or the API response does not include updated data.
- **Solution:** Ensure that the frontend **polls** or listens for updates when the data changes. Implement real-time updates using technologies like **WebSockets** or **Server-Sent Events (SSE)** for real-time data synchronization.

8. Deployment Issues

Incorrect Configuration During Deployment

- **Description:** The application may fail to work properly in the production environment due to configuration issues (e.g., incorrect environment variables, missing build files).
- **Cause:** Differences between the local development environment and the production environment.
- **Solution:** Ensure that the **production configuration** is properly set, including database URIs, JWT secrets, and API keys. Always test the deployment in a staging environment before going live.

11. Future Enhancements:

As the **Shopsphere E-Commerce Application** continues to evolve, several enhancements can be made to improve the system's functionality, performance, user experience, and scalability. These enhancements will focus on adding new features, improving current functionalities, and ensuring that the system remains competitive and meets the growing demands of users.

Below are some **future enhancements** for the project, categorized based on different areas of focus:

1. User Experience and Interface Enhancements

Personalized User Experience

- **Objective:** Tailor the shopping experience to individual users based on their preferences, browsing history, and purchase patterns.
- **Implementation:**

- Implement **recommendation engines** that suggest products based on users' browsing and purchase history.
- Use **machine learning** algorithms to personalize the homepage, product listings, and ads.
- Introduce **AI-powered chatbots** for better customer service and personalized shopping advice.
- **Dynamic pricing models** that adjust prices based on user behavior, such as offering discounts for loyal customers.

Advanced Search and Filter Options

- **Objective:** Improve the search functionality to help users find products quickly and efficiently.
- **Implementation:**
 - Add **autocomplete suggestions** and **fuzzy search** for better search results.
 - Implement **advanced filtering** based on multiple parameters like price range, brand, ratings, and more.
 - Integrate **voice search** capabilities to enable users to search by voice commands.
 - Enhance search speed with **search indexing** and **Elasticsearch** or **Solr** for handling large product catalogs.

Enhanced Mobile Experience

- **Objective:** Make the platform more mobile-friendly for users on smartphones and tablets.
- **Implementation:**
 - Implement **progressive web apps (PWA)** for offline functionality and faster load times on mobile devices.
 - Enhance **responsive design** to ensure all features work well across devices, improving accessibility for mobile users.
 - Optimize mobile UI for **single-handed navigation** and **thumb-friendly layouts**.

Augmented Reality (AR) for Product Visualization

- **Objective:** Provide users with an interactive and immersive shopping experience.
- **Implementation:**
 - Implement **AR features** where users can see products (e.g., furniture, clothing) in their real-world environment using their smartphone cameras.
 - Use **3D models** for products to give users a better understanding of the product dimensions and features.

2. Performance and Scalability Enhancements

Multi-Region and Multi-Language Support

- **Objective:** Expand the application to multiple regions and languages to cater to a global audience.
- **Implementation:**
 - Add **multi-language support** by implementing **localization** and **internationalization** (i18n and l10n) to allow the site to adapt to different languages and currencies.
 - Set up **multi-region databases** and **CDN (Content Delivery Network)** to improve load times and performance for users in different geographical locations.
 - Integrate **currency conversion** for international users.

Cloud Scalability with Auto-Scaling

- **Objective:** Ensure that the application can scale dynamically to handle traffic spikes, especially during promotions, holidays, or product launches.
- **Implementation:**
 - Move the application to **cloud-based infrastructure** (e.g., AWS, Azure, Google Cloud) that offers automatic scaling.
 - Implement **load balancers** to distribute traffic efficiently across multiple servers and **auto-scaling groups** to spin up additional resources during high traffic periods.
 - Use **serverless architecture** for specific components (e.g., order processing) to reduce overhead costs during low usage periods.

Caching for Improved Performance

- **Objective:** Reduce response times and server load by caching frequently requested data.
- **Implementation:**
 - Use **Redis** or **Memcached** to cache product listings, user sessions, and other frequently accessed data.
 - Implement **content delivery networks (CDNs)** to cache static assets like images, CSS, and JavaScript files globally.
 - Cache API responses for non-dynamic content (e.g., product catalogs, category lists) with cache expiry logic to ensure up-to-date data.

3. Security Enhancements

Two-Factor Authentication (2FA)

- **Objective:** Enhance account security by requiring a second factor for login (e.g., a one-time password sent via SMS or email).
- **Implementation:**

- Integrate **two-factor authentication** using services like **Twilio** for SMS OTP or **Google Authenticator** for time-based codes.
- Allow users to enable and manage 2FA from their account settings.

Secure Payment Integration

- **Objective:** Ensure the platform is fully PCI-DSS compliant and provides a secure checkout process for users.
- **Implementation:**
 - Integrate **secure payment gateways** (e.g., Stripe, PayPal, Razorpay) that are PCI-DSS compliant for handling credit card transactions securely.
 - Implement **tokenization** for sensitive payment data to reduce the risk of data breaches.
 - Use **3D Secure 2.0** to add additional security layers during payment processing.

Real-Time Fraud Detection and Prevention

- **Objective:** Implement a robust fraud detection system to minimize chargebacks and fraudulent transactions.
- **Implementation:**
 - Use **machine learning** models to analyze user behavior and detect anomalies that could indicate fraudulent activity (e.g., sudden high-value purchases, mismatched billing and shipping addresses).
 - Integrate fraud prevention tools like **Signifyd** or **Kount** to automatically flag suspicious transactions.

4. Data Analytics and Insights

Advanced Analytics for Users and Admins

- **Objective:** Provide insights into user behavior, sales trends, and business performance.
- **Implementation:**
 - Integrate **Google Analytics** and **Heatmaps** to understand user interactions, bounce rates, and click patterns.
 - Use **machine learning** for predicting sales trends, product demand, and identifying cross-sell and up-sell opportunities.
 - Implement **business intelligence tools** (e.g., Power BI, Tableau) to allow admins to access detailed reports on sales, traffic, and customer behavior.

Customer Feedback and Reviews

- **Objective:** Enable customers to leave reviews and provide feedback on products and the overall shopping experience.
- **Implementation:**

- Allow users to **rate products** and write **reviews**, with the ability to filter reviews based on different criteria (e.g., rating, date).
- Implement an automated **sentiment analysis** system to analyze reviews and flag negative feedback for follow-up.
- Display **verified buyer badges** to ensure the authenticity of reviews.

5. New Features and Integrations

Social Media Integration

- **Objective:** Increase engagement by allowing users to share products and purchases on social media.
- **Implementation:**
 - Add **social sharing buttons** for platforms like Facebook, Twitter, Instagram, and Pinterest.
 - Allow users to sign in using their social media accounts (**OAuth** integration with Facebook, Google, etc.).

Subscription and Loyalty Programs

- **Objective:** Build customer loyalty by offering rewards, discounts, and subscription services.
- **Implementation:**
 - Integrate **subscription services** that allow customers to subscribe to products on a recurring basis (e.g., monthly deliveries).
 - Implement a **loyalty program** that rewards customers with points or discounts for each purchase, referral, or review.

Voice Commerce Integration

- **Objective:** Enable users to make purchases via voice commands.
- **Implementation:**
 - Integrate **voice commerce capabilities** using platforms like **Amazon Alexa** or **Google Assistant** to allow users to add products to the cart and place orders using their voice.

6. Monitoring and Maintenance

Real-Time Application Monitoring

- **Objective:** Ensure the system is always running smoothly with real-time monitoring and alerting.
- **Implementation:**
 - Implement **monitoring tools** like **New Relic**, **Datadog**, or **Prometheus** to track server health, uptime, and performance.

- Set up **alerts** for critical issues such as downtime, high server load, or failed payment transactions.

Automatic Database Backups

- **Objective:** Ensure that critical data is protected and can be restored in case of failure.
- **Implementation:**
 - Set up **automatic backups** for the MongoDB database using **cron jobs** or cloud-based solutions (e.g., **AWS RDS Snapshots**, **MongoDB Atlas Backups**).