shopez: e-commerce application

PROJECT DOCUMENTATION

**Introduction**

* **Project Title:** shopez: e-commerce application.

MERN Stack For the ecommerce website, we have adopted the MERN stack, which comprises MongoDB, Express.js, React.js, and Node.js. This technology stack allows us to build a dynamic and user-friendly web application that leverages the power of both front-end and back-end technologies.

* **Team Members:**

**Surya Prakash G** – Frontend development

**Santhosh T** - Backend development

**Harish N**- DataBase management

**Mohammed Rizwan F** - Testing and deployment

**Project Overview**

* **Purpose:** The purpose of this project is to develop a dynamic, scalable, and user-friendly eCommerce website using the MERN stack (MongoDB, Express.js, React.js, and Node.js). The platform aims to provide a seamless shopping experience for users while ensuring robust functionality and operational efficiency for administrators. By leveraging modern technologies, the website will cater to the growing demand for online shopping, ensuring accessibility, security, and reliability for both customers and vendors.
* **Key goals include:**

Enabling users to browse, search, and purchase products easily.

Providing secure payment options and ensuring data security.

Offering efficient inventory management and order processing for administrators.

Delivering a responsive and accessible interface for diverse user groups.

* **Features:**

**The eCommerce** website is designed with comprehensive features to address user and business needs effectively:

* **User Registration and Authentication**

Secure user registration with unique credentials.

User login and account management capabilities.

Integration of session-based or token-based authentication for secure access.

**Product Catalog :**

Display of a detailed catalog of products with name, description, price, and images.

Categorized and searchable products for easy navigation.

Real-time updates for product availability and inventory.

**Shopping Cart and Checkout :**

Functionality to add, update, and remove items from the shopping cart.

Comprehensive checkout process including order summary, shipping, and payment.

Integration with third-party payment gateways (e.g., Stripe, PayPal) for secure transactions.

**Order Management :**

Users can view order history and track current orders.

Admin users can manage, process, and update order statuses.

Automated email notifications for order confirmation and updates.

**Product Recommendations :**

Personalized recommendations based on user browsing and purchase history.

Display of trending and related products for enhanced shopping options.

**Wishlist and Favorites :**

Option for users to save products for future reference.

Easy management and sharing of wishlist items.

**Customer Reviews and Ratings :**

Users can leave reviews and ratings for purchased products.

Display of average product ratings to aid other users in decision-making.

**Inventory Management :**

Real-time tracking of product stock levels.

Notifications for out-of-stock or low-quantity items to users and admins

**Architecture**

The architecture of the eCommerce website follows a modular and layered structure based on the MERN stack. This design ensures efficient communication between the front-end, back-end, and database layers while maintaining scalability, performance, and security.

* **Frontend:**

The frontend layer is responsible for user interaction, ensuring a responsive and intuitive interface. It is developed using React.js to deliver a dynamic and seamless user experience.

**Key Components:**

React.js Framework: React enables the creation of reusable and modular components, such as product listings, navigation menus, and checkout forms.

React Router: Handles client-side routing, allowing users to navigate through pages without reloading the application.

**State Management:**

Redux: Centralizes the application's state for predictable state transitions.

Context API: An alternative for smaller-scale applications to manage state efficiently.

Responsive Design: Utilizes CSS frameworks like Bootstrap or custom CSS to ensure compatibility with various devices and screen sizes.

Integration with Back-end APIs: Fetches data from the server (e.g., product catalog, user authentication) using tools like Axios or Fetch API.

**Workflow:**

Users interact with the interface to browse products, manage the shopping cart, and perform transactions.

React handles view rendering using the virtual DOM for efficient updates.

* **Backend:**

The backend serves as the intermediary between the frontend and database. Built using Node.js and Express.js, it handles business logic, API endpoints, and secure data transactions.

**Key Components:**

**Node.js:**

Provides a runtime environment for executing JavaScript on the server side.

**Express.js Framework:**

Simplifies route handling for incoming HTTP requests.

Enables the creation of RESTful APIs for operations like user authentication, product management, and order processing.

**Middleware:**

For parsing requests (e.g., body-parser, JSON parsing).

Authentication middleware to secure routes (e.g., Passport.js or JWT verification).

Error-handling middleware for debugging and maintaining server stability.

Secure Payment Integration: Handles payment processes with third-party gateways like Stripe or PayPal.

Scalability: Designed for horizontal scaling using clusters or containerized environments like Docker.

**Workflow:**

Processes requests from the frontend via RESTful APIs.

Implements business logic, such as validating user data or calculating order totals.

**Database:**

The database layer stores all persistent data, such as user information, product details, and order history. MongoDB, a NoSQL database, is chosen for its flexibility, scalability, and ability to handle unstructured data.

**Key Features**

**Schema Design:**

Users Collection: Stores user profiles, login credentials (hashed passwords), and preferences.

Products Collection: Includes product details (name, description, price, category, images, stock levels).

Orders Collection: Tracks order details (order ID, user ID, products ordered, status, timestamps).

Reviews Collection: Holds product reviews and ratings submitted by users.

Document-Oriented: Uses BSON (Binary JSON) format to store data, ensuring quick access and retrieval.

**Scalability:**

Supports sharding for horizontal scaling as the user base and product catalog grow.

**Data Integrity:**

Implements validation rules at the schema level to ensure accurate data storage.

Uses indexing for faster querying on frequently accessed fields like product name or user ID.

**Backup and Recovery:**

Regular backups to prevent data loss.

Replica sets to ensure data availability even during failures.

**Workflow:**

Stores structured and unstructured data from the backend.

Executes CRUD operations requested by the server.

**Setup Instructions**

This section provides a detailed guide to set up the eCommerce website locally or on a server. It covers prerequisites and installation steps to ensure a smooth setup process.

* **Prerequisites:**

Before proceeding with the installation, ensure the following requirements are met:

**System Requirements:**

Operating System: Windows, macOS, or Linux.

Hardware: Minimum of 4 GB RAM, 20 GB free disk space.

**Software Requirements:**

**Node.js:**

Install Node.js (v14 or later) from Node.js official website.

Verify installation:

bash

Copy code

node -v

npm -v

**MongoDB:**

Install MongoDB Community Edition from MongoDB official website.

Start the MongoDB service:

bash

Copy code

mongod

Alternatively, use MongoDB Atlas for a cloud-hosted database.

**Git:**

Install Git from Git official website.

**Verify installation:**

bash

Copy code

git --version

**Package Manager:**

Install npm (comes with Node.js) or Yarn for dependency management.

Optional Tools:

Code Editor: Visual Studio Code is recommended for development.

Postman: For API testing and debugging.

Docker: For containerized deployment (if needed).

**Installation:**

Follow these steps to set up the project on your local machine:

**Step 1:** Clone the Repository

Open your terminal or command prompt.

Clone the project repository:

git clone <repository-url>

Navigate to the project directory:

cd <project-folder-name>

**Step 2:** Install Dependencies

Install server-side dependencies:

cd backend

npm install

Install client-side dependencies:

cd ../frontend

npm install

**Step 3:** Set Up Environment Variables

Create a .env file in the backend directory.

Add the following environment variables:

env

PORT=5000

MONGO\_URI=<your-mongodb-connection-string>

JWT\_SECRET=<your-secret-key>

STRIPE\_API\_KEY=<your-stripe-api-key>

PAYPAL\_CLIENT\_ID=<your-paypal-client-id>

For the frontend, configure API base URLs and third-party keys as needed.

**Step 4:** Start the Database

If using a local MongoDB installation, ensure the MongoDB service is running:

mongod

If using MongoDB Atlas, ensure your IP address is whitelisted in the Atlas settings.

**Step 5:** Start the Backend Server

Navigate to the backend directory:

cd backend

Start the backend server:

npm run dev

The backend server will run on http://localhost:5000.

**Step 6:** Start the Frontend Application

**Navigate to the frontend directory:**

cd ../frontend

**Start the frontend application:**

npm start

The frontend will run on http://localhost:3000.

Testing the Application

Open a browser and navigate to http://localhost:3000 to access the frontend.

Verify backend API responses using Postman or by checking logs in the terminal.

Register a test user, browse products, and perform checkout to validate the workflow.

Troubleshooting

If dependencies fail to install:

**Clear npm cache:**

npm cache clean --force

**Reinstall dependencies:**

npm install

**If MongoDB doesn't connect:**

Verify the MONGO\_URI in the .env file.

Check if MongoDB service is running.

This completes the setup process. You now have a fully functional eCommerce application running locally!

**Folder Structure**

The project folder structure is organized to ensure a clean separation of concerns, making the application scalable and maintainable. The project is divided into two main directories: Client and Server.

* **Client Folder Structure**

The Client folder contains all the code related to the frontend, built with React.js. It focuses on the user interface, client-side routing, and interactions.

client/

├── public/

│ ├── index.html # Main HTML file for React

│ ├── favicon.ico # Website favicon

│ └── manifest.json # Metadata for Progressive Web App (optional)

├── src/

│ ├── components/ # Reusable UI components (e.g., Header, Footer, ProductCard)

│ │ ├── Header.js

│ │ ├── Footer.js

│ │ └── ProductCard.js

│ ├── pages/ # Page-level components for routing

│ │ ├── HomePage.js

│ │ ├── ProductPage.js

│ │ ├── CartPage.js

│ │ └── CheckoutPage.js

│ ├── redux/ # (If using Redux for state management)

│ │ ├── actions/ # Redux actions

│ │ ├── reducers/ # Redux reducers

│ │ ├── store.js # Redux store configuration

│ ├── context/ # (If using Context API instead of Redux)

│ │ └── AppContext.js # Context and provider for global state

│ ├── App.js # Root React component

│ ├── index.js # ReactDOM entry point

│ ├── routes/ # Routes configuration using React Router

│ │ └── Routes.js

│ ├── assets/ # Static assets like images and fonts

│ │ ├── images/

│ │ └── styles/

│ └── utils/ # Utility functions and helpers

│ └── api.js # Axios configurations and API calls

├── .env # Environment variables for client

└── package.json # Dependencies and scripts for client

**2. Server Folder Structure**

The Server folder contains all the backend code, built with Node.js and Express.js. It focuses on API creation, database interaction, and business logic.

server/

├── config/

│ ├── db.js # MongoDB connection setup

│ ├── jwt.js # JWT token handling

│ ├── keys.js # API keys and sensitive configurations

├── controllers/

│ ├── authController.js # Handles authentication logic (login, register)

│ ├── productController.js # CRUD operations for products

│ ├── cartController.js # Handles cart-related logic

│ ├── orderController.js # Order processing logic

│ └── userController.js # User profile and settings management

├── middleware/

│ ├── authMiddleware.js # Middleware for route protection

│ ├── errorHandler.js # Centralized error handling middleware

│ └── validation.js # Request validation logic

├── models/

│ ├── User.js # User schema and model

│ ├── Product.js # Product schema and model

│ ├── Order.js # Order schema and model

│ └── Cart.js # Cart schema and model

├── routes/

│ ├── authRoutes.js # Routes for authentication

│ ├── productRoutes.js # Routes for product management

│ ├── cartRoutes.js # Routes for cart operations

│ └── orderRoutes.js # Routes for orders

├── utils/

│ ├── email.js # Email notification utility

│ ├── payment.js # Payment gateway integration

│ └── logger.js # Logging utility

├── app.js # Main Express app configuration

├── server.js # Server entry point

├── .env # Environment variables for server

└── package.json # Dependencies and scripts for

**Server:**

* **Key Highlights:**

**Modular Design:**

Components, pages, routes, and utilities in the Client are grouped logically for clarity.

Controllers, models, and routes in the Server ensure clear separation of business logic.

**Environment Variables:**

.env files in both Client and Server store sensitive configurations like API keys, database URIs, and JWT secrets.

Scalability:

Adding new features is straightforward with well-organized folders.

Each folder/module can be expanded independently without affecting the overall structure.

**Running the Application**

To run the eCommerce application locally, both the frontend and backend servers need to be started. The frontend handles the user interface, while the backend manages the server-side logic, APIs, and database interactions.

Below are detailed instructions and commands for starting both the frontend and backend servers**.**

**1. Frontend: Starting the Client Server**

The frontend is developed using React.js, and the development server is powered by Webpack (via Create React App). This server provides a live-reloading experience for development.

Steps to Start the Frontend Server

Navigate to the client directory in your terminal:

**cd client**

**Install all dependencies (if not already installed):**

**npm install**

**Start the development server**

**npm start**

**Access the application:**

The frontend server runs by default at http://localhost:3000.

Open this URL in a web browser to interact with the application.

Expected Behavior

The application will open in the default web browser.

Any changes to the React components or styles will trigger hot-reloading, reflecting updates immediately**.**

**2. Backend: Starting the Server**

The backend is built using Node.js and Express.js, and it handles APIs, authentication, database operations, and other server-side functionalities.

Steps to Start the Backend Server

Navigate to the server directory in your terminal:

cd server

Install all dependencies (if not already installed):

npm install

Start the development server:

npm start

Access the backend server:

The backend server runs by default at http://localhost:5000.

Use tools like Postman or curl to test API endpoints.

Expected Behavior

The server will output logs indicating successful startup and connection to the database.

API endpoints will be accessible at http://localhost:5000/api.

**3. Simultaneously Running Frontend and Backend**

To run both servers simultaneously:

Open two terminal windows or tabs.

In one terminal, navigate to the client directory and start the frontend:

cd client

npm start

In the second terminal, navigate to the server directory and start the backend:

cd server

npm start

**4. Development Environment**

In development, the frontend and backend run on separate ports:

Frontend: http://localhost:3000

Backend: http://localhost:5000

To ensure seamless communication between the frontend and backend:

The frontend proxies API requests to the backend. This is typically configured in the package.json file of the client:

**json**

"proxy": "http://localhost:5000"

Common Issues and Troubleshooting

**Port Already in Use:**

If localhost:3000 or localhost:5000 is occupied, change the port or stop the conflicting service.

**For the frontend:**

PORT=3001 npm start

For the backend: Modify the PORT value in the .env file.

Dependency Issues:

Clear the node\_modules folder and reinstall dependencies:

rm -rf node\_modules

npm install

**Database Connection Failure:**

Ensure MongoDB is running locally or check the connection string in the .env file.

Summary of Commands

**Frontend:**

cd client

npm start

Runs the React development server at http://localhost:3000.

**Backend:**

cd server

npm start

Runs the Node.js server at http://localhost:5000.

Both servers working in tandem enable a fully functional local development environment for the eCommerce application.

**API Documentation**

The API is the backbone of the eCommerce website, facilitating communication between the frontend and backend. It is designed using RESTful principles, ensuring a modular, stateless, and scalable architecture. Below is an overview of key API endpoints, their purposes, and usage instructions.

Base URL

For a local setup:

http://localhost:5000/api

For a production environment:

arduino

https://your-production-domain/api

**Authentication APIs**

Endpoints related to user authentication, such as registration, login, and profile management.

Endpoint Method Description Headers Body

/auth/register POST Register a new user None { "name": "John Doe", "email": "john@example.com", "password": "123456" }

**Authentication:**

Authentication is a critical part of the eCommerce application, ensuring secure access to user accounts and protecting sensitive data. This project implements JWT (JSON Web Token) for stateless authentication and optionally supports Passport.js for session-based authentication

**Key Features**

User Registration:

Allows new users

to create an account with their name, email, and password.

Passwords are hashed using a secure algorithm (e.g., bcrypt) before storage in the database.

User Login:

Authenticates users based on their email and password.

On successful login, a JWT token is issued, containing user information (e.g., user ID, role).

Token-Based Authentication:

The backend uses JWT to validate user sessions without maintaining server-side session storage.

Tokens are signed with a secret key and include an expiration time for added security.

Authorization:

Protects specific routes (e.g., /cart, /orders) by verifying the JWT token.

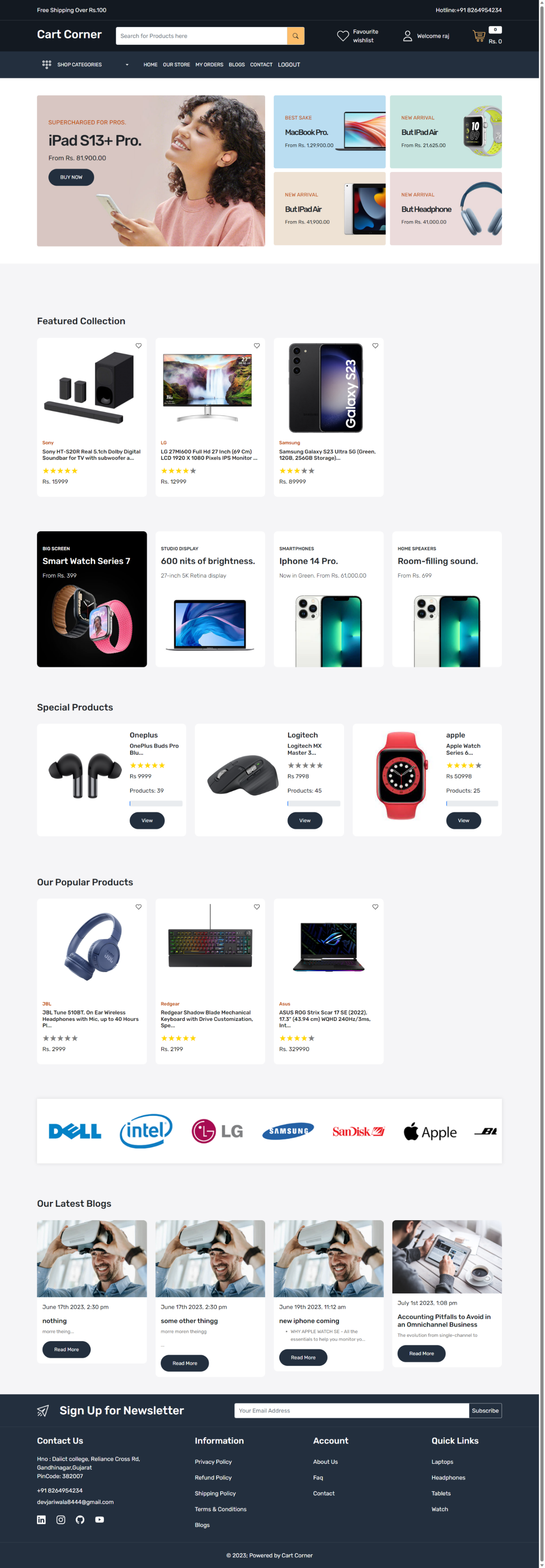
Role-based access control (RBAC) is implemented for distinguishing between regular users and admins.

**User Interface:**

**Product Catalog:**

The website should display a comprehensive catalog of products with details such as name, description, price, and images.

Products should be categorized and easily searchable to facilitate user navigation.

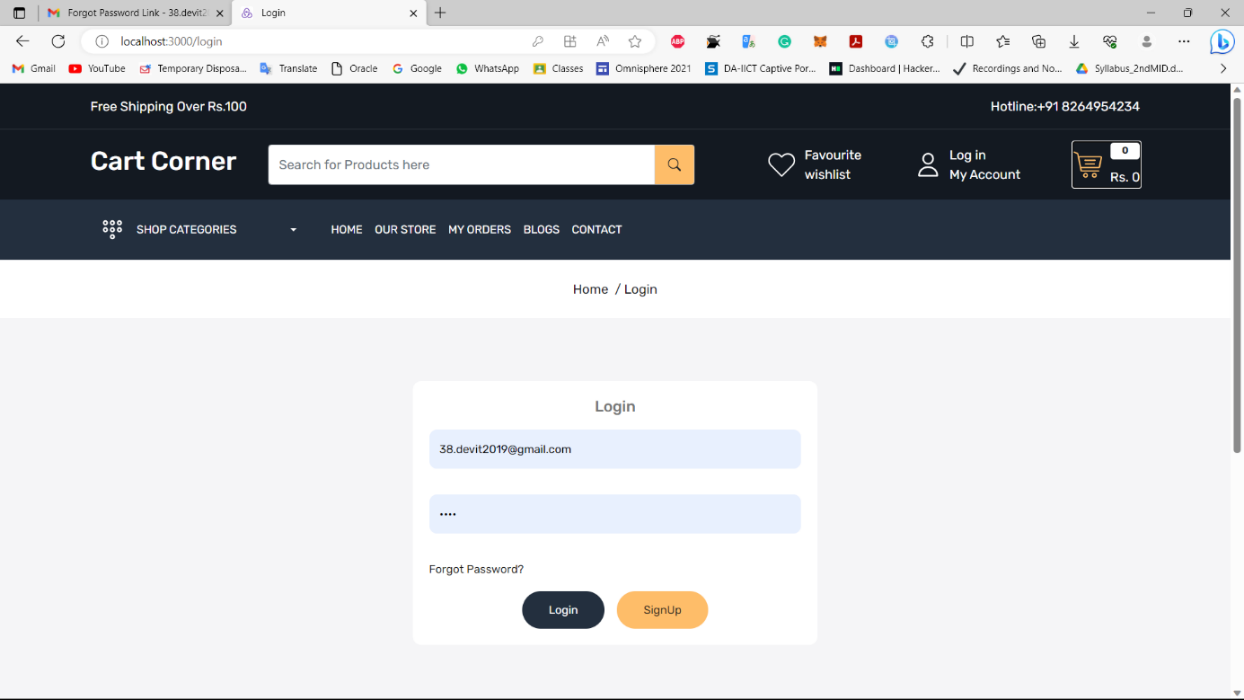


**User Registration and Authentication:**

Users should be able to register on the website with a unique username and password.

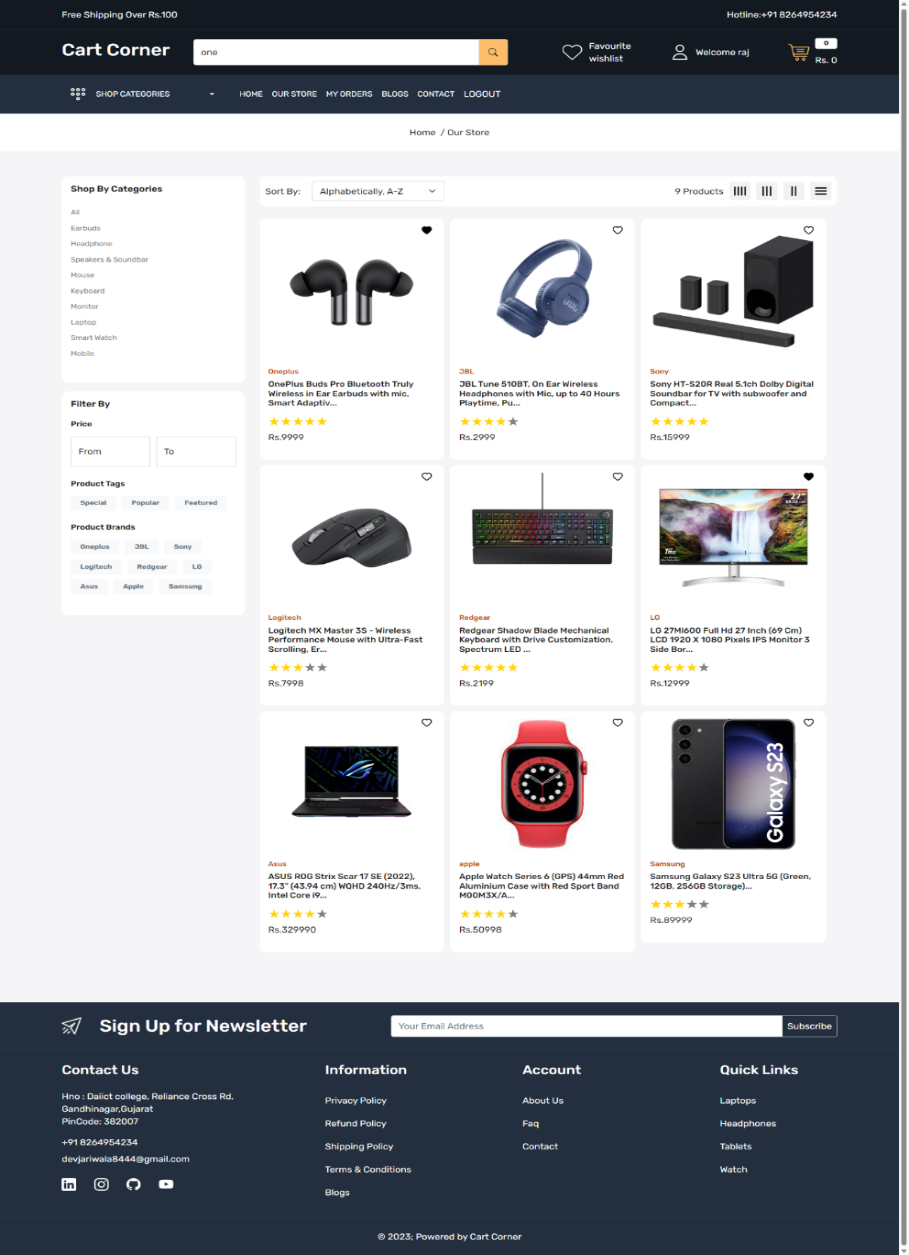
Registered users should be able to log in and access their personalized accounts.

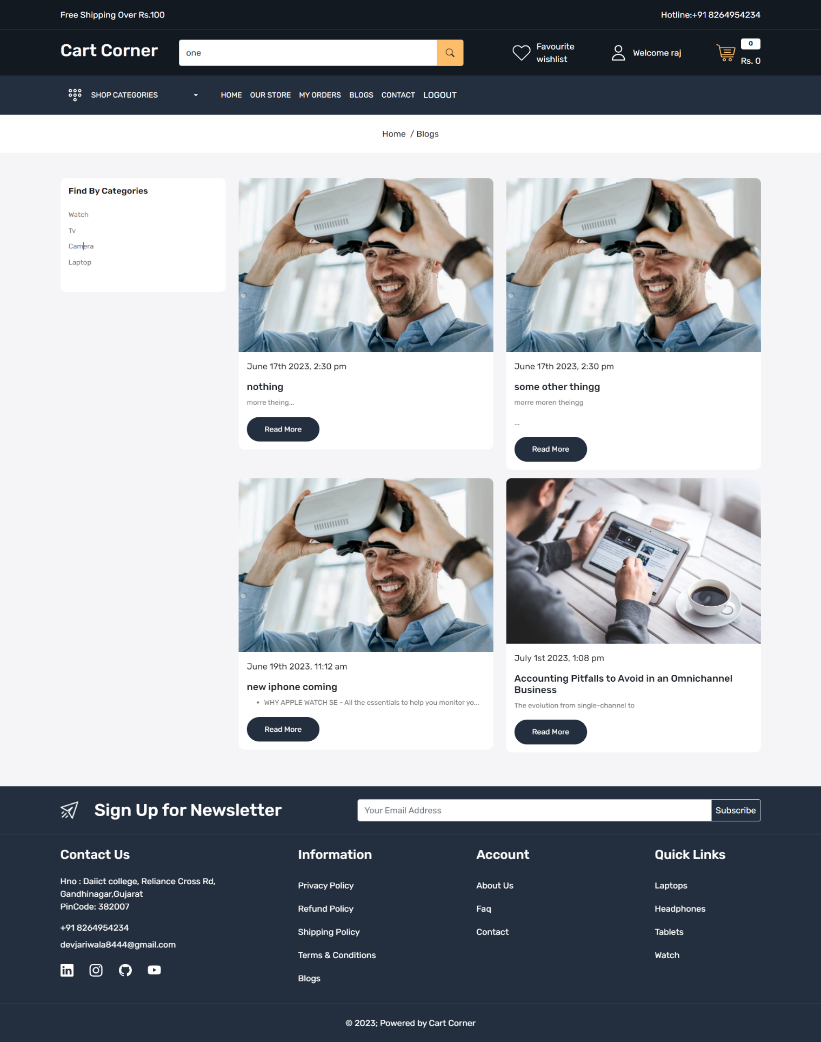




**Product Catalog:**

* + The website should display a comprehensive catalog of products with details such as name, description, price, and images.
  + Products should be categorized and easily searchable to facilitate user navigation.





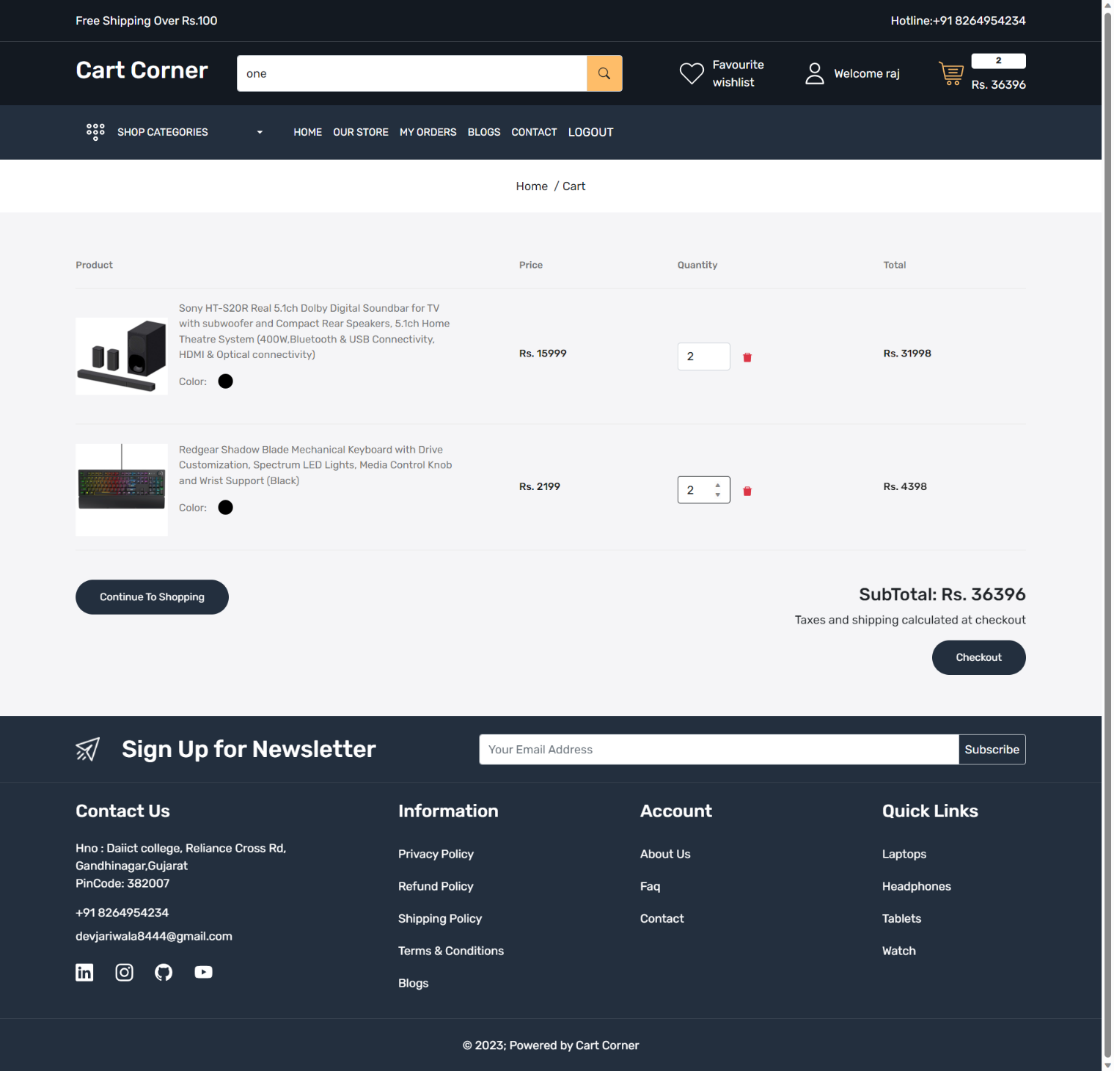
**Customer Reviews and Ratings:**

* + Users should be able to leave reviews and ratings for products they have purchased.
  + Average product ratings should be displayed to assist other users in making purchase decisions.



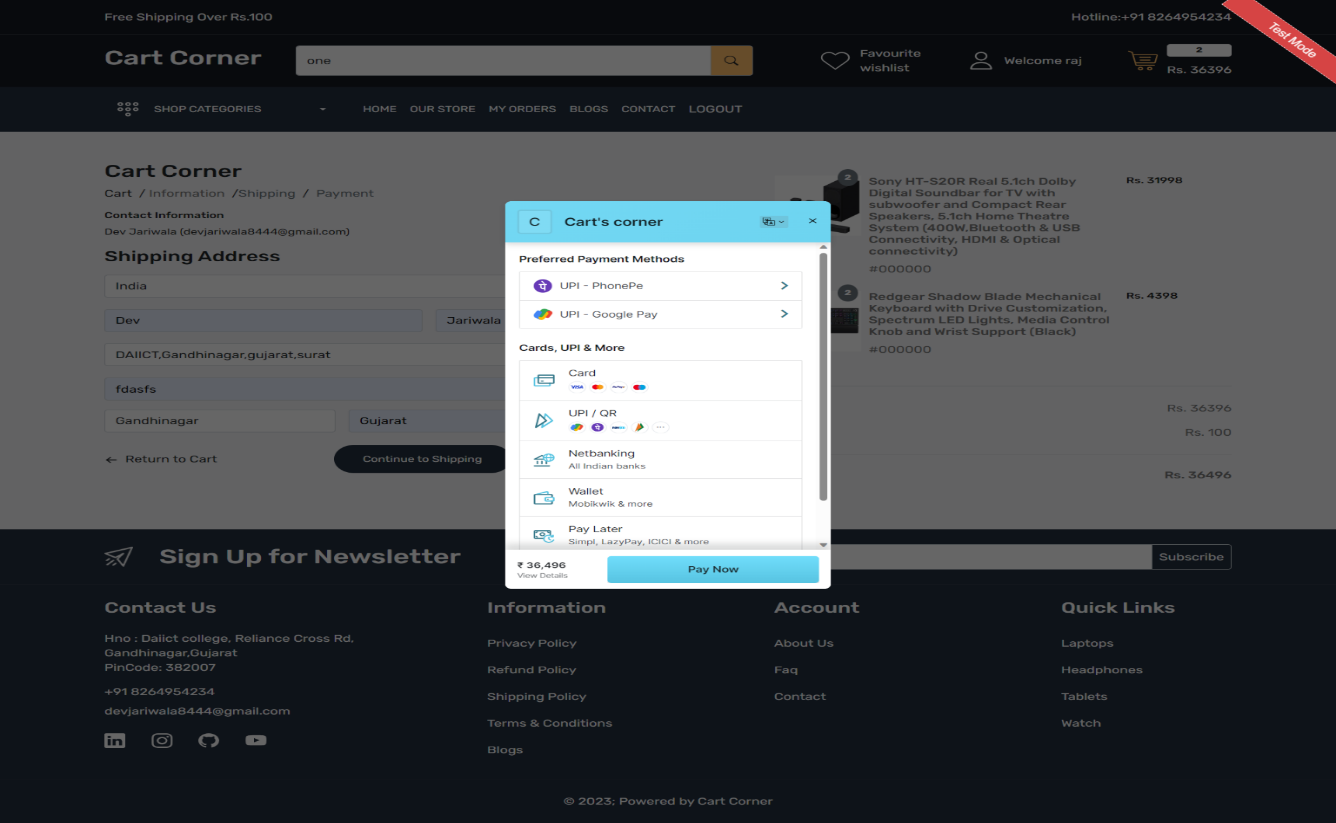
**Shopping Cart and Checkout:**

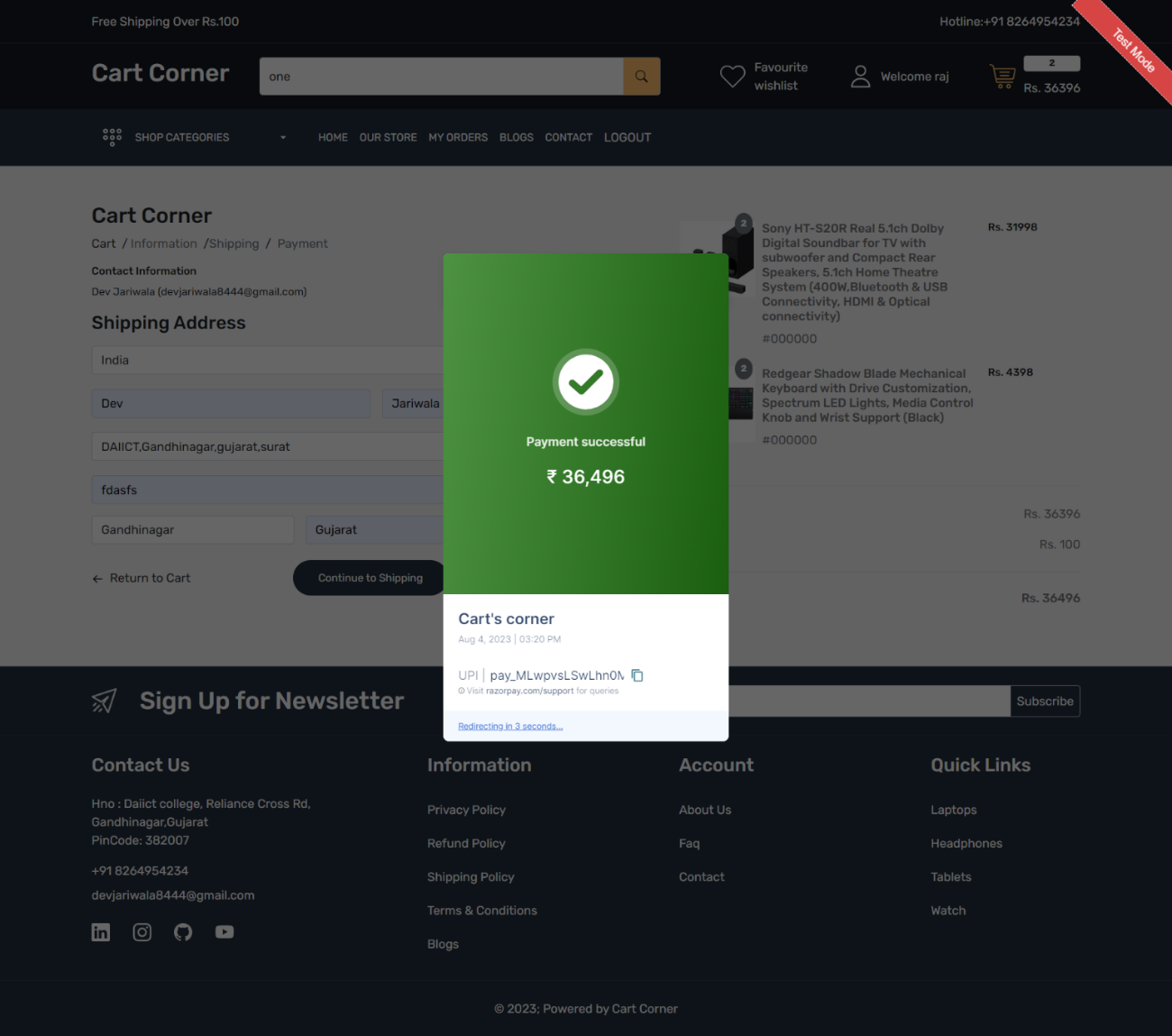
* + Users should be able to add products to their shopping cart and review cart contents before proceeding to checkout.
  + During the checkout process, users should provide shipping and billing information and choose a payment method.



Payment Gateway Integration:

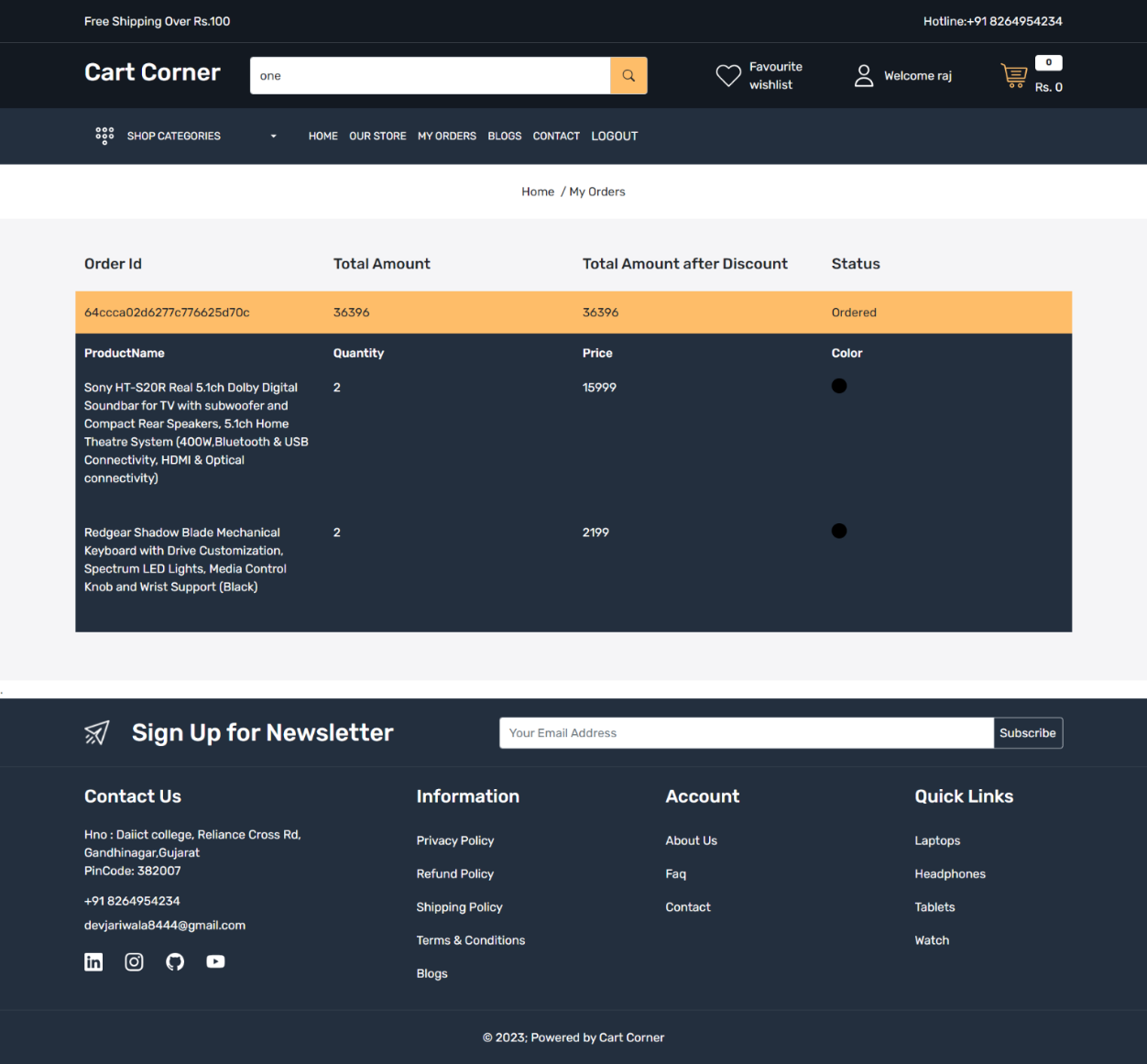
* + The website should integrate with a secure payment gateway to process online transactions.
  + Users should have multiple payment options, such as credit/debit cards, PayPal, or other relevant payment methods.





**Order Confirmation and Email Notifications:**

* + Users should receive an order confirmation email after completing a purchase.
  + Email notifications should be sent for order updates, such as shipping and delivery status.. 2.2 Non-Functional Requirements



**Testing**

**1. Testing Strategy**

Unit Testing: Verifies individual components and functions (e.g., API endpoints, React components).

Integration Testing: Ensures different modules (e.g., frontend and backend) work seamlessly together.

End-to-End Testing: Tests the complete user journey, from browsing products to placing orders.

Regression Testing: Ensures new changes do not break existing functionality**.**

**2. Tools Used**

**Frontend Testing:**

Jest: For unit and integration tests of React components.

React Testing Library: Simulates user interactions and verifies UI behavior.

**Backend Testing:**

Mocha & Chai: For testing API endpoints and server-side logic.

Postman: Manual API testing and debugging**.**

**End-to-End Testing:**

**C**ypress: Automates user flows and verifies the overall system functionality.

**Product Recommendations:**

* + **The website should provide personalized product recommendations based on user browsing and purchase history.**
  + **Recommended products should appear on the homepage or product pages.**

**Wishlist and Favorites:**

* + **Users should have the option to add products to their wishlist or favorites list for future reference.**
  + **Wishlist items should be easily managed and shareable.**

**Inventory Management:**

* + The website should track product inventory and display availability to users.
  + Users should be notified when a product is out of stock or low in quantity.

**Order Confirmation and Email Notifications:**

* + Users should receive an order confirmation email after completing a purchase.
  + Email notifications should be sent for order updates, such as shipping and delivery status.. 2.2 Non-Functional Requirements

**Performance:**

* + The website should load quickly, with minimal page load times, to enhance user experience.
  + It should be able to handle a large number of concurrent users without significant performance degradation.

**Security:**

* + User data, including personal information and payment details, should be encrypted and stored securely.
  + The website should be protected against common security threats, such as SQL injection and cross-site scripting (XSS) attacks**.**

**Scalability:**

* + The system should be designed to accommodate future growth and increased traffic.
  + It should be scalable to handle a larger product catalog and user base without compromising performance.

**Accessibility:**

* + The website should be accessible to users with disabilities, following WCAG (Web Content Accessibility Guidelines) standards.
  + It should support screen readers, keyboard navigation, and other accessibility features.

**Reliability and Availability:**

* + The website should have high uptime and be available to users at all times, with minimal downtime for maintenance.

**User-Friendly Interface:**

* + The website's user interface should be intuitive and easy to navigate, ensuring a positive user experience**.**

**Compliance:**

* + The website should adhere to relevant industry standards, data protection regulations (e.g., GDPR), and legal requirements.

**Backup and Recovery:**

* + Regular data backups should be performed to prevent data loss in case of system failures.
  + A robust data recovery mechanism should be in place to restore the website in the event of any unforeseen issues.

**Performance Monitoring and Analytics:**

* + The website should have performance monitoring tools to track user behavior, identify bottlenecks, and gather insights for continuous improvement.



**Screenshots or Demo**

**Demo Vedio Link:** <https://drive.google.com/drive/folders/1-6zaKWCSqjY2Tix5kMrbdHb0bz0KBWBv>

**Known Issues**

**Session Timeout on Frontend**

**Issue:** Users may need to manually refresh the page after token expiration to log in again.

**Temporary Fix:** Implement token refresh or logout logic**.**

**Slow Product Search:**

**Issue:** Product search may be slow for large datasets.

**Temporary Fix:** Optimize database queries or implement indexing in MongoDB.

**Cart Quantity Sync:**

**Issue:** Cart quantities may not update immediately after backend API call.

**Temporary Fix**: Refresh the page or add real-time updates using WebSocket.

**13. Future Enhancements**

**1. Enhanced Product Search**

**Objective:** Improve the product discovery experience for users.

**Planned Features:**

**Advanced Filters:** Allow users to filter search results based on:

Price range

Categories

Ratings

Availability (in stock/out of stock)

**Auto-Suggestions**: Display suggested products or categories as users type in the search bar.

**Spell-Check Functionality:** Automatically correct or suggest alternatives for misspelled search terms to ensure accurate results.

**2. Real-Time Updates**

**Objective**: Improve the user experience with real-time data synchronization.

**Planned Features:**

**WebSocket Integration**: Use WebSocket or similar protocols to enable real-time updates for:

Cart changes (e.g., quantity updates, item additions or removals).

Order status updates (e.g., confirmed, shipped, delivered).

**Notification System**: Push notifications for key actions like order confirmation or stock replenishment.