SHOP.IT – E-Commerce Web Application (React)

Introduction

The Modern E-Commerce Application is a React-based web platform designed to bring a complete online shopping experience to users in a smooth and efficient way. It combines a clean and visually appealing interface with practical features that make browsing, selecting, and purchasing products simple and enjoyable. Built using React.js, JSON-Server, and Tailwind CSS, the application demonstrates how modern web tools can work together to create a fully functional, responsive online store.

In most online shopping scenarios, product information, inventory updates, and purchase history are often scattered or hard to track. This application centralizes all this information, giving users a single place to view products, add items to the cart or wishlist, and complete purchases quickly. With real-time updates, toast notifications, and mobile-friendly design, the app ensures users can shop efficiently and confidently.

The application is not only functional but also educational, showing how React components, state management, and API integration work together to build a complete e-commerce solution. It is suitable for learning, prototyping, and real-world small-scale online stores.

Scenario-Based Intro

To understand how the application works in real life, let's follow a typical user journey. Imagine Sarah, a 28-year-old freelance graphic designer, with a busy schedule and little time to spare.

One afternoon, Sarah needs to buy a secondary monitor for work and a waterproof jacket for an upcoming hiking trip. She opens the e-commerce application on her laptop and is immediately greeted by a fast-loading, well-organized homepage. The hero section clearly highlights popular categories and a "Shop Now" button, guiding her to the products she needs.

Sarah first navigates to the Electronics section. Using the filters, she quickly finds the Samsung 49-inch QLED monitor. Clicking on it, she sees high-quality images, specifications, and descriptions. She adds it to the cart, and a small notification confirms her action without interrupting her shopping flow.

Next, she switches to the Women's Clothing category. The page updates instantly without any reload, and she finds a stylish rain jacket. She adds it to her cart and also saves a gold bracelet she likes in her wishlist for future purchase

Finally, she reviews her cart, checks the subtotal, and proceeds to checkout. In less than six minutes, Sarah has purchased the items she needs and saved a future product—all without stress or delay. This journey highlights the platform's speed, ease of use, and user-centered design.

Target Audience

The Modern E-Commerce Application is designed for a variety of users, each with unique needs and expectations. Professionals like Sarah, who have limited time and high expectations for speed, benefit from the smooth navigation and quick-add features that reduce friction during shopping. Tech-savvy users, such as Millennials and Gen Z, who are accustomed to mobile apps with responsive, fluid interfaces, will appreciate the platform's mobile-first design and adaptive layouts. Casual shoppers, who enjoy browsing and saving items for future purchases, are supported by the wishlist feature, allowing them to engage with the platform without immediate commitment. Additionally, students and developers can use the project as an educational tool to study real-world React architecture, component design, and state management techniques, making the application both practical and instructive.

Project Goals and Objectives

The primary goal of this project is to create an online shopping platform that is fast, user-friendly, and visually appealing. It aims to combine technical excellence with practical functionality, ensuring that users can browse products, manage their cart, and complete purchases quickly and efficiently. One of the main objectives is to implement a Single Page Application (SPA) structure, where page content updates dynamically without full reloads, providing a native app-like experience. Another important objective is to handle data dynamically through a backend API using JSON-Server, allowing product information to be updated or added without affecting the frontend.

Efficient state management is also a key goal. The application ensures that the cart, wishlist, and totals are consistently updated across different pages, so users always have accurate information. Additionally, the platform aims to minimize user effort and cognitive load by providing clear layouts, readable typography, and intuitive navigation. Finally, the project serves an educational purpose by demonstrating how React, state management, API communication, and responsive design can be combined to create a complete e-commerce solution.

Key Features

1. Dashboard Overview

The dashboard of the E-Commerce Application acts as the user's starting point, giving an instant overview of the shopping activity happening on the platform. Instead of overwhelming the user with too much information, the dashboard presents a simple and clear summary, such as the available product categories, featured items, and any active offers or deals. It creates a smooth entry into the shopping flow by allowing users to glance through essential updates and navigate quickly to different sections of the application. This helps users understand where they can go next—whether it is exploring products, checking their cart, or viewing their wishlist—without confusion.

2. Products Management

The application includes a complete product management flow that allows users to explore, view, and interact with a wide variety of items. All product details such as name, price, description, ratings, and images are presented in a visually clean format. Users can browse by category, open detailed product pages, and decide whether to buy or save a product for later. Behind the scenes, the system stores all product data and ensures a smooth experience by loading only the necessary information on each page. The goal is to create an organized shopping environment where every item can be viewed in a structured and user-friendly manner.

3. Projects Management

Cart management is one of the key features of the application, where users can add products they intend to purchase. The cart updates instantly the moment a user adds, removes, or changes quantities of items. A clean summary of item prices, total cost, and quantity is always displayed so that the user is aware of their spending. The application ensures that this process feels natural—like placing items in a basket in a real store. Each change is reflected immediately using React state updates, making the cart an interactive and essential part of the shopping workflow.

4. Wishlist Management

The wishlist section helps users save products they like but don't want to purchase immediately. This feature provides an effortless way to mark items for future consideration without losing track of them. The system maintains the wishlist even when the user moves across different pages, offering a more personalized shopping experience. Many users enjoy exploring items before making a decision, and this feature supports that behaviour by keeping their favourite products organized in one place.

5. Checkout & Order Summary

When the user is ready to complete the purchase, the checkout page presents a well-structured summary of all the items in the cart, along with pricing details, taxes, total

amount, and delivery information. The purpose of this page is to ensure the user feels confident before confirming their order. The design is made simple and clean so the user can easily verify everything at a glance. After placing the order, the system stores the data and ensures the user receives confirmation immediately.

6. Payments Management

The Payments section is designed to make it easy to record any payment made toward a specific invoice. Whenever a customer completes a payment, the details can be entered into the system so that all financial records stay accurate and updated. The application supports multiple payment methods, including bank transfers, card payments, cash, PayPal, and several others, allowing users to select whichever mode the customer used. Once the payment information is saved, the system automatically compares the amount received with the total invoice value. If the full amount has been paid, the corresponding invoice status is instantly updated to "Paid," ensuring that the user does not have to manually track or modify the status. This automatic update helps maintain clarity and reduces the chances of errors in payment tracking.

7. Form Validation Notifications

Throughout the application, form validation plays an important role in maintaining smooth user interaction. Whenever the user performs actions like adding an item to the cart or wishlist, attempting checkout, or navigating through sections, instant toast notifications appear to confirm the result of their action. This makes the application feel alive and responsive, helping users understand that their inputs have been processed successfully. Any incorrect or incomplete entry is flagged with a clear message, ensuring error-free interactions.

8. Responsive Design and Layout

The entire application is built with a fully responsive layout powered by Tailwind CSS. Every page—from the homepage to the product details and cart—is optimized to look clean on all screen sizes. Whether the user shops on a phone, tablet, or laptop, the UI adjusts automatically to offer the same smooth experience. Responsive grids, flexible images, and mobile-friendly navigation ensure that the application remains usable and visually appealing regardless of the device being used.

PRE-REQUISITES

Here are the key prerequisites for developing and running the ClientSync – CRM Web Application built with React, Vite, Tailwind CSS, and JSON-Server.

1. Node.js and npm

Node.js is a JavaScript runtime environment that allows you to run JavaScript outside the browser, on your local machine. It is required to install dependencies and run both the React frontend and the JSON-Server backend.

- Install Node.js (which includes npm, the Node package manager) on your system.
- Node.js and npm are used to:
  - Install project dependencies (npm install)
  - Run development servers (npm run dev, npm run json-server)
  - Build the production version of the app (npm run build)

Download:

- Node.js: https://nodejs.org/en/download/
- Installation via package manager: https://nodejs.org/en/download/package-manager/

2. React, Vite, and Project Setup

SHOP.IT is built using React 18 with Vite as the build tool and dev server. To create a similar React + Vite project:

1. Create a new Vite React app:

   npm create vite@latest

2. Enter a project name (for example, SHOP.IT) and choose the React framework (and JavaScript).
3. Navigate to the project directory and install dependencies:

cd application

npm install

4. SHOP.IT uses two servers during development:

- JSON-Server (mock REST API)
- Vite dev server (React frontend)

In two terminals:

npm run json-server

npm run dev

Then open the app in your browser at:

- Frontend: http://localhost:5173
- Backend API (JSON-Server): http://localhost:4000

3. JSON-Server

JSON-Server acts as a simple backend for our E-Commerce project. Instead of creating a full backend from scratch, JSON-Server reads the data from a file called db.json and automatically creates API routes. These routes allow our frontend to get product details, manage the shopping cart, and store orders. For example, when the React app

requests /products, JSON-Server returns a list of all products. If the user adds something to the cart, a request is sent to /cart and JSON-Server stores it. This makes it easy to test features like "Add to Cart", "Update Quantity", and "Place Order" without needing a real database. JSON-Server supports all basic operations such as creating, reading, updating, and deleting data, making development faster and simpler.

4. Tailwind CSS

Tailwind CSS is used to design the user interface of the E-Commerce Application. It is a utility-first CSS framework, which means it provides small classes that can be directly applied to HTML elements to design the UI quickly. Instead of writing long CSS files manually, we can simply use Tailwind classes to control spacing, colors, fonts, borders, grids, and layouts. Tailwind is configured in files like tailwind.config.js and postcss.config.js, and its styles are included in index.css. Using Tailwind helps the project look modern, clean, and responsive across all screen sizes. It is used to style product grids, navigation bars, buttons, forms, and all shopping pages, making the overall UI consistent and professional.

5. HTML, CSS, and JavaScript Fundamentals

To work smoothly with this project, a basic understanding of HTML, CSS, and JavaScript is helpful. HTML structures the content on each page, such as displaying product cards or placing buttons. CSS is responsible for the appearance of the layout, and in this project Tailwind CSS is used to make styling much easier. JavaScript (especially ES6 features) adds interactivity and logic to the website. It is used to manage state, handle button clicks, make API requests, filter products, and update the shopping cart. Since React is built on JavaScript, knowing the basics makes it easier to understand components, hooks, and event handling inside the E-Commerce application.

6. Version Control (Git)

Using Git for version control is recommended to track changes, manage versions, and collaborate on the ClientSync project.

- Initialize a Git repository in the project directory.
- Push your project to platforms like GitHub, GitLab, or Bitbucket for backup and collaboration.

Git Download:

- https://git-scm.com/downloads

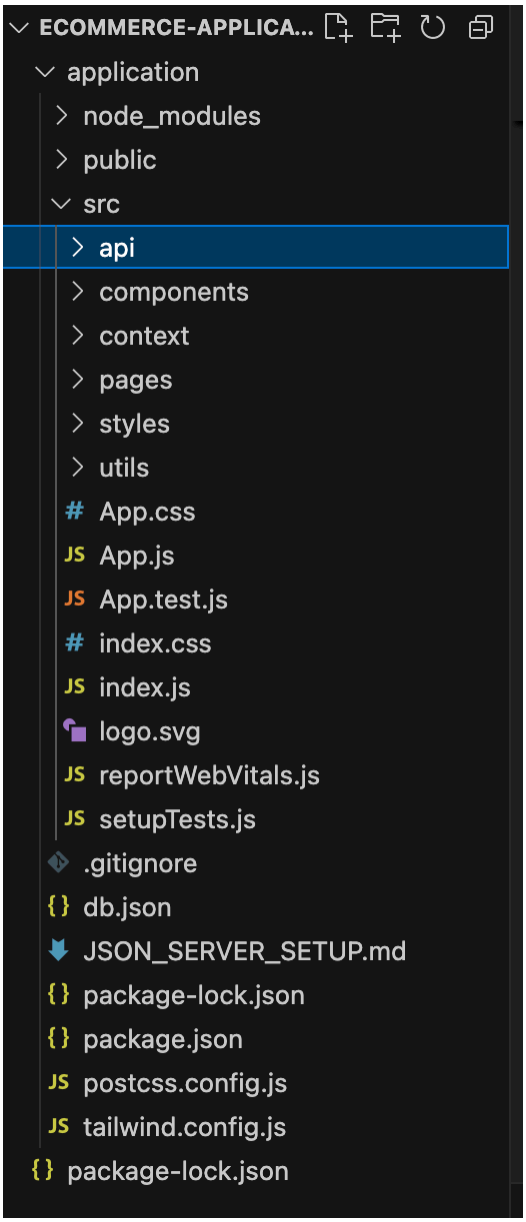
7. Development Environment (Code Editor)

Visual Studio Code

◦ Lightweight, popular editor with great React/JS tooling.

◦ Download: https://code.visualstudio.com/download

Project structure



Root files like db.json, vite.config.js, tailwind.config.js, and postcss.config.js define the mock database and tooling configuration, while package.json manages all dependencies and npm scripts for running and building the ClientSync application.

The structure of the project is arranged in a clean and organized way so that all important parts of the application are separated clearly. This makes the entire codebase easier to understand, maintain, and expand when new features are added. The main files—src/index.js, src/App.js, and src/index.css—form the core foundation of the React application. The index.js file acts as the entry point, where the React app is mounted onto the main index.html. The App.js file works as the root component of the application and contains the routing structure that connects all the

main pages such as Home, Products, Cart, Login, and other screens. The index.css file stores global styles along with the Tailwind CSS directives used throughout the project.

Inside the src folder, different subfolders are used to organize code based on functionality. The components folder contains reusable UI elements—such as navigation bars, product cards, buttons, and loaders—that appear in multiple parts of the application. The pages folder includes all main screens of the project like Homepage, Product Listing, Product Details, Cart, Wishlist, Signup, and Login pages. The api folder contains files that handle all API communication between the frontend and the JSON-Server backend, making it easier to manage HTTP requests from a single place. The context folder is used to store React Context files, which help manage global states such as authentication state or cart data. The styles folder contains custom CSS and Tailwind-based styling files, while the utils folder stores helper functions that are shared across the app, such as currency formatting or form validation logic.

At the root of the project, files like db.json, tailwind.config.js, postcss.config.js, and package.json play an important role in the setup. The db.json file acts as the mock database for JSON-Server, while the Tailwind and PostCSS configuration files handle styling. The package.json file lists all project dependencies and the scripts needed to run or build the application. Together, this structure ensures that every part of the project has a clear place and the workflow remains smooth and scalable.

PROJECT FLOW

Before starting to work on this project, let's see the demo.

GitHub link:

https://github.com/Varshiniamara/ecommerce-website

Milestone 1: Project Setup and Configuration:

1. Install required tools and software:

Installation of required tools:

Open the project folder to install the necessary tools for SHOP.IT Web Application.

In this project, we use:

- React.js (with Vite)
- React Router DOM
- Tailwind CSS
- Axios

- JSON-Server
- react-hook-form
- react-toastify
- react-icons
- jsPDF
- html2canvas

For further reference, use the following resources:

- React installation: https://react.dev/learn/installation
- Vite + React guide: https://vitejs.dev/guide/
- Tailwind CSS installation: https://tailwindcss.com/docs/installation
- JSON-Server guide: https://github.com/typicode/json-server
- React Router docs: https://reactrouter.com/en/main/start/overview

Milestone 2: Web Development

1. Setup React Application

Client-side routing in SHOP.IT is handled using React Router DOM, defined inside src/App.jsx. The application uses BrowserRouter (imported as Router) to wrap the entire project, enabling seamless navigation without reloading the page.

Inside the Router, multiple routes are declared using Routes and Route, including:

- / – Homepage
- /signin – SignIn page
- /signup – SignUp page
- /dashboard – Dashboard (protected)
- /products – Products page (protected)
- /cart – Cart page (protected)
- /wishlist – Wishlist page (protected)
- /orders – Orders page (protected)
- /profile – Profile page (protected)
- * – redirects all unknown paths back to /

To secure important sections like Dashboard, Products, Cart, Wishlist, and Orders, the project uses a ProtectedRoute component. This component checks whether the user is

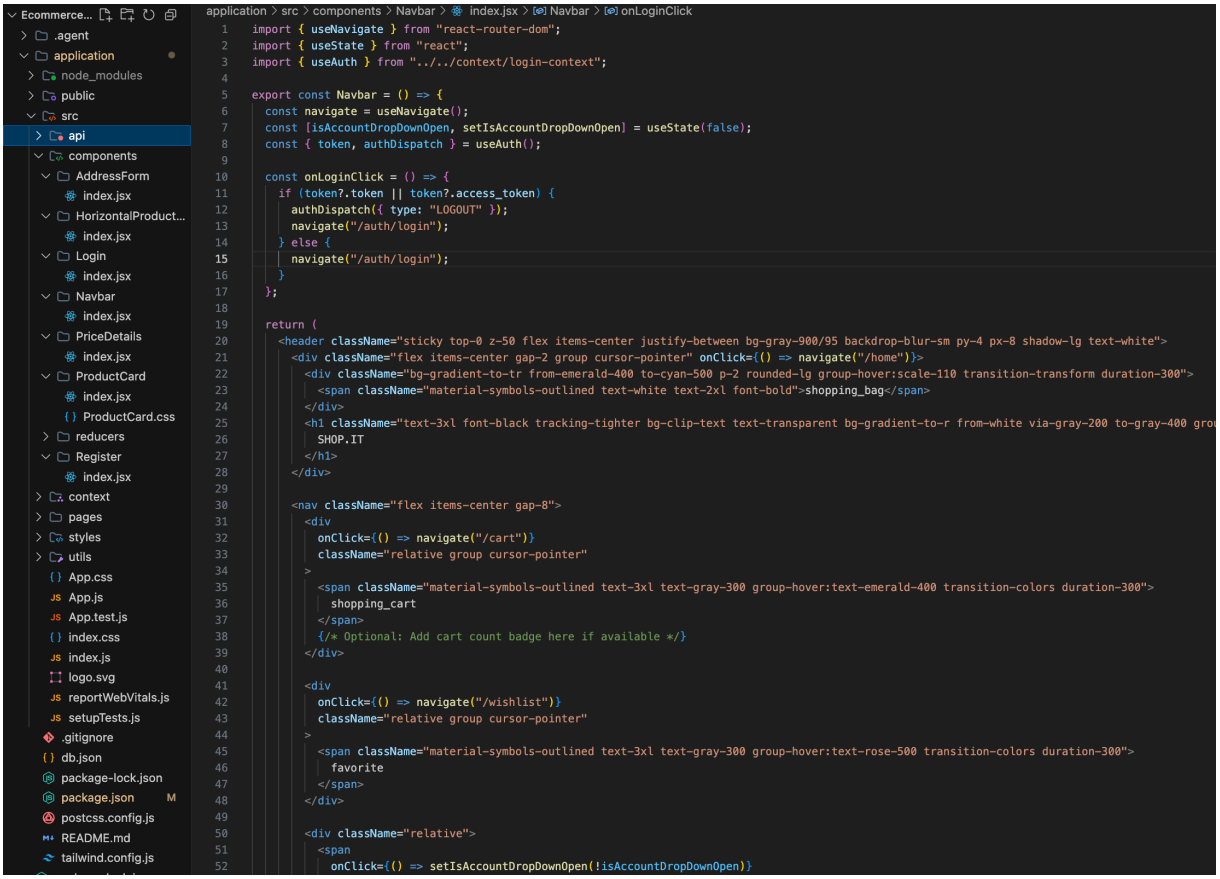
logged in; if not, it automatically redirects them to the SignIn page. This ensures only authenticated users can access shopping features.

Each protected route is wrapped with a Layout component, which provides the structure common to all internal pages—such as the sidebar, top navigation bar, and content area—ensuring a consistent user interface throughout the application.

The app uses:

The SHOP.IT application uses several important libraries to ensure smooth routing, API communication, form handling, notifications, styling, and PDF generation:

- react-router-dom is used for page routing and navigation across the app. It allows users to move between pages like Homepage, Products, Cart, Wishlist, Orders, and Profile without reloading the page.
- axios is used to make API calls to the JSON-Server backend. All CRUD operations for products, cart items, wishlist data, and orders are handled through Axios requests.
- react-hook-form helps manage all form inputs and validations, such as Sign Up, Sign In, Profile updates, and checkout forms. It ensures lightweight and efficient form handling.
- react-toastify is used for showing global success and error messages—like "Item added to cart," "Login successful," or "Product removed from wishlist." The ToastContainer is configured inside App.jsx so messages can appear from anywhere in the app.
- react-icons provides icons used in the entire UI—for buttons, labels, navigation links, cart badges, wishlist hearts, and more—making the interface cleaner and more visually appealing.
- tailwindcss and PostCSS provide styling across the whole application. With Tailwind's utility classes, pages like Products, Cart, Wishlist, Dashboard, and Orders are designed to be responsive, organized, and visually consistent.



Code Description – src/App.jsx component:

In the SHOP.IT web application, the main routing and navigation flow is handled using React Router DOM. At the top of the App component, BrowserRouter is imported and aliased as Router, along with Routes and Route, to define all navigable paths of the application. The app also imports the ToastContainer component from react-toastify, along with its stylesheet, allowing the entire application to display success and error notifications globally. In addition to these, the App component imports all page components such as Home, Products, Cart, Checkout, Orders, AdminPanel, Login, and Register, depending on what you have in the SHOP.IT app. A custom ProtectedRoute component is used to ensure that only authenticated users can access sensitive pages such as Cart, Checkout, Orders, and AdminPanel. Inside the App function, the JSX structure begins by wrapping the entire application inside , enabling client-side navigation. Each route is defined inside , where public pages like Home, Login, Register, and Products are accessible to all users, while protected pages are wrapped inside to check login status before rendering the actual page. If the user is not authenticated, they are automatically redirected to the Login page. The App component also integrates Tailwind CSS classes for layout styling and uses Axios for API calls to the JSON-Server backend that stores products, users, orders, and cart items. At the bottom, is included so that every page in the application can trigger toast messages for actions like "Item added to cart," "Order placed successfully," or "Login failed." Finally, the App component is exported as the default export and is mounted by main.jsx, making it the central entry point through which the entire SHOP.IT application operates.

2. Design UI Components

In the SHOP.IT web application, all reusable UI elements are organized inside the src/ components/ folder to keep the structure clean and modular. Components like Navbar, Footer, ProductCard, CartItem, and ProtectedRoute are created so they can be reused across multiple pages. The main Layout component is responsible for wrapping the core content of each page and includes common elements like the navbar at the top and the footer at the bottom, ensuring a consistent look throughout the app. The Navbar itself contains navigation links to important sections such as Home, Products, Cart, Orders, and Login/Account, making it easy for users to move between pages. The ProductCard component is used to display individual product details in grids on the Products page, while CartItem manages how items appear inside the shopping cart. The ProtectedRoute component checks whether a user is logged in before allowing access to private pages such as the Cart, Checkout, Orders, or Admin dashboard.

The entire UI design and styling are implemented using Tailwind CSS, which provides utility classes for layout management, spacing, colors, grid structures, and flexbox alignment. This ensures the SHOP.IT interface is clean, modern, and fully responsive. As a result, the layout automatically adjusts across devices ranging from large desktop screens to mobile phones, with features like collapsible menus, stacked product cards, and responsive tables where required. Navigation within the application is handled by React Router DOM, and all links inside the navbar, product lists, buttons, and admin controls directly use Router Link components to switch pages without reloading the browser. This allows users to smoothly move between Home, Products, Cart, Checkout, and Orders just like a real e-commerce platform. Overall, the combination of reusable components, Tailwind-based styling, and React Router navigation creates a polished, fast, and user-friendly shopping experience.

3. Implement Frontend Logic

In the SHOP.IT application, the connection between the frontend and the backend is handled using Axios, which is configured inside src/config/axiosConfig.js to communicate with the JSON-Server database stored in db.json. All feature pages—such as Products, Cart, Users, and Orders—send requests to endpoints like /products, /cart, /users, and /orders to perform complete CRUD operations. Whenever the user browses products, adds an item to the cart, places an order, or updates account information, these pages interact with the corresponding API routes to fetch and store data in real time.

Form handling and data binding across the application rely on React hooks such as useState and useEffect, along with react-hook-form, which simplifies managing input values. These tools ensure that whatever the user types into forms—whether it's login information, shipping details, or new product entries in the admin panel—gets properly captured and stored in the component state before being submitted to the server. Whenever API data changes, such as products being added, updated, or removed, the UI automatically refreshes the lists, tables, and cards without requiring a page reload, creating a seamless shopping experience.

To make the interface more user-friendly, SHOP.IT integrates field validation and instant feedback. react-hook-form checks for required fields, validates email formats, ensures numeric inputs where needed, and prevents incomplete submissions. Whenever users perform actions like adding a product to the cart, updating quantities, submitting a form, or completing a purchase, react-toastify displays success or error notifications. These toast messages provide clear, instant feedback and help guide the user through each interaction smoothly.

Data Provider

Code Description:-

The auth.js file in the SHOP.IT application is responsible for handling all authentication-related operations, including user registration, login, logout, and session management. This module acts as the central logic for verifying users and ensuring secure access to protected pages such as Dashboard, Products, Cart, Wishlist, and Orders. It communicates with the JSON-Server backend to validate user credentials and stores essential session details inside the browser's localStorage.

Inside the file, the signUp function sends a POST request to /users to create a new user account with fields such as name, email, and password. Before creating the account, the system checks if an email already exists to prevent duplicate registrations. The signIn function handles user login by verifying the provided email and password against the stored user records. If the credentials match, the user's id, name, and email are saved to localStorage, and the user is redirected to the Dashboard.

The module also contains an isAuthenticated helper, which checks whether a user session exists by confirming the presence of a valid userId in localStorage. This function is used by ProtectedRoute components to restrict access to internal pages like Cart, Wishlist, and Orders, ensuring that only logged-in users can interact with these features. Additionally, the logout function clears all authentication data from localStorage and redirects the user back to the login page, ensuring a clean and secure logout process.

By centralizing all authentication logic in one place, the auth.js file keeps the project organized, improves security, and makes it easier for the entire application to remain consistent when handling user sessions.

auth.js

application > src > api > auth.js > ...

```
// Frontend-only mock auth using localStorage so the app
// works without any backend server.

const USERS_KEY = "mockUsers";

const loadUsers = () => {
  try {
    return JSON.parse(localStorage.getItem(USERS_KEY)) || [];
  } catch {
    return [];
  }
};

const saveUsers = (users) => {
  localStorage.setItem(USERS_KEY, JSON.stringify(users));
};

export const userRegister = async (name, email, password) => {
  const users = loadUsers();
  const existing = users.find((u) => u.email === email);
  if (existing) {
    throw new Error("User already exists");
  }

  const newUser = {
    id: Date.now(),
    name,
    email,
    password, // For demo only — do NOT store plain passwords in real apps
  };

  const updated = [...users, newUser];
  saveUsers(updated);

  // Return a fake token like a real backend would
  return {
    token: `mock-token-${newUser.id}`,
    user: { id: newUser.id, name: newUser.name, email: newUser.email },
  };
};

export const userLogin = async (email, password) => {
  const users = loadUsers();
  const user = users.find((u) => u.email === email && u.password === password);
  if (!user) {
    throw new Error("Invalid email or password");
  }

  return {
    token: `mock-token-${user.id}`,
    user: { id: user.id, name: user.name, email: user.email },
  };
};
```

Code Component

cation > src > api > JS cartAPI.js > ...

```
import axios from "axios";
const API_URL = 'http://localhost:3003';

// Cart Operations
export const getCart = async (userId) => {
  try {
    const { data } = await axios.get(`${API_URL}/carts?userId=${userId}`);
    return data[0] || { userId, items: [] };
  } catch (error) {
    console.error('Error fetching cart:', error);
    throw error;
  }
};

export const addToCart = async (userId, productId, quantity = 1) => {
  try {
    // First, get the current cart or create a new one
    const cart = await getCart(userId);

    // Check if the product is already in the cart
    const existingItemIndex = cart.items?.findIndex(item => item.productId === productId) || -1;

    if (existingItemIndex >= 0) {
      // Update quantity if product exists
      cart.items[existingItemIndex].quantity += quantity;
    } else {
      // Add new item to cart
      if (!cart.items) {
        cart.items = [];
      }
      cart.items.push({ productId, quantity });
    }

    // Save the updated cart
    if (cart.id) {
      const { data } = await axios.put(`${API_URL}/carts/${cart.id}`, cart);
      return data;
    } else {
      const { data } = await axios.post(`${API_URL}/carts`, { ...cart, userId });
      return data;
    }
  } catch (error) {
    console.error('Error adding to cart:', error);
    throw error;
  }
};

export const removeFromCart = async (cartId, productId) => {
  try {
    const { data: cart } = await axios.get(`${API_URL}/carts/${cartId}`);
    cart.items = cart.items.filter(item => item.productId !== productId);

    const { data } = await axios.put(`${API_URL}/carts/${cartId}`, cart);
    return data;
  } catch (error) {
```

Code Description :-

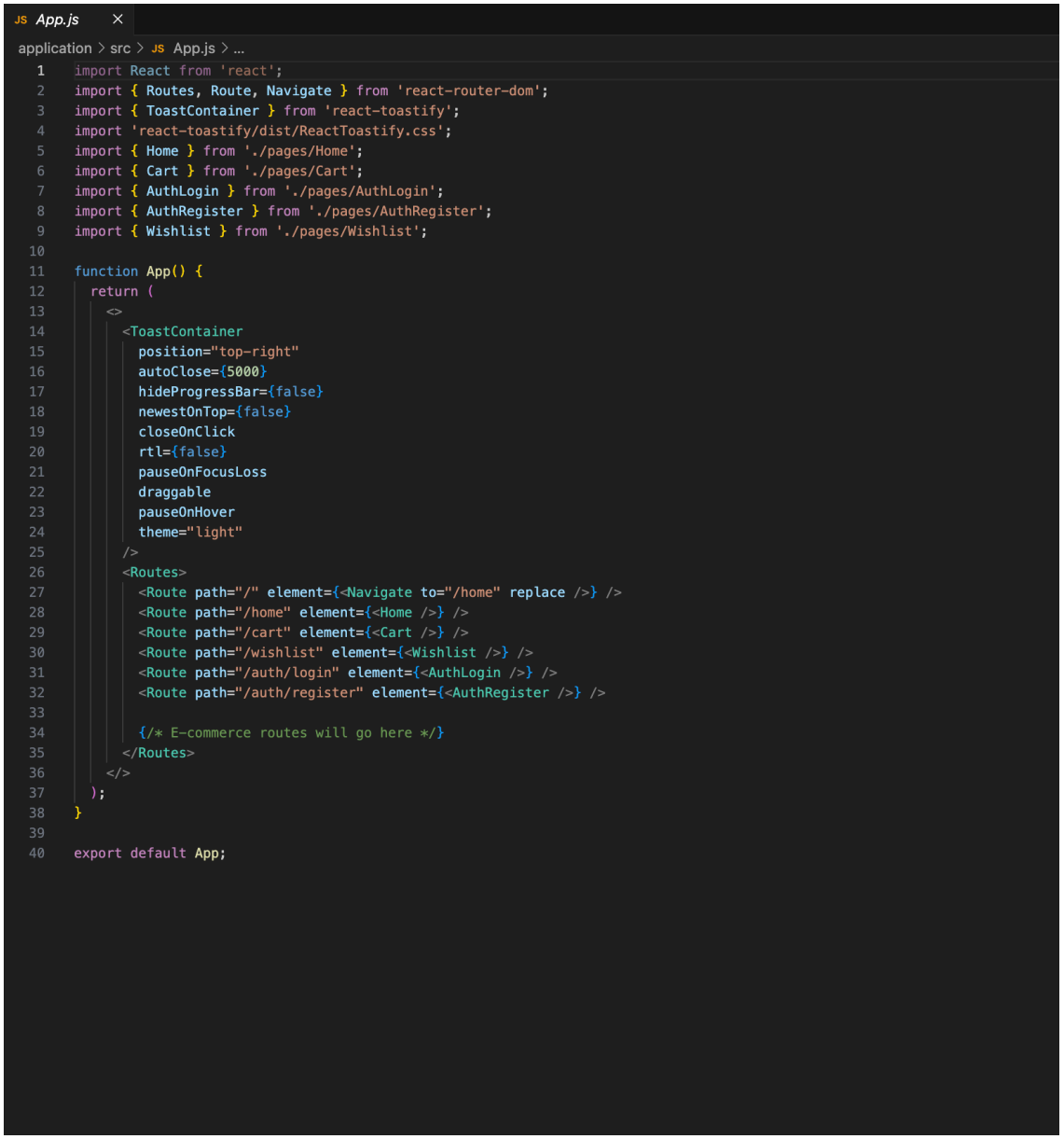
The Dashboard component in the SHOP.IT application begins by importing useState and useEffect from React to manage the dashboard values and automatically fetch all required data when the page loads. Instead of clients and projects, the dashboard in this application works with product-related modules such as products, cart items, wishlist items, and orders. The component communicates with different API services—productAPI, cartAPI, wishlistAPI, and orderAPI—to collect real-time information from the backend JSON-Server. For the UI, the component uses a reusable StatsCard component along with icons from react-icons to present the overall summary in a clean, user-friendly layout. Several state variables are defined to store dashboard statistics such as total number of products, number of items added to cart, wishlist count, total orders, delivered orders, and pending orders, along with a loading indicator. When the

Dashboard component loads, the useEffect hook triggers a function that fetches all data simultaneously using Promise.all, which makes the page load faster and ensures that the counts are accurate. Once the data is received, the dashboard calculates all important metrics, including how many products are available in the store, how many items users have in their cart or wishlist, the number of successful orders, and the overall activity of the application. For orders, it also checks status fields to categorize them into delivered, pending, or cancelled. The layout of the Dashboard is styled using Tailwind CSS, featuring a header with the dashboard title, a responsive grid of StatsCards, and a separate order summary displaying the overall shopping activity of the user. While the data is being loaded, a simple "Loading..." message provides better user experience. Finally, the Dashboard component is exported and connected to the protected /dashboard route so that only authenticated SHOP.IT users can view it.Code-Editor Component

Code Description :-

The Invoices page imports React hooks like useState and useEffect to manage the invoice list, modal visibility, and the dynamic line items used inside the invoice form. It also uses useForm from react-hook-form to control every input field, perform validation, and automatically update values like tax and discount for live total calculations. For user feedback, toast from react-toastify displays success or error messages, while useNavigate helps redirect the user to the detailed invoice view page. The component loads data from the backend using invoiceAPI, projectAPI, and clientAPI defined in axiosConfig.js, allowing it to fetch and store invoices, projects, and clients. Several state variables are maintained, including lists for invoices, projects, and clients, a loading state, a showModal state to open and close the editor, an editingInvoice to track when an invoice is being updated, and an items array to hold all the line items with quantity and price. When the component loads, useEffect runs a fetchData function that retrieves invoices, projects, and clients in parallel and saves them into state.

Total calculations for each invoice are handled by a calculateTotals function, which computes the subtotal, tax amount, discount amount, and the final total. When the form is submitted, it builds an invoiceData object containing client and project details, dates, totals, status, and cleaned line items, and then either creates a new invoice or updates an existing one depending on whether editingInvoice is active. Various handler functions such as handleEdit, handleDelete, handleAddNew, addItem, removeItem, updateItem, and handleCloseModal control the modal and the dynamic line items. Helper functions like getClientName, getProjectName, and getStatusBadge make the UI more readable by showing proper names and colored status badges in the table. The overall UI is built with Tailwind CSS, including a clean header with an "Invoices" title, an Add Invoice button, a responsive table listing all invoices, and a modal that works as the invoice editor. Finally, the component is exported so that App.jsx can use it on the /invoices route.



Project Execution – ClientSync

Step 1: Install Dependencies

To begin running the ClientSync application, the first step is to install all necessary dependencies. After opening the project folder in a code editor or terminal, the command npm install is executed. This installs all required Node modules listed in the package.json file, including React, Vite, Tailwind CSS, React Router, Axios, JSON-Server, react-hook-form, react-toastify, and other supporting libraries. Once the installation is complete, the environment is fully prepared to run both the frontend and backend parts of the application.

Step 2: Start JSON-Server (Backend API)

The backend of ClientSync is powered by JSON-Server, which acts as a lightweight mock REST API. In a separate terminal, running the command npm run json-server

starts the backend service on http://localhost:4000. This server uses the db.json file to store and manage all data related to clients, projects, invoices, and payments. Every CRUD operation performed in the application communicates with this backend, making it essential for testing and running the project during development.

Step 3: Start React Application (Frontend)

Once the backend is running, the next step is to start the frontend. Opening a second terminal and executing npm run dev launches the React application using Vite's fast development server. The frontend becomes accessible at http://localhost:5173, allowing you to interact with the user interface. Vite handles hot-reloading, so any changes made to the code are instantly reflected in the browser without needing to restart the server.

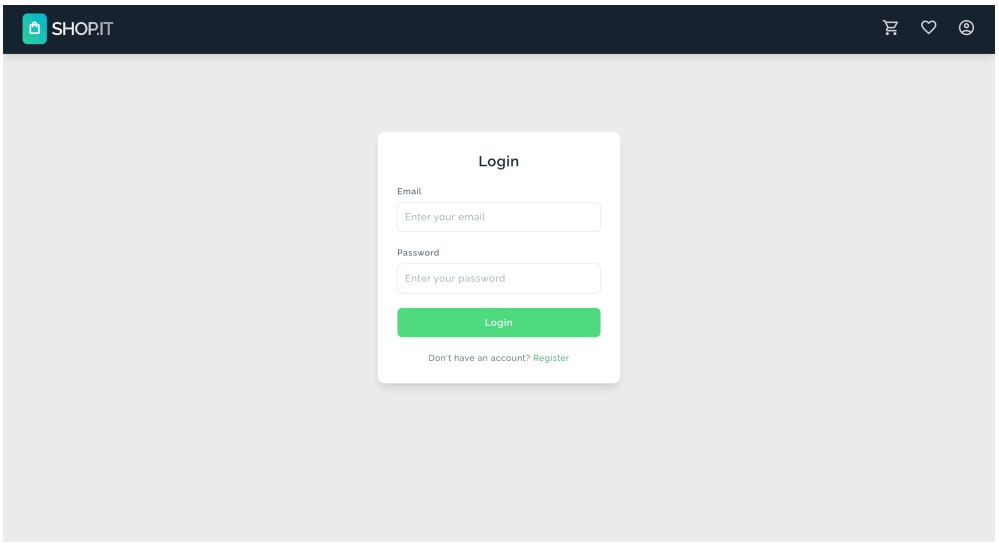
Step 4: Access the Application

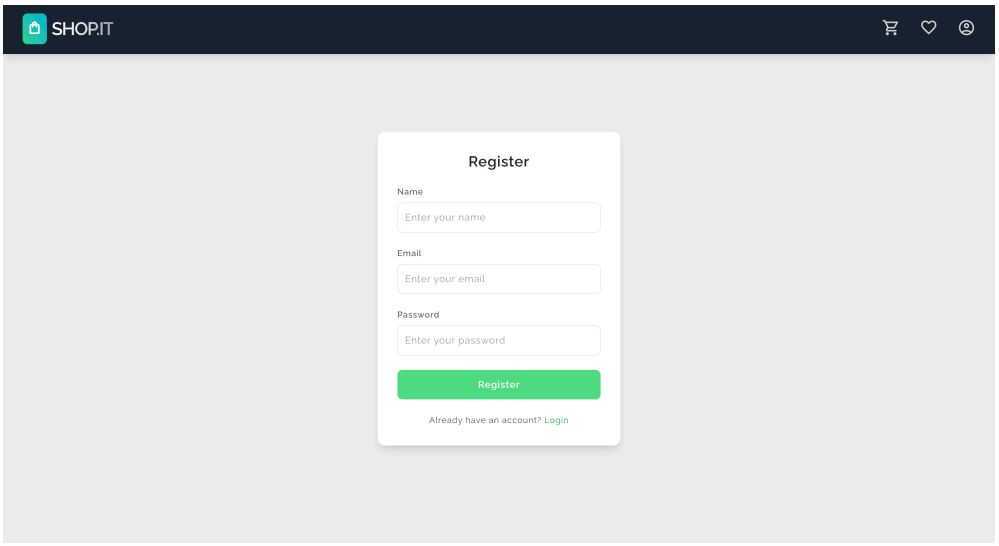
After both servers are running, the application can be accessed through a web browser by navigating to http://localhost:5173. From here, users can use the Homepage, Sign Up, and Sign In screens to log into the system. Once authenticated, they can explore all major modules of ClientSync. The Dashboard provides an overview of key metrics such as total clients, projects, invoices, and revenue. The Clients page allows adding, editing, and deleting client information. Projects can be linked to clients and managed easily. The Invoices module supports creating invoices with multiple line items and automatic total calculations. The Invoice View page allows users to open a specific invoice and export it as a PDF. The Payments section records payments made toward invoices and updates their status accordingly. All these modules work together to deliver a smooth CRM and billing experience.

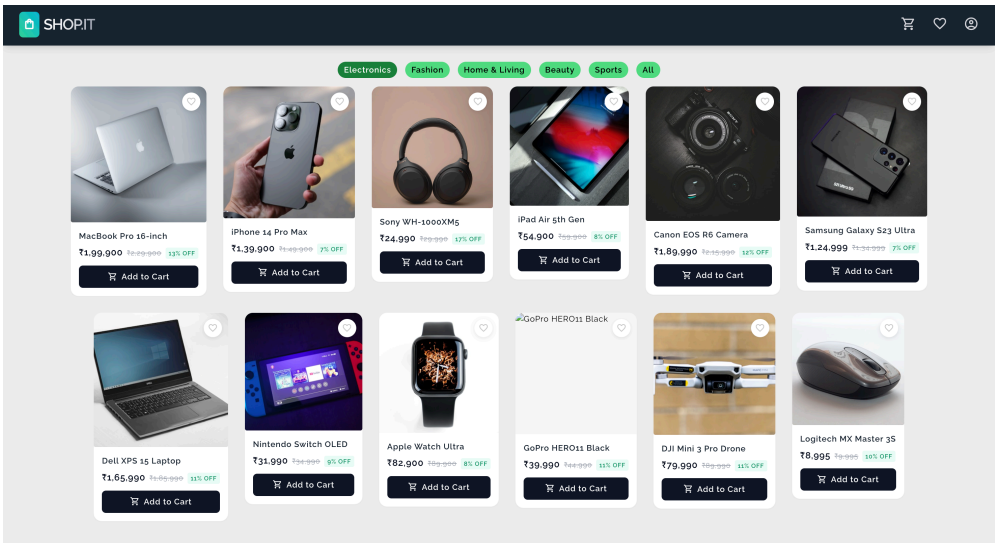
Step 5: Validate Functionality

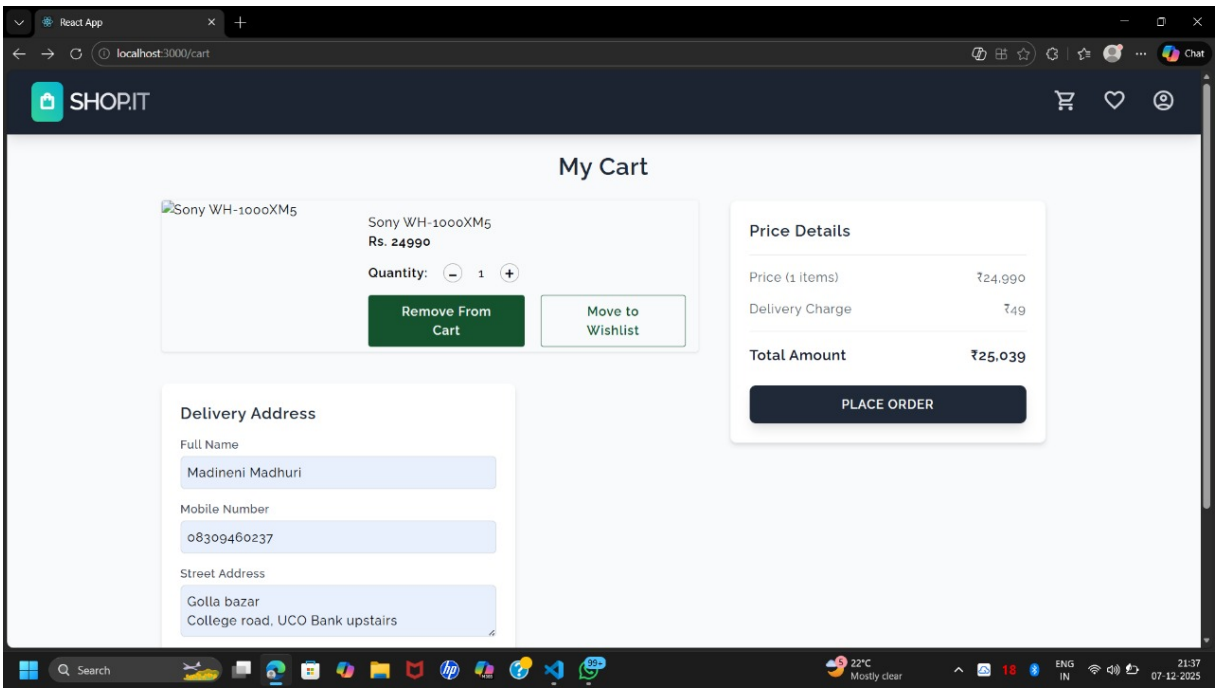
The final step involves validating the overall functionality of the application. This includes checking whether all CRUD operations—such as adding, viewing, updating, and deleting records—work correctly across all modules. Form validations must be tested to ensure that users enter proper data, while toast notifications should appear appropriately for success or error messages. Additionally, the responsiveness of the application should be verified across different screen sizes to confirm that the layout, tables, forms, and navigation behave correctly on both desktop and mobile devices.

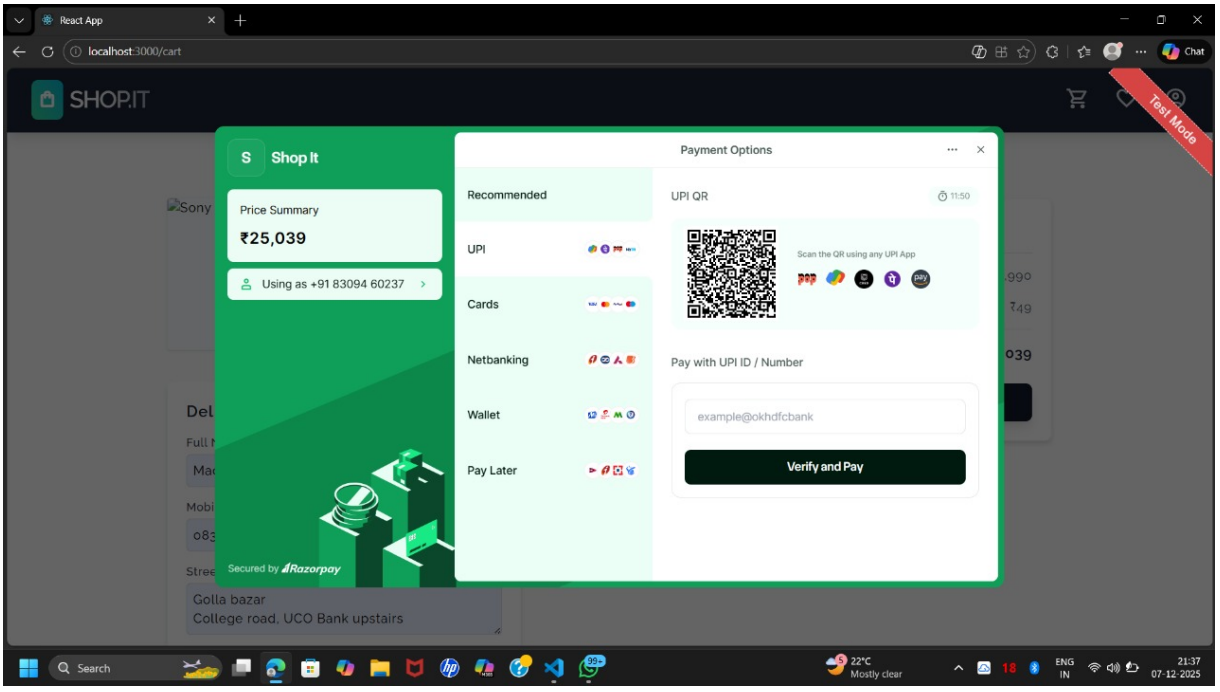
OUTPUT

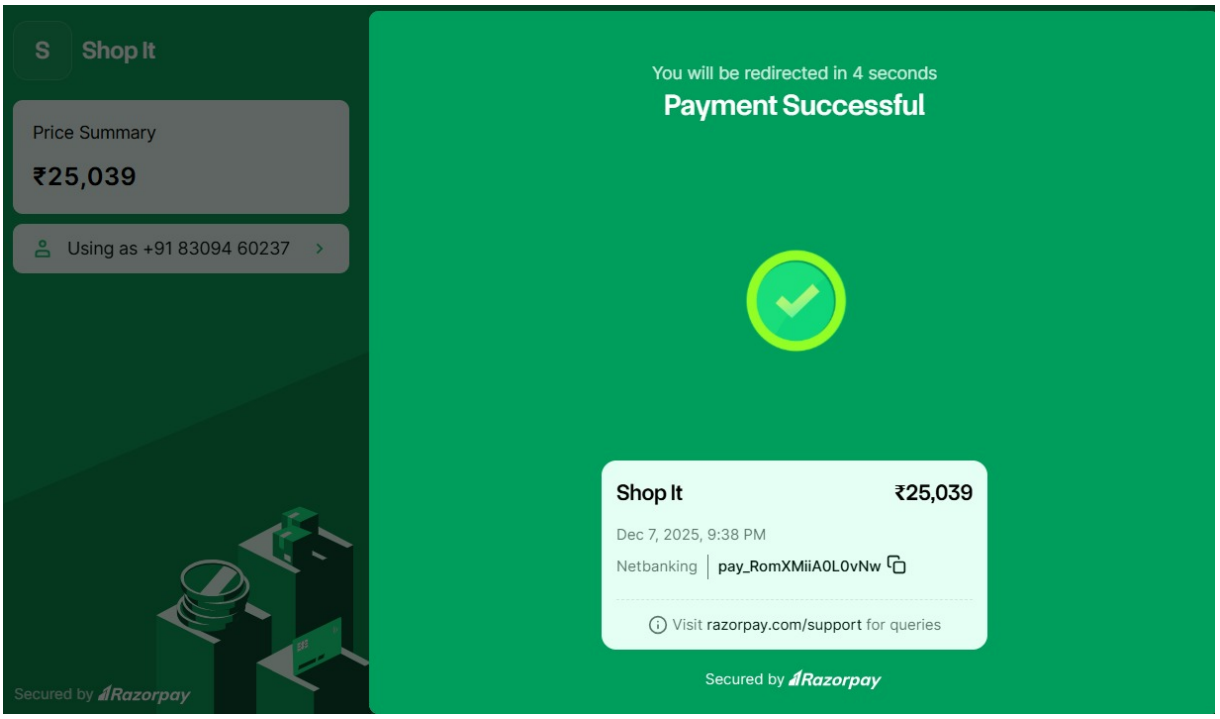

SHOP.IT
Login
Email
Enter your email
Password
Enter your password
Login
Don't have an account? Register


SHOP.IT
Register
Name
Enter your name
Email
Enter your email
Password
Enter your password
Register
Already have an account? Login


SHOP.IT
Electronics
Fashion
Home & Living
Beauty
Sports
All
MacBook Pro 16-inch
₹1,99,900
Add to Cart
iPhone 14 Pro Max
₹1,39,900
Add to Cart
Sony WH-1000XM5
₹24,990
Add to Cart
iPad Air 5th Gen
₹54,900
Add to Cart
Canon EOS R6 Camera
₹1,89,990
Add to Cart
Samsung Galaxy S23 Ultra
₹1,24,999
Add to Cart
Dell XPS 15 Laptop
₹1,65,990
Add to Cart
Nintendo Switch OLED
₹31,990
Add to Cart
Apple Watch Ultra
₹82,900
Add to Cart
GoPro HERO11 Black
₹39,990
Add to Cart
DJI Mini 3 Pro Drone
₹79,990
Add to Cart
Logitech MX Master 3S
₹8,995
Add to Cart


React App
localhost:3000/cart
SHOP.IT
My Cart
Sony WH-1000XM5
Sony WH-1000XM5
Rs. 24990
Quantity: 1
Remove From Cart
Move to Wishlist
Price Details
Price (1 items) ₹24,990
Delivery Charge ₹49
Total Amount ₹25,039
PLACE ORDER
Delivery Address
Full Name
Madineni Madhuri
Mobile Number
08309460237
Street Address
Golla bazar
College road, UCO Bank upstairs


React App
localhost:3000/cart
SHOP.IT
Test Mode
Shop It
Price Summary
₹25,039
Using as +91 83094 60237
Payment Options
Recommended
UPI
Cards
Netbanking
Wallet
Pay Later
UPI QR
Scan the QR using any UPI App
Pay with UPI ID / Number
example@okhdfcbank
Verify and Pay
Secured by Razorpay
Golla bazar
College road, UCO Bank upstairs


S
Shop It
Price Summary
₹25,039
Using as +91 83094 60237
Secured by Razorpay
You will be redirected in 4 seconds
Payment Successful
Shop It
₹25,039
Dec 7, 2025, 9:38 PM
Netbanking
pay_RomXMiiA0L0vNw
Visit razorpay.com/support for queries
Secured by Razorpay