

E-Commerce App

Introduction:

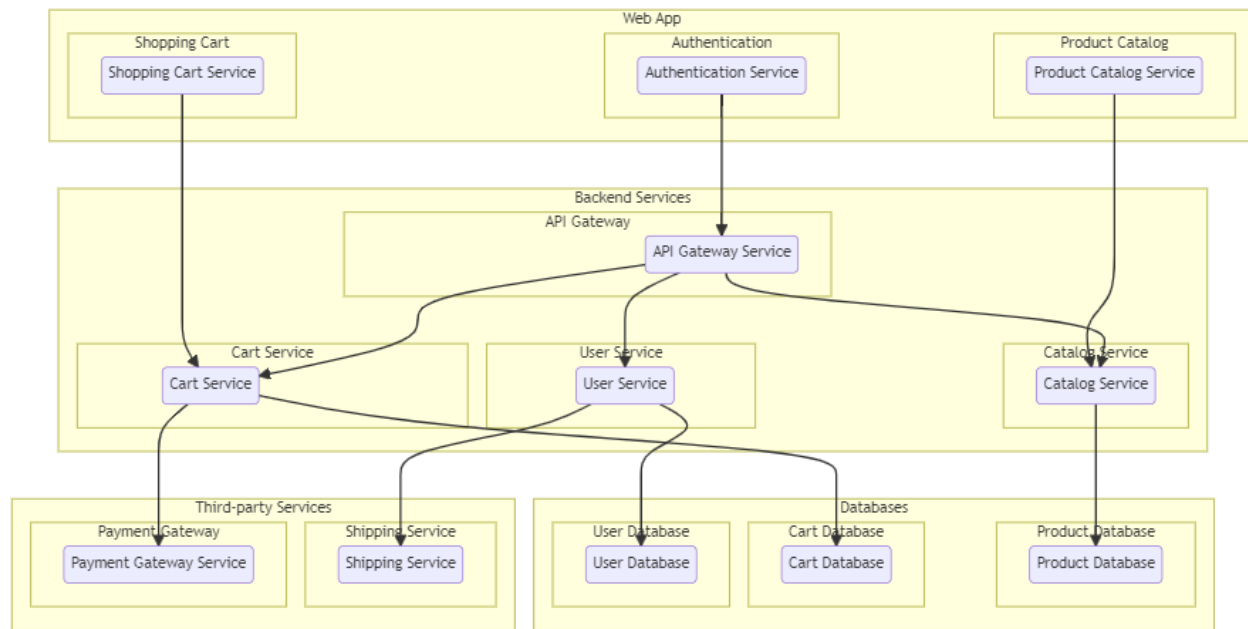
Welcome to our innovative e-commerce platform, where shopping becomes a seamless and delightful experience for everyone. Whether you're a tech enthusiast, a fashion-forward trendsetter, or a practical homemaker seeking everyday essentials, our app caters to all your needs. With a sleek and user-friendly interface, browsing through diverse categories and discovering exciting products has never been easier. Add items to your cart effortlessly and complete your purchases securely with our streamlined checkout process. Our top priority is your satisfaction, and we're committed to providing a hassle-free and enjoyable shopping journey.

For our valued sellers, managing your product listings, inventory, and orders is a breeze with our robust backend functionalities. Stay in control and watch your business thrive with ease. Meanwhile, our dedicated administrators ensure that your shopping experience is smooth sailing by handling customer inquiries, processing payments securely, and continuously monitoring the app's performance.

At the heart of our platform lies a commitment to security and privacy. Rest assured that your data is protected, transactions are secure, and your personal information remains confidential. Building trust with our customers is of utmost importance, and we strive to maintain a safe and secure platform for your online shopping needs.

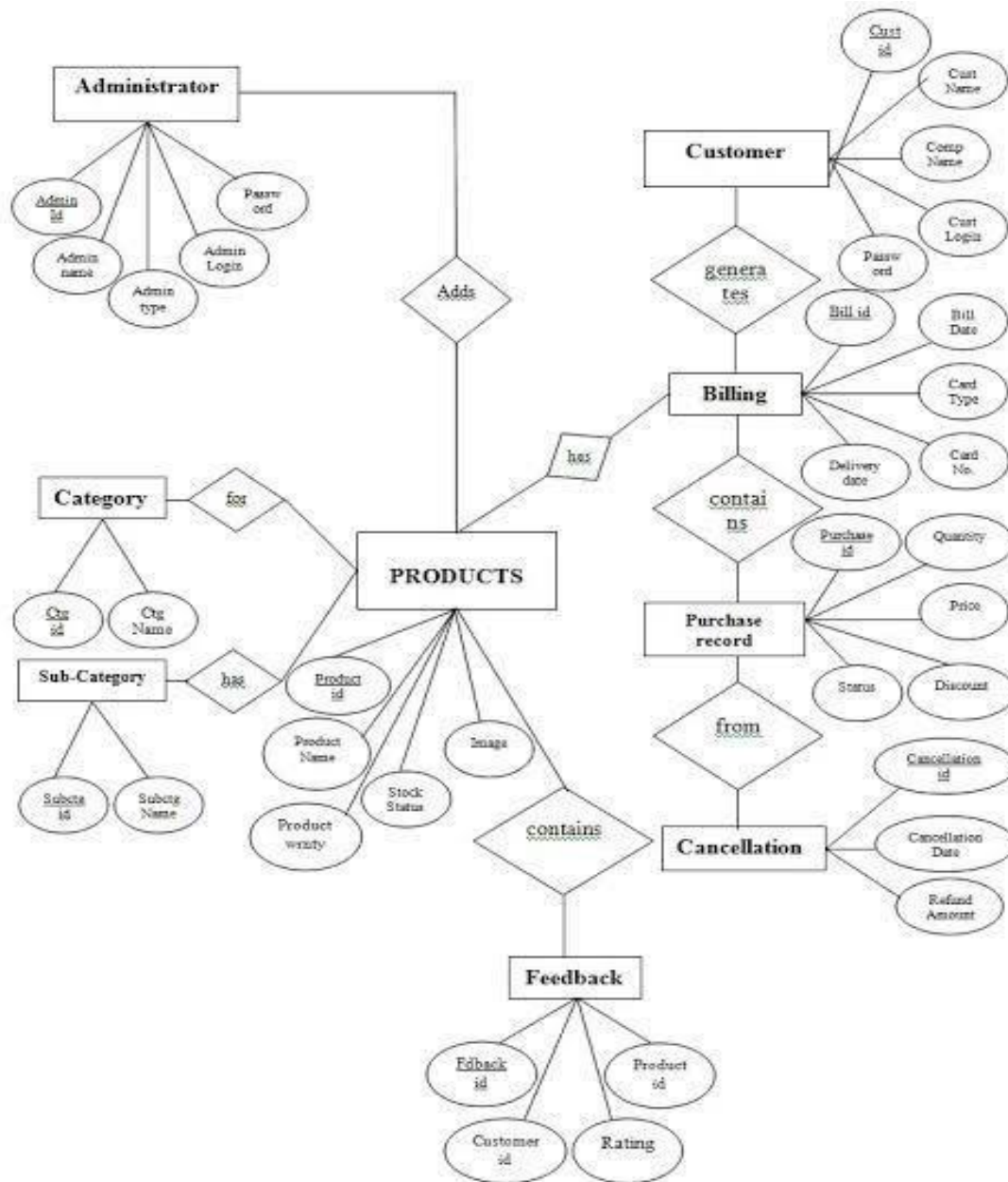
Thank you for joining us, and we look forward to providing you with an exceptional shopping experience. Happy shopping with our cutting-edge e-commerce app!

Technical Architecture:



The technical architecture of an e-commerce app typically involves a client-server model, where the frontend represents the client and the backend serves as the server. The frontend is responsible for user interface, interaction, and presentation, while the backend handles data storage, business logic, and integration with external services like payment gateways and databases. Communication between the frontend and backend is typically facilitated through APIs, enabling seamless data exchange and functionality.

ER Diagram:



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

Key Features:

Product Catalog: Our E-commerce app provides an extensive product catalog with various categories and subcategories. Users can easily search, browse, and filter products based on their preferences, making it effortless to find the desired items.

Shopping Cart and Checkout: The app includes a shopping cart feature that enables users to add products, review their cart, and proceed to checkout. The checkout process offers multiple payment options, ensuring a smooth and secure transaction experience.

Product Reviews and Ratings: Customers can provide feedback and rate products, helping other users make informed purchasing decisions. This feature fosters a sense of community and trust among users.

Order Tracking: Once an order is placed, users can track its status in real-time. They receive updates on order processing, shipping, and delivery, providing transparency and peace of mind.

Admin Dashboard: For administrators, our E-commerce app offers a comprehensive dashboard to manage products, inventory, orders, and customer information. It provides insights into sales performance, stock levels, and customer analytics, enabling efficient business operations.

Order Management: The app manages the order lifecycle, including order placement, tracking, and status updates. Users can view their order history, track shipments, and request returns or cancellations.

Search and Filtering: Users can search for products using keywords and apply filters to narrow down the search results based on criteria such as price range, brand, or customer ratings.

PRE REQUISITES:

To develop a full-stack e-commerce app using AngularJS, Node.js, and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

Node.js and npm: Install Node.js, which includes npm (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server side.

- Download: <https://nodejs.org/en/download/>
- Installation instructions: <https://nodejs.org/en/download/package-manager/>

MongoDB: Set up a MongoDB database to store hotel and booking information. Install MongoDB locally or use a cloud-based MongoDB service.

- Download: <https://www.mongodb.com/try/download/community>
- Installation instructions: <https://docs.mongodb.com/manual/installation/>

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing, middleware, and API development.

- Installation: Open your command prompt or terminal and run the following command: **npm install express**

Angular: Angular is a JavaScript framework for building client-side applications. Install Angular CLI (Command Line Interface) globally to create and manage your Angular project.

Install Angular CLI:

- Angular provides a command-line interface (CLI) tool that helps with project setup and development.
- Install the Angular CLI globally by running the following command:
npm install -g @angular/cli

Verify the Angular CLI installation:

- Run the following command to verify that the Angular CLI is installed correctly: **ng version**

You should see the version of the Angular CLI printed in the terminal if the installation was successful.

Create a new Angular project:

- Choose or create a directory where you want to set up your Angular project.
- Open your terminal or command prompt.
- Navigate to the selected directory using the **cd** command.
- Create a new Angular project by running the following command: **ng new client** Wait for the project to be created:

- The Angular CLI will generate the basic project structure and install the necessary dependencies

Navigate into the project directory:

- After the project creation is complete, navigate into the project directory by running the following command: **cd client**

Start the development server:

- To launch the development server and see your Angular app in the browser, run the following command: **ng serve / npm start**
- The Angular CLI will compile your app and start the development server.
- Open your web browser and navigate to <http://localhost:4200> to see your Angular app running.

You have successfully set up Angular on your machine and created a new Angular project. You can now start building your app by modifying the generated project files in the src directory.

Please note that these instructions provide a basic setup for Angular. You can explore more advanced configurations and features by referring to the official Angular documentation: <https://angular.io>

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

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

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

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

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

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

- Visual Studio Code: Download from <https://code.visualstudio.com/download>

- Sublime Text: Download from <https://www.sublimetext.com/download>
- WebStorm: Download from <https://www.jetbrains.com/webstorm/download>

To Connect the Database with Node JS go through the below provided link:

- Link: <https://www.section.io/engineering-education/nodejs-mongoosejs-mongodb/>

To run the existing E-Commerce App project downloaded from github:

Follow below steps:

1. Clone the Repository:

- Open your terminal or command prompt.
- Navigate to the directory where you want to store the e-commerce app.
- Execute the following command to clone the repository:

git clone <https://github.com/ManuvidyaS/E-commerce>

2. Install Dependencies:

- Navigate into the cloned repository directory:

```
cd E-CommerceApp
```

- Install the required dependencies by running the following command:

```
npm install
```

3. Start the Development Server:

- To start the development server, execute the following command:

```
npm run dev or npm run start
```

- The e-commerce app will be accessible at <http://localhost:5100> by default. You can change the port configuration in the .env file if needed.

4. Access the App:

- Open your web browser and navigate to <http://localhost:5100>.
- You should see the e-commerce app's homepage, indicating that the installation and setup were successful.

Video Tutorial Link to clone the project: -

<https://drive.google.com/file/d/1UWa9dngKUTIk8Evdf5Z-2I18DI-7koG8/view?usp=drivesdk>

Project Repository Link: <https://github.com/ManuvidyaS/E-commerce>

Congratulations! You have successfully installed and set up the e-commerce app on your local machine. You can now proceed with further customization, development, and testing as needed.

Roles and Responsibilities:

The project has two types of users – Seller and Customer. The roles and responsibilities of these two types of users can be inferred from the API endpoints defined in the code. Here is a summary:

Customer:

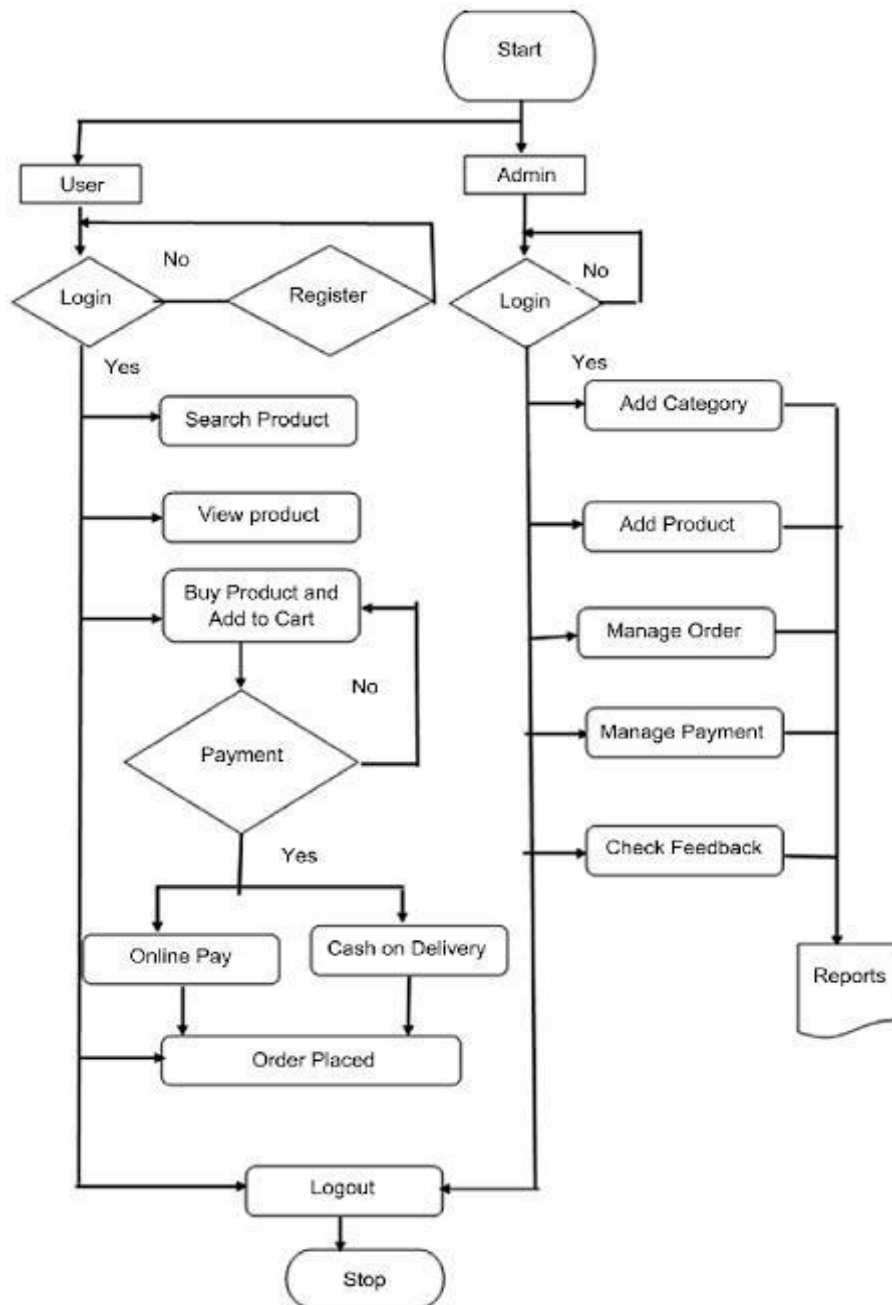
1. Create an account and log in to the system using their email and password.
2. Browse and search for products available on the platform.
3. View detailed product information, including description, price, and availability.
4. Add products to their cart for future purchase.
5. Proceed to checkout and place orders for selected products.
6. Make secure online payments for their orders.
7. Track the status of their orders.
8. Manage their profile information, including personal details and shipping addresses.
9. Provide feedback and reviews for products and sellers.
10. Access customer support for any queries or issues related to their orders.

Admin:

1. Manage and monitor the overall operation of the e-commerce platform.
2. Approve and onboard new sellers.
3. Monitor and moderate product listings, ensuring compliance with guidelines and policies.
4. Handle customer disputes and resolve issues.
5. Manage user accounts, including customer and seller profiles.
6. Analyze platform performance and generate reports on sales, customer behavior, and product popularity.
7. Implement and enforce platform policies, terms of service, and privacy regulations.
8. Continuously improve the platform's functionality, user experience, and security measure.

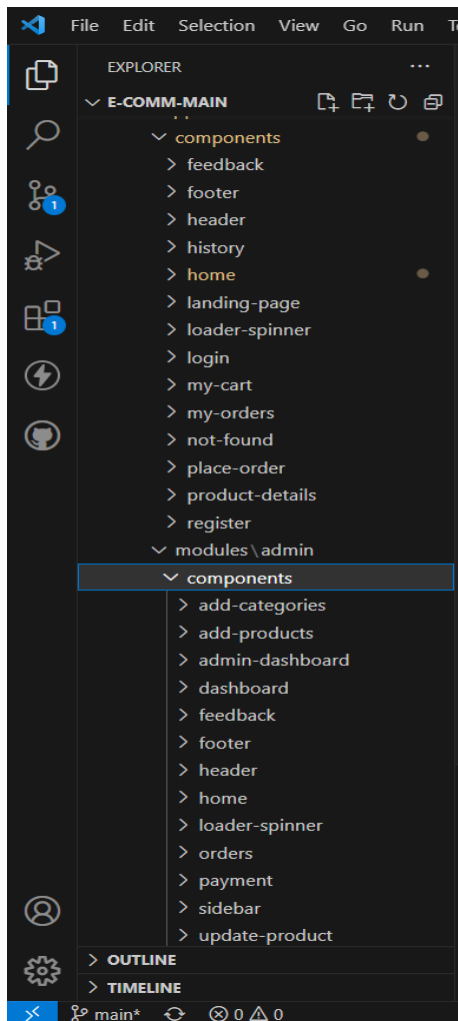
These roles and responsibilities are aimed at ensuring a smooth and efficient operation of the e-commerce app, providing a seamless experience for customers, sellers, and administrators.

Admin and User Flow:



The project flow for an e-commerce app involves user actions such as browsing products, adding items to the cart, proceeding to checkout, providing shipping details, selecting payment methods, making payments, and receiving order confirmation. Admin actions include managing products, viewing and processing orders, managing customers, and updating product details.

PROJECT STRUCTURE:



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

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

Project Flow:

Milestone 1: Project Setup and Configuration:

1. Install required tools and software:

- Node.js.
- MongoDB.
- Angular CLI.

2. Create project folders and files:

- Client folders.
- Server folders.

Milestone 2: Backend Development:

Setup express server:

- Install express.
- Create app.js file.
- Define API's

Configure MongoDB:

- Install Mongoose.
- Create database connection.
- Create Models.

Implement API end points:

- Implement CRUD operations.
- Test API endpoints.

Milestone 3: Web Development:

1. Setup Angular Application:

- Create Angular application using angular CLI.
- Configure Routing.
- Install required libraries.

2. Design UI components:

- Create Components.

- Implement layout and styling.
- Add navigation.

3. Implement frontend logic:

- Integration with API endpoints.
- Implement data binding.

Backend:

1. Set Up Project Structure:

- Create a new directory for your project and set up a package.json file using npm init command.
- Install necessary dependencies such as Express.js, Mongoose, and other required packages.

2. Database Configuration:

- Set up a MongoDB database either locally or using a cloud-based MongoDB service like MongoDB Atlas.
- Create a database and define the necessary collections for hotels, users, bookings, and other relevant data.

3. Create Express.js Server:

- Set up an Express.js server to handle HTTP requests and serve API endpoints.
- Configure middleware such as body-parser for parsing request bodies and cors for handling cross-origin requests.

4. Define API Routes:

- Create separate route files for different API functionalities such as hotels, users, bookings, and authentication.
- Define the necessary routes for listing hotels, handling user registration and login, managing bookings, etc.
- Implement route handlers using Express.js to handle requests and interact with the database.

5. Implement Data Models:

- Define Mongoose schemas for the different data entities like hotels, users, and bookings.
- Create corresponding Mongoose models to interact with the MongoDB database.
- Implement CRUD operations (Create, Read, Update, Delete) for each model to perform database operations.

User Management and Authentication:

- Implement user registration and login functionality.
- Choose an authentication mechanism like session-based authentication or token-based authentication (e.g., JWT).
- Store and hash user credentials securely.
- Implement middleware to authenticate API requests and authorize access to protected routes.

Product Catalog and Inventory Management:

- Design the database schema to store product details, pricing, availability, and inventory levels.
- Create APIs to retrieve product information, update inventory quantities, and handle search and filtering.
- Implement validations to ensure data integrity and consistency.

Shopping Cart and Order Management:

- Design the database schema to store shopping cart details and order information.
- Create APIs to handle cart operations like adding items, modifying quantities, and placing orders.
- Implement logic to calculate totals, apply discounts, and manage the order lifecycle.

Payment Gateway Integration:

- Choose a suitable payment gateway provider (e.g., Stripe, COD).
- Integrate the payment gateway SDK or API to handle secure payment processing.
- Implement APIs or callback endpoints to initiate transactions, handle payment callbacks, and receive payment confirmation.

Shipping and Logistics Integration:

- Identify shipping and logistics providers that align with your application's requirements.
- Utilize the APIs provided by these providers to calculate shipping costs, generate shipping labels, and track shipments.
- Implement APIs or services to fetch rates, generate labels, and obtain tracking information.

Database Integration:

- Choose a suitable database technology (e.g., MySQL, PostgreSQL, MongoDB) based on your application's requirements.
- Design the database schema to efficiently store and retrieve e-commerce data.
- Establish a connection to the database and handle data persistence and retrieval.

External Service Integration:

- Identify third-party services like email service providers, analytics services, or CRM systems that are required for your application.
- Utilize the APIs or SDKs provided by these services to exchange data and perform necessary operations.
- Implement the integration logic to send order confirmations, track user behavior, or manage customer relationships.

Security and Data Protection:

- Apply appropriate security measures like encryption techniques for secure data transmission and storage.
- Implement input validation and sanitization to prevent common security vulnerabilities.
- Implement access control to ensure authorized access to sensitive data.

Error Handling and Logging:

- Implement error handling mechanisms to handle exceptions and provide meaningful error messages to the frontend.
- Use logging frameworks to record application logs for monitoring and troubleshooting purposes.

Schema Usecase:

1. Users:

- Schema: userSchema
- Model: 'User'
- Purpose: Represents the schema and model for user data, including information like name, email, password, and other relevant details. It is used for user registration, authentication, and managing user-related functionalities.

2. Category:

- Schema: categorySchema
- Model: 'Category'
- Purpose: Represents the schema and model for product categories. It defines the structure for category data, such as name, description, and any other attributes related to categorizing products. It is used to manage and organize product categories within the e-commerce app.

3. Product:

- Schema: productSchema
- Model: 'Product'
- Purpose: Represents the schema and model for individual products available in the e-commerce app. It includes attributes like name, price, description, images, and other details specific to each product. It is used for product listing, details, and management within the app

4. AddToCart:

- Schema: addToCartSchema
- Model: 'AddToCart'
- Purpose: Represents the schema and model for items added to a user's cart. It captures information about the user, the product, quantity, and any additional details related to the cart item. It is used to manage the shopping cart functionality within the app.

5. Order:

- Schema: orderSchema
- Model: 'Order'
- Purpose: Represents the schema and model for customer orders placed in the e-commerce app. It includes details like order items, user information, payment status, shipping details, and more. It is used for managing the order lifecycle, tracking, and processing.

6. Payment:

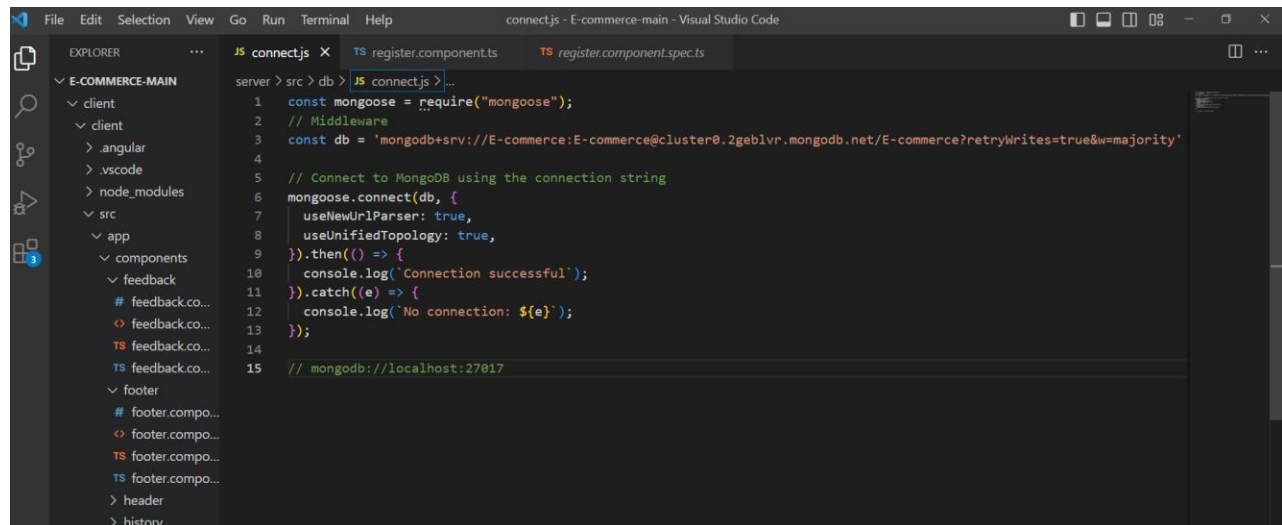
- Schema: paymentSchema
- Model: 'Payment'
- Purpose: Represents the schema and model for payment information associated with customer orders. It includes details like payment method, transaction ID, amount, and other relevant payment-related data. It is used to handle payment processing and tracking within the app.

7. Feedback:

- Schema: feedbackSchema
- Model: 'Feedback'
- Purpose: Represents the schema and model for customer feedback or reviews. It captures feedback text, ratings, user information, and any other relevant details. It is used to manage and display customer reviews for products or the overall e-commerce experience. These schemas and models provide the structure and functionality needed to interact with the respective MongoDB collections and perform CRUD operations (Create, Read, Update, Delete) for users, categories, products, cart items, orders, payments, and feedback within the e-commerce app.

Backend Explanation with code snippets:

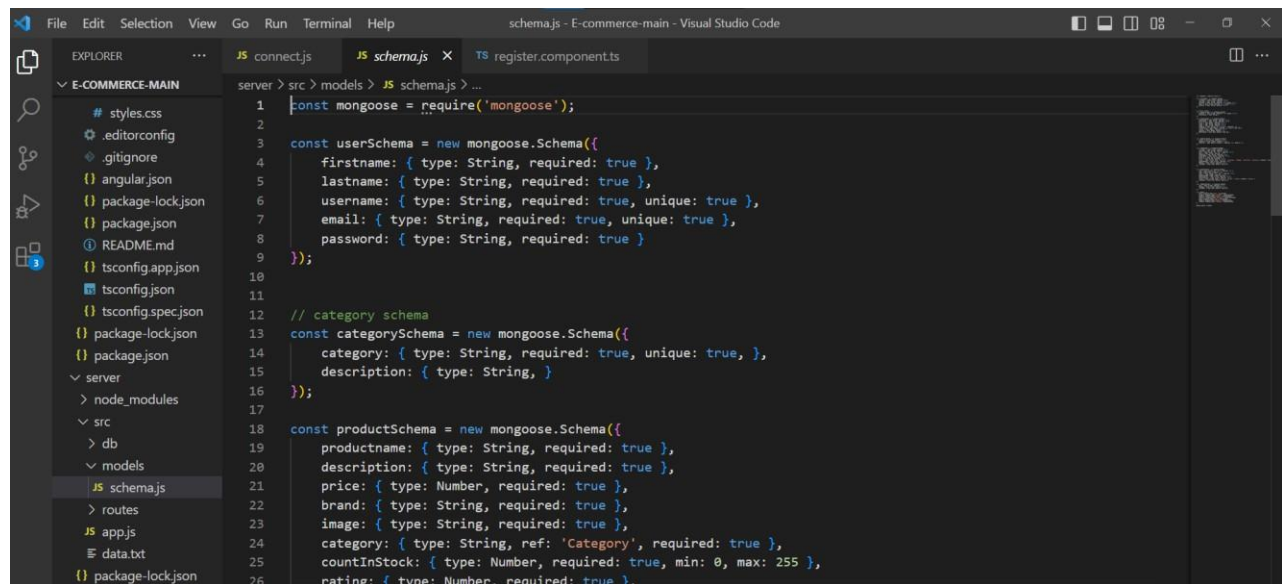
Database Connection:



The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The main editor window shows the file `register.component.ts` with the following code:

```
server > src > db > JS connectjs > ...
1  const mongoose = require("mongoose");
2  // Middleware
3  const db = 'mongodb+srv://E-commerce:E-commerce@cluster0.2geblvr.mongodb.net/E-commerce?retryWrites=true&w=majority'
4
5  // Connect to MongoDB using the connection string
6  mongoose.connect(db, {
7    useNewUrlParser: true,
8    useUnifiedTopology: true,
9  }).then(() => {
10     console.log('Connection successful');
11  }).catch((e) => {
12     console.log('No connection: ${e}');
13  });
14
15  // mongodb://localhost:27017
```

Schemas:



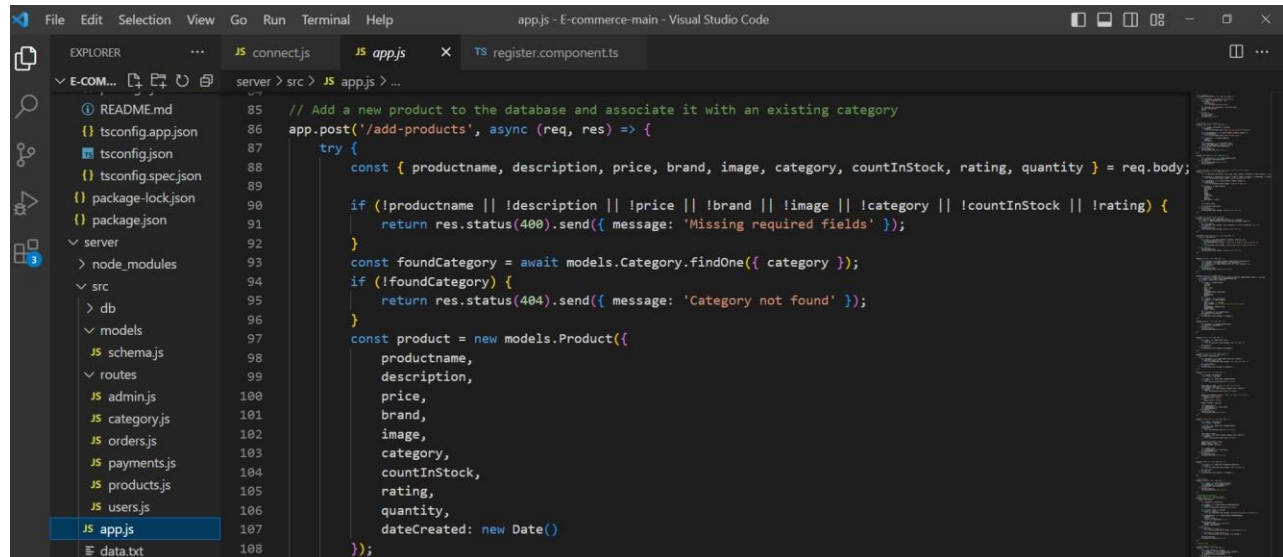
The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The main editor window shows the file `schema.js` with the following code:

```
server > src > models > JS schemas > ...
1  const mongoose = require('mongoose');
2
3  const userSchema = new mongoose.Schema({
4    firstname: { type: String, required: true },
5    lastname: { type: String, required: true },
6    username: { type: String, required: true, unique: true },
7    email: { type: String, required: true, unique: true },
8    password: { type: String, required: true }
9  });
10
11  // category schema
12  const categorySchema = new mongoose.Schema({
13    category: { type: String, required: true, unique: true },
14    description: { type: String, }
15  });
16
17  const productSchema = new mongoose.Schema({
18    productname: { type: String, required: true },
19    description: { type: String, required: true },
20    price: { type: Number, required: true },
21    brand: { type: String, required: true },
22    image: { type: String, required: true },
23    category: { type: String, ref: 'Category', required: true },
24    countInStock: { type: Number, required: true, min: 0, max: 255 },
25    rating: { type: Number, required: true },
26  });
```

```
server > src > models > JS schema.js > ...
28   dateCreated: { type: Date, default: Date.now }
29   });
30
31
32
33   const addToCartSchema = new mongoose.Schema({
34     userId: { type: String, required: true },
35     productId: { type: String, required: true },
36     quantity: { type: Number, minimum: 1, required: true, default: 1 },
37   });
38
39
40   const orderSchema = new mongoose.Schema({
41     firstName: { type: String, required: true },
42     lastName: { type: String, required: true },
43     user: { type: String, ref: 'User', required: true },
44     phone: { type: String, required: true },
45     productId: { type: String, required: true },
46     productName: { type: String, required: true },
47     quantity: { type: String, default: 1 },
48     price: { type: String, required: true },
49     status: { type: String, enum: ['Pending', 'Confirmed', 'Shipped', 'Delivered', 'Canceled'], default: 'Pending' },
50     paymentMethod: { type: String, required: true },
51     address: { type: String, required: true },
52     createdAt: { type: Date, default: Date.now }
53   });
```

```
59   order: { type: String, ref: 'Order', required: true },
60   amount: { type: Number, required: true },
61   paymentMethod: { type: String, required: true },
62   deliveryStatus: { type: String, required: true },
63   status: { type: String, enum: ['Pending', 'Success', 'Failed'], default: 'Pending' },
64   createdAt: { type: Date, default: Date.now }
65   });
66
67   const feedbackSchema = new mongoose.Schema({
68     user: { type: String, required: true },
69     message: { type: String, required: true },
70     createdAt: { type: Date, default: Date.now }
71   });
72
73   const models = {
74     Users: mongoose.model('User', userSchema),
75     Category: mongoose.model('Category', categorySchema),
76     Product: mongoose.model('Product', productSchema),
77     AddToCart: mongoose.model('AddToCart', addToCartSchema),
78     Order: mongoose.model('Order', orderSchema),
79     Payment: mongoose.model('Payment', paymentSchema),
80     Feedback: mongoose.model('Feedback', feedbackSchema)
81   };
82
83   module.exports = models;
84
```

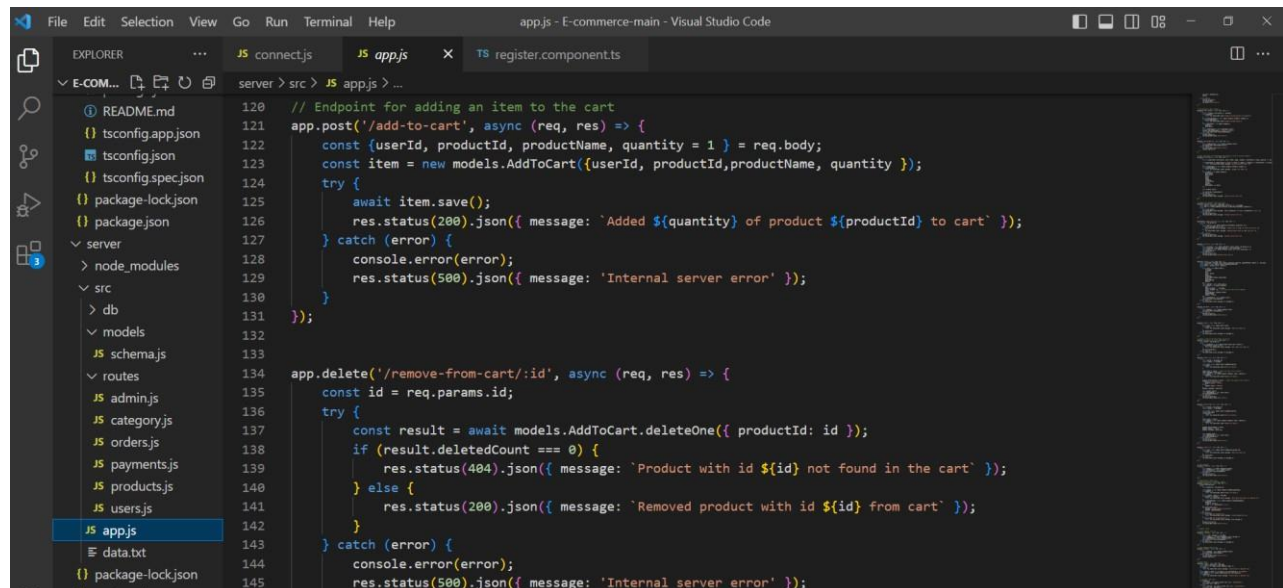
API's for Add Product:



```
85 // Add a new product to the database and associate it with an existing category
86 app.post('/add-products', async (req, res) => {
87   try {
88     const { productname, description, price, brand, image, category, countInStock, rating, quantity } = req.body;
89
90     if (!productname || !description || !price || !brand || !image || !category || !countInStock || !rating) {
91       return res.status(400).send({ message: 'Missing required fields' });
92     }
93     const foundCategory = await models.Category.findOne({ category });
94     if (!foundCategory) {
95       return res.status(404).send({ message: 'Category not found' });
96     }
97     const product = new models.Product({
98       productname,
99       description,
100       price,
101       brand,
102       image,
103       category,
104       countInStock,
105       rating,
106       quantity,
107       dateCreated: new Date()
108     });
```

The Add Product API enables users to create and add a new product to a system or database.

API's for Add to Cart, Remove from cart and get cart by id:

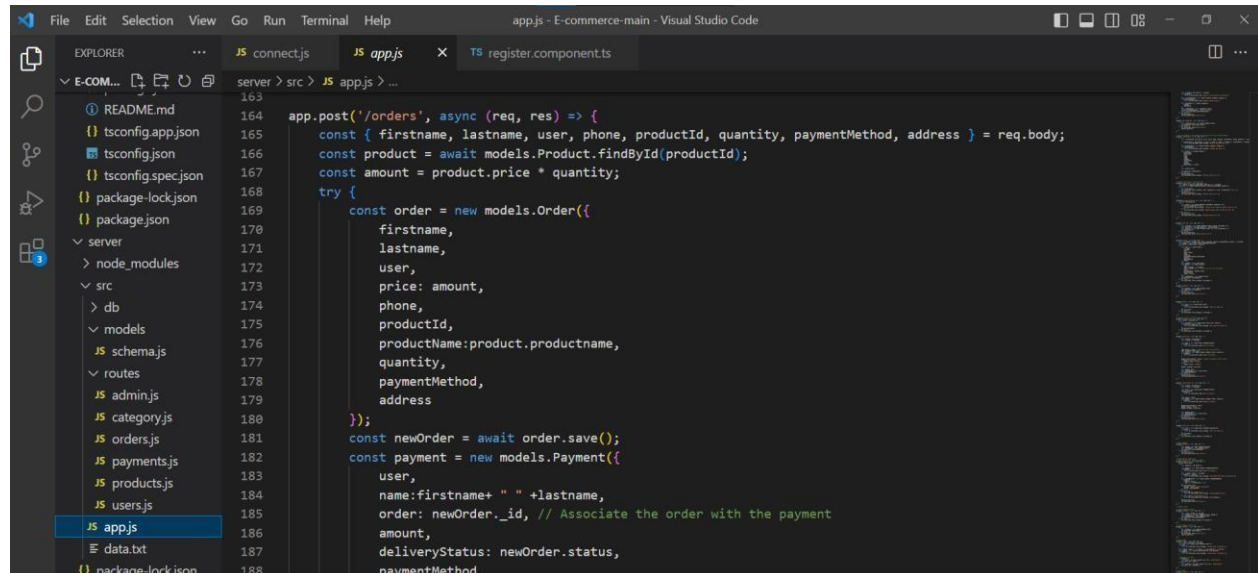


```
120 // Endpoint for adding an item to the cart
121 app.post('/add-to-cart', async (req, res) => {
122   const {userId, productId, productName, quantity = 1 } = req.body;
123   const item = new models.AddToCart({userId, productId,productName, quantity });
124   try {
125     await item.save();
126     res.status(200).json({ message: `Added ${quantity} of product ${productId} to cart` });
127   } catch (error) {
128     console.error(error);
129     res.status(500).json({ message: 'Internal server error' });
130   }
131 });
132
133 app.delete('/remove-from-cart/:id', async (req, res) => {
134   const id = req.params.id;
135   try {
136     const result = await models.AddToCart.deleteOne({ productId: id });
137     if (result.deletedCount === 0) {
138       res.status(404).json({ message: 'Product with id ${id} not found in the cart' });
139     } else {
140       res.status(200).json({ message: 'Removed product with id ${id} from cart' });
141     }
142   } catch (error) {
143     console.error(error);
144     res.status(500).json({ message: 'Internal server error' });
145   }
```

The Add to Cart API enables users to add items to a cart, the Remove from Cart API allows users to remove items from a cart, and the Get Cart by ID API retrieves the cart details basedon the provided ID.

API's for Post Orders and Get Payments:

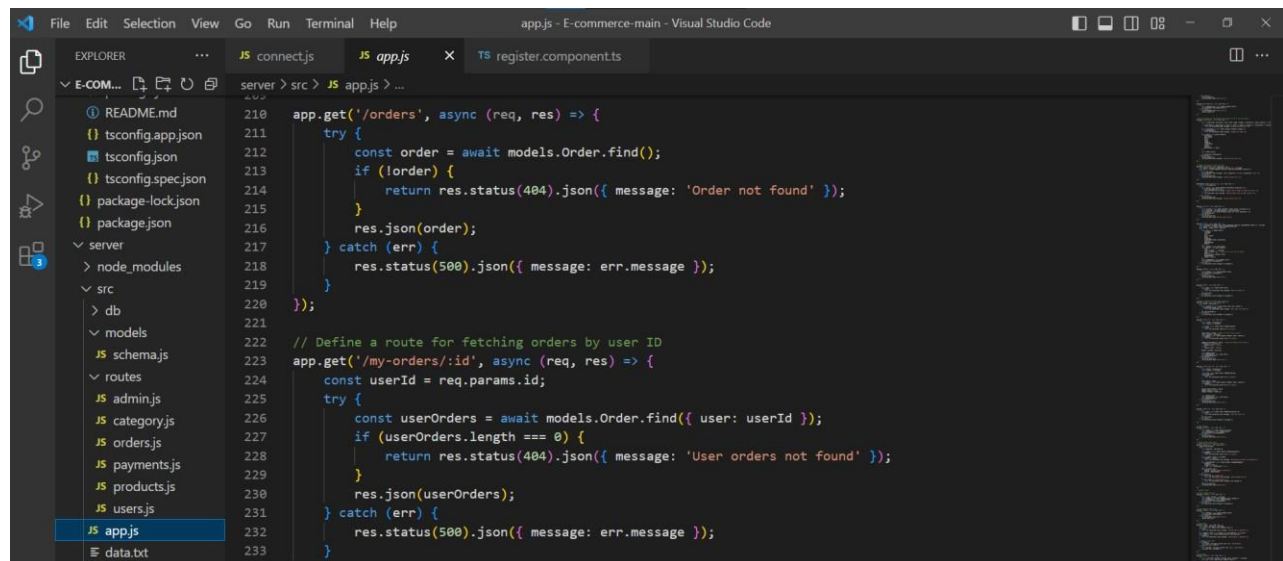
The Post Orders API is used to create and submit new orders, while the Get Payments API retrieves payment information for specific orders.



```
server > src > JS app.js > ...
163
164 app.post('/orders', async (req, res) => {
165   const { firstname, lastname, user, phone, productId, quantity, paymentMethod, address } = req.body;
166   const product = await models.Product.findById(productId);
167   const amount = product.price * quantity;
168   try {
169     const order = new models.Order({
170       firstname,
171       lastname,
172       user,
173       price: amount,
174       phone,
175       productId,
176       productName: product.productname,
177       quantity,
178       paymentMethod,
179       address
180     });
181     const newOrder = await order.save();
182     const payment = new models.Payment({
183       user,
184       name: firstname + " " + lastname,
185       order: newOrder._id, // Associate the order with the payment
186       amount,
187       deliveryStatus: newOrder.status,
188       paymentMethod,
```

API's for Get orders and get orders based on user Id:

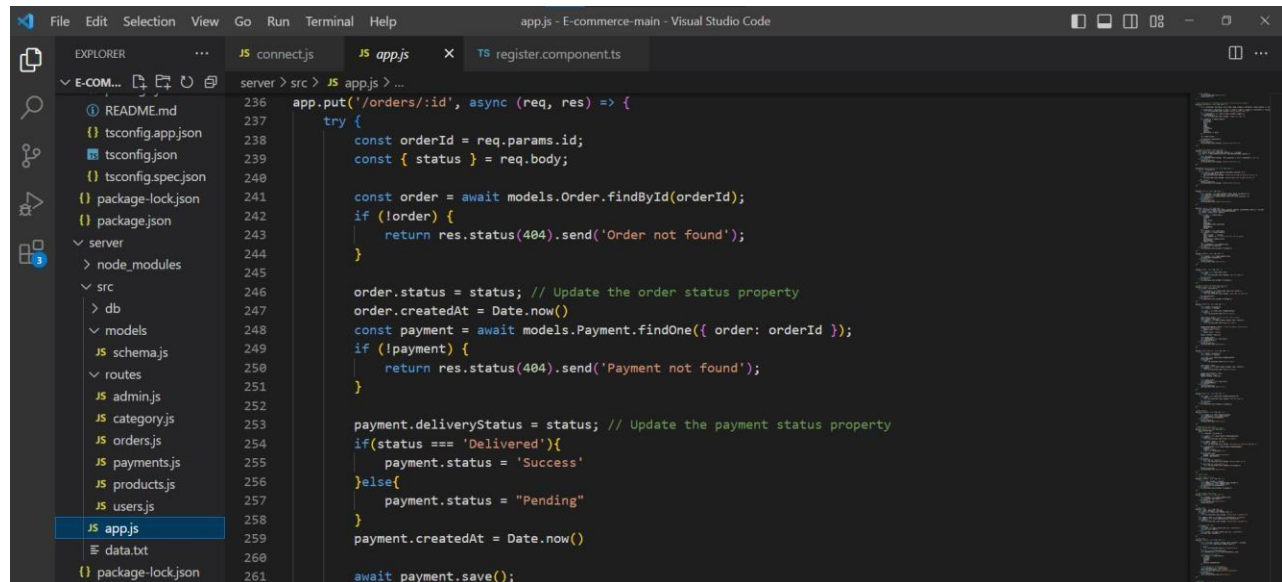
The Get Orders API retrieves all orders from the system, while the Get Orders by User ID API retrieves orders specific to a user ID.



```
server > src > JS app.js > ...
210 app.get('/orders', async (req, res) => {
211   try {
212     const order = await models.Order.find();
213     if (!order) {
214       return res.status(404).json({ message: 'Order not found' });
215     }
216     res.json(order);
217   } catch (err) {
218     res.status(500).json({ message: err.message });
219   }
220 });
221
222 // Define a route for fetching orders by user ID
223 app.get('/my-orders/:id', async (req, res) => {
224   const userId = req.params.id;
225   try {
226     const userOrders = await models.Order.find({ user: userId });
227     if (userOrders.length === 0) {
228       return res.status(404).json({ message: 'User orders not found' });
229     }
230     res.json(userOrders);
231   } catch (err) {
232     res.status(500).json({ message: err.message });
233   }
234 }
```


API for Update order based on Id:

The Update Order API allows users to modify an order based on its unique ID.

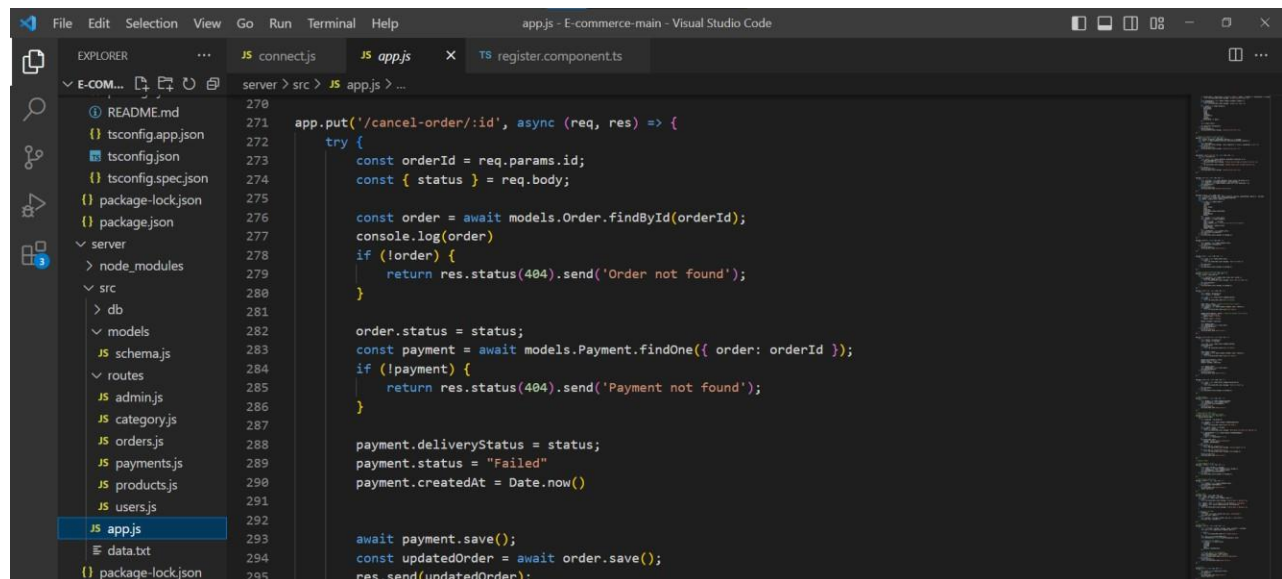


The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows the project structure with files like README.md, tsconfig.app.json, tsconfig.spec.json, package-lock.json, package.json, server, node_modules, src, db, models, schemas, routes, admin.js, category.js, orders.js, payments.js, products.js, users.js, app.js, data.txt, and package-lock.json. The code editor shows the implementation of the Update Order API in app.js. The code is as follows:

```
server > src > JS app.js > ...
236 app.put('/orders/:id', async (req, res) => {
237   try {
238     const orderId = req.params.id;
239     const { status } = req.body;
240
241     const order = await models.Order.findById(orderId);
242     if (!order) {
243       return res.status(404).send('Order not found');
244     }
245
246     order.status = status; // Update the order status property
247     order.createdAt = Date.now()
248     const payment = await models.Payment.findOne({ order: orderId });
249     if (!payment) {
250       return res.status(404).send('Payment not found');
251     }
252
253     payment.deliveryStatus = status; // Update the payment status property
254     if(status === 'Delivered'){
255       payment.status = 'Success'
256     }else{
257       payment.status = "Pending"
258     }
259     payment.createdAt = Date.now()
260
261     await payment.save();
```

API for cancel order based on Id:

The cancel Order API enables users to cancel an order by its unique ID.

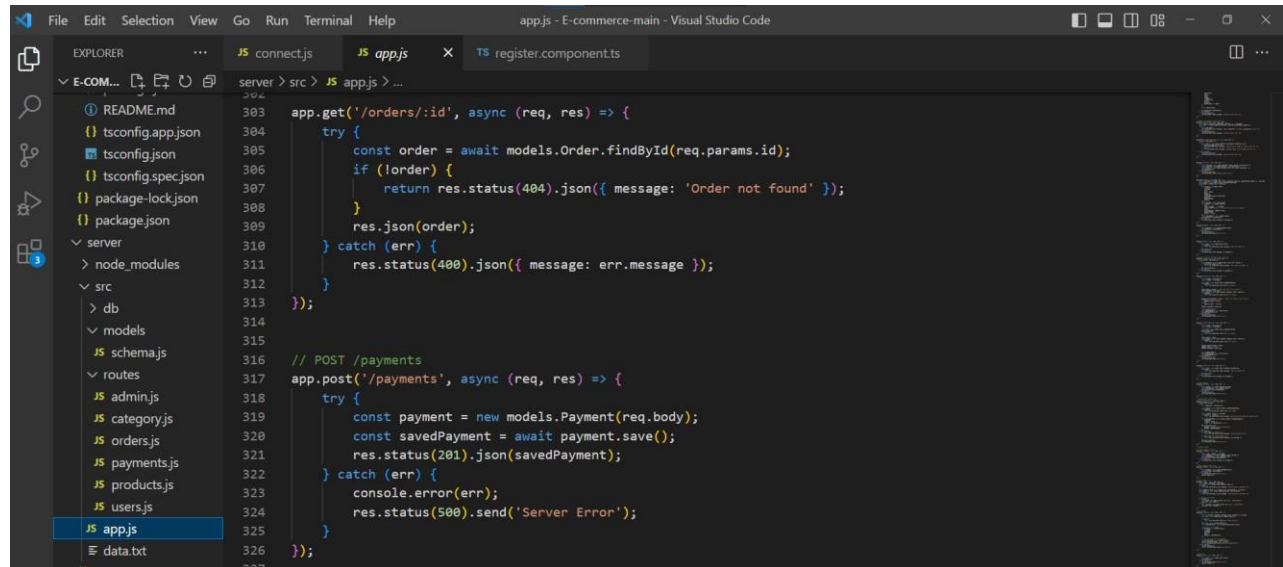


The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows the project structure with files like README.md, tsconfig.app.json, tsconfig.spec.json, package-lock.json, package.json, server, node_modules, src, db, models, schemas, routes, admin.js, category.js, orders.js, payments.js, products.js, users.js, app.js, data.txt, and package-lock.json. The code editor shows the implementation of the Cancel Order API in app.js. The code is as follows:

```
server > src > JS app.js > ...
270 app.put('/cancel-order/:id', async (req, res) => {
271   try {
272     const orderId = req.params.id;
273     const { status } = req.body;
274
275     const order = await models.Order.findById(orderId);
276     console.log(order)
277     if (!order) {
278       return res.status(404).send('Order not found');
279     }
280
281     order.status = status;
282     const payment = await models.Payment.findOne({ order: orderId });
283     if (!payment) {
284       return res.status(404).send('Payment not found');
285     }
286
287     payment.deliveryStatus = status;
288     payment.status = "Failed"
289     payment.createdAt = Date.now()
290
291     await payment.save();
292     const updatedOrder = await order.save();
293     res.send(updatedOrder);
294
295
```

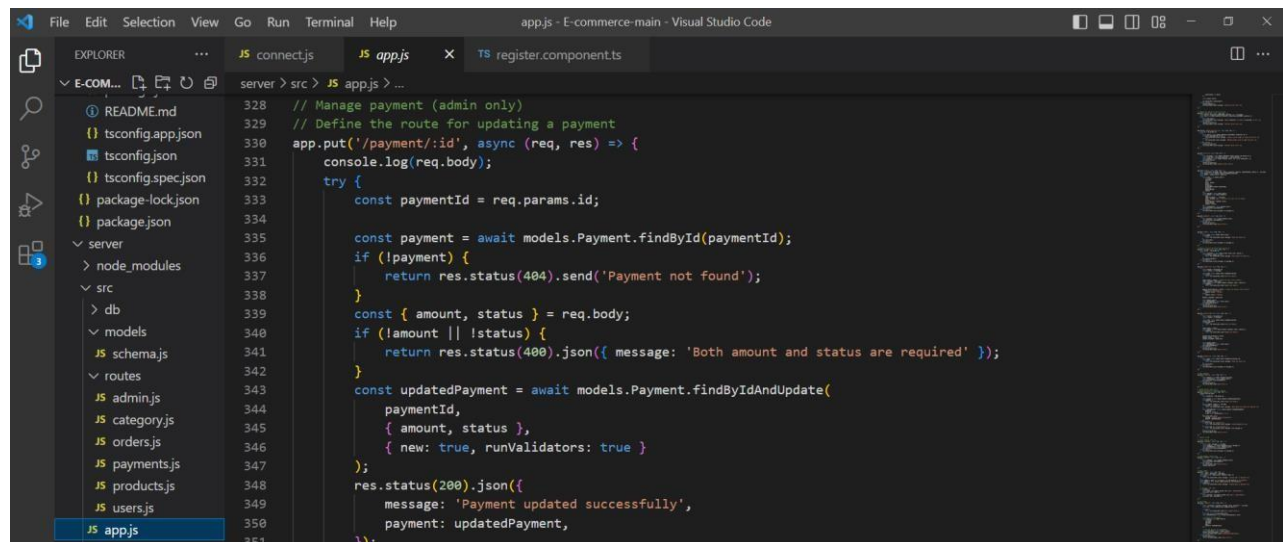
API's for Get orders based on user Id and post payment:

The Get Orders API retrieves orders based on user ID, while the Post Payment API processes and records a payment transaction.



This screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The main editor window shows the `app.js` file with the following code:

```
server > src > JS app.js > ...
303 app.get('/orders/:id', async (req, res) => {
304   try {
305     const order = await models.Order.findById(req.params.id);
306     if (!order) {
307       return res.status(404).json({ message: 'Order not found' });
308     }
309     res.json(order);
310   } catch (err) {
311     res.status(400).json({ message: err.message });
312   }
313 });
314
315 // POST /payments
316 app.post('/payments', async (req, res) => {
317   try {
318     const payment = new models.Payment(req.body);
319     const savedPayment = await payment.save();
320     res.status(201).json(savedPayment);
321   } catch (err) {
322     console.error(err);
323     res.status(500).send('Server Error');
324   }
325 }
326 );
327
```

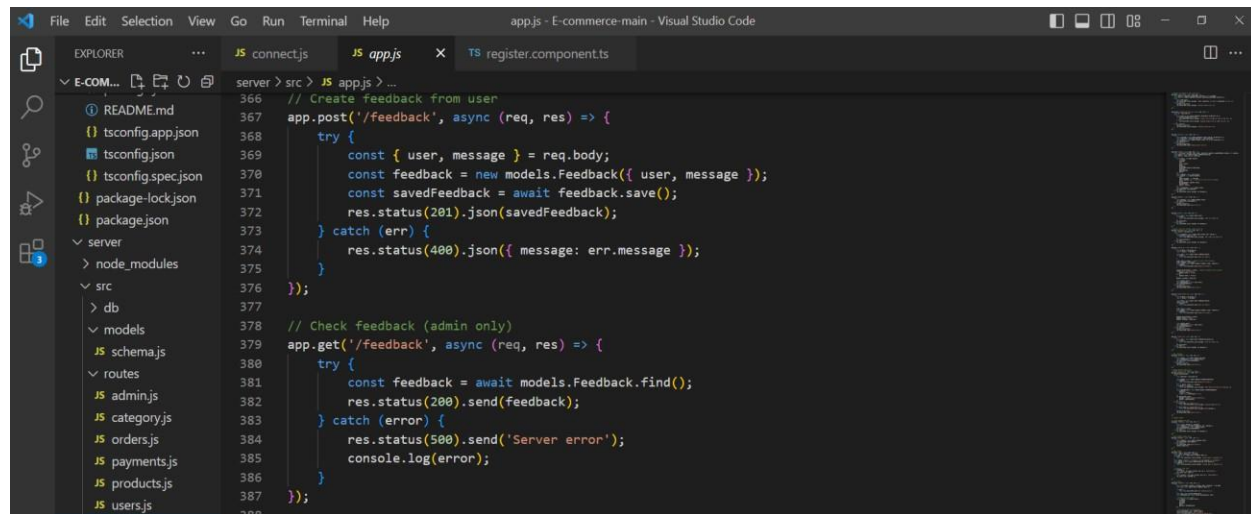


This screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The main editor window shows the `app.js` file with the following code:

```
server > src > JS app.js > ...
328 // Manage payment (admin only)
329 // Define the route for updating a payment
330 app.put('/payment/:id', async (req, res) => {
331   console.log(req.body);
332   try {
333     const paymentId = req.params.id;
334
335     const payment = await models.Payment.findById(paymentId);
336     if (!payment) {
337       return res.status(404).send('Payment not found');
338     }
339     const { amount, status } = req.body;
340     if (!amount || !status) {
341       return res.status(400).json({ message: 'Both amount and status are required' });
342     }
343     const updatedPayment = await models.Payment.findByIdAndUpdate(
344       paymentId,
345       { amount, status },
346       { new: true, runValidators: true }
347     );
348     res.status(200).json({
349       message: 'Payment updated successfully',
350       payment: updatedPayment,
351     });
352   }
353 }
354 );
355
```

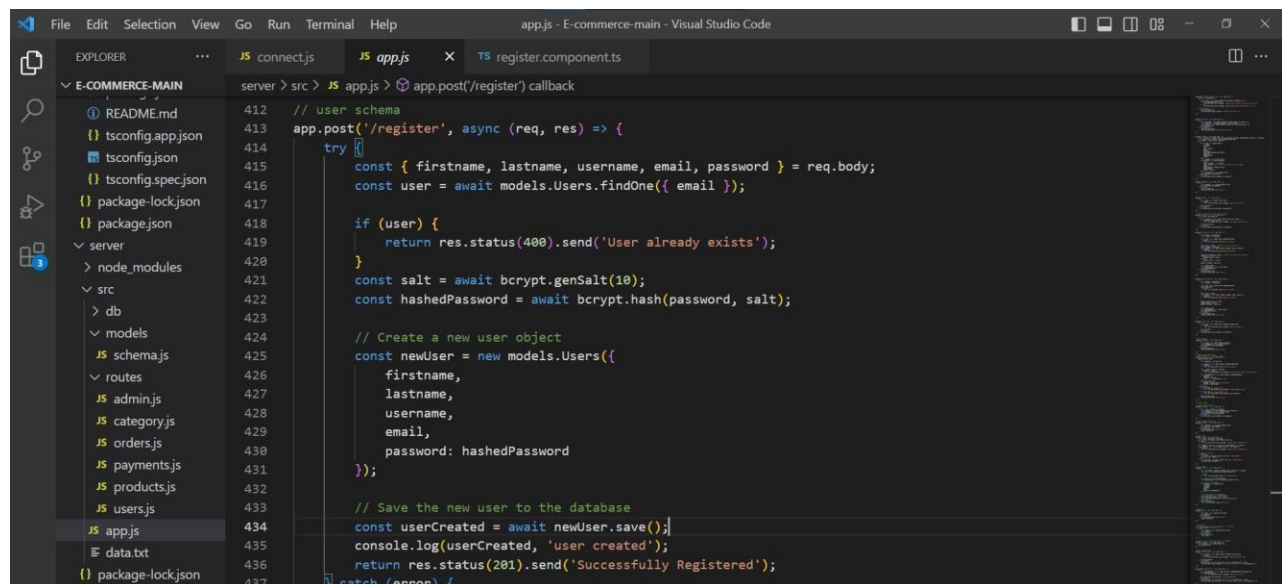
API's for feedback and login:

The Feedback API enables users to submit feedback or suggestions, while the Login API allows users to authenticate and access secured areas of an application or website.



This screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The main editor window shows the `app.js` file with the following code:

```
server > src > JS app.js > ...
366 // Create feedback from user
367 app.post('/feedback', async (req, res) => {
368   try {
369     const { user, message } = req.body;
370     const feedback = new models.Feedback({ user, message });
371     const savedFeedback = await feedback.save();
372     res.status(201).json(savedFeedback);
373   } catch (err) {
374     res.status(400).json({ message: err.message });
375   }
376 });
377
378 // Check feedback (admin only)
379 app.get('/feedback', async (req, res) => {
380   try {
381     const feedback = await models.Feedback.find();
382     res.status(200).send(feedback);
383   } catch (error) {
384     res.status(500).send('Server error');
385     console.log(error);
386   }
387 });
388
```

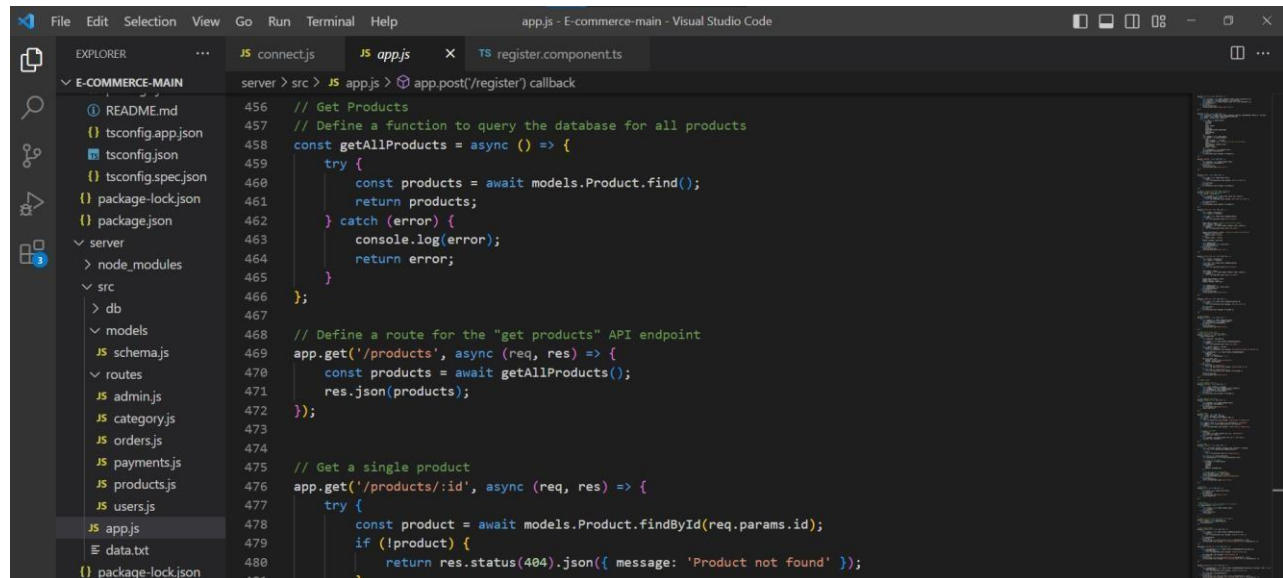


This screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure. The main editor window shows the `app.js` file with the following code:

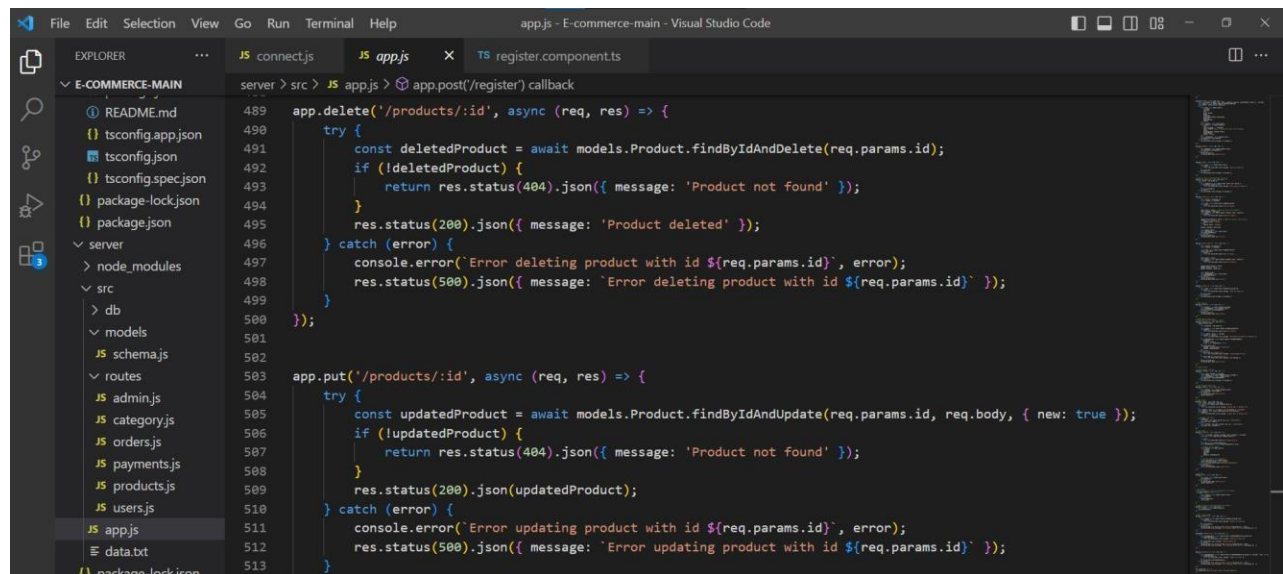
```
server > src > JS app.js > app.post('/register') callback
412 // user schema
413 app.post('/register', async (req, res) => {
414   try {
415     const { firstname, lastname, username, email, password } = req.body;
416     const user = await models.Users.findOne({ email });
417
418     if (user) {
419       return res.status(400).send('User already exists');
420     }
421
422     const salt = await bcrypt.genSalt(10);
423     const hashedPassword = await bcrypt.hash(password, salt);
424
425     // Create a new user object
426     const newUser = new models.Users({
427       firstname,
428       lastname,
429       username,
430       email,
431       password: hashedPassword
432     });
433
434     // Save the new user to the database
435     const userCreated = await newUser.save();
436     console.log(userCreated, 'user created');
437     return res.status(201).send('Successfully Registered');
438   } catch (error) {
439
440   }
441 }
442
```

API's for Products:

The Product API enables users to perform various operations related to products, such as creating, updating, deleting, and retrieving product information from a system or database.



```
456 // Get Products
457 // Define a function to query the database for all products
458 const getAllProducts = async () => {
459   try {
460     const products = await models.Product.find();
461     return products;
462   } catch (error) {
463     console.log(error);
464     return error;
465   }
466 };
467
468 // Define a route for the "get products" API endpoint
469 app.get('/products', async (req, res) => {
470   const products = await getAllProducts();
471   res.json(products);
472 });
473
474 // Get a single product
475 app.get('/products/:id', async (req, res) => {
476   try {
477     const product = await models.Product.findById(req.params.id);
478     if (!product) {
479       return res.status(404).json({ message: 'Product not found' });
480     }
481   }
```



```
489 app.delete('/products/:id', async (req, res) => {
490   try {
491     const deletedProduct = await models.Product.findByIdAndDelete(req.params.id);
492     if (!deletedProduct) {
493       return res.status(404).json({ message: 'Product not found' });
494     }
495     res.status(200).json({ message: 'Product deleted' });
496   } catch (error) {
497     console.error('Error deleting product with id ${req.params.id}', error);
498     res.status(500).json({ message: 'Error deleting product with id ${req.params.id}' });
499   }
500 });
501
502 app.put('/products/:id', async (req, res) => {
503   try {
504     const updatedProduct = await models.Product.findByIdAndUpdate(req.params.id, req.body, { new: true });
505     if (!updatedProduct) {
506       return res.status(404).json({ message: 'Product not found' });
507     }
508     res.status(200).json(updatedProduct);
509   } catch (error) {
510     console.error('Error updating product with id ${req.params.id}', error);
511     res.status(500).json({ message: 'Error updating product with id ${req.params.id}' });
512   }
513 }
```

END.

Frontend:

User Interface (UI) Design:

- Create a visually appealing and consistent design using modern design principles.
- Use a UI design tool like Adobe XD, Sketch, Figma, or InVision to create wireframes and mockups.
- Pay attention to typography, color schemes, spacing, and visual hierarchy.
- Use responsive design techniques to ensure the app looks great on different devices.

Responsive Design:

- Utilize CSS media queries and responsive design frameworks like Bootstrap or Tailwind CSS to create a responsive layout.
- Test your app on various devices and screen sizes to ensure a seamless user experience.

Product Catalog:

- Design and implement a product listing page that displays product images, titles, descriptions, prices, and other relevant details.
- Implement search functionality to allow users to find products easily.
- Include filters and sorting options to enhance the browsing experience.

Shopping Cart and Checkout Process:

- Design and develop a shopping cart component to allow users to add products, view cart contents, update quantities, and remove items.
- Create a checkout process with multiple steps, including shipping information, payment selection, and order review.

User Authentication and Account Management:

- Design and implement a user registration and login system.
- Create user profile pages where users can view and edit their personal information, addresses, payment methods, and order history.
- Implement authentication guards to restrict access to certain pages or features.

Payment Integration:

- Integrate with a payment gateway service like Stripe, PayPal, or Braintree.
- Implement a secure and seamless payment flow that allows users to enter payment details and complete transactions.

- Handle transaction success and failure scenarios and provide appropriate feedback to the user.

Accessibility:

- Follow the Web Content Accessibility Guidelines (WCAG) to ensure your app is accessible to users with disabilities.
- Use semantic HTML tags and proper ARIA attributes.
- Provide alternative text for images and captions for videos.
- Ensure keyboard navigation support and focus indicators.
- Test your app using accessibility evaluation tools and conduct manual testing with assistive technologies.