E-Commerce App

Introduction:

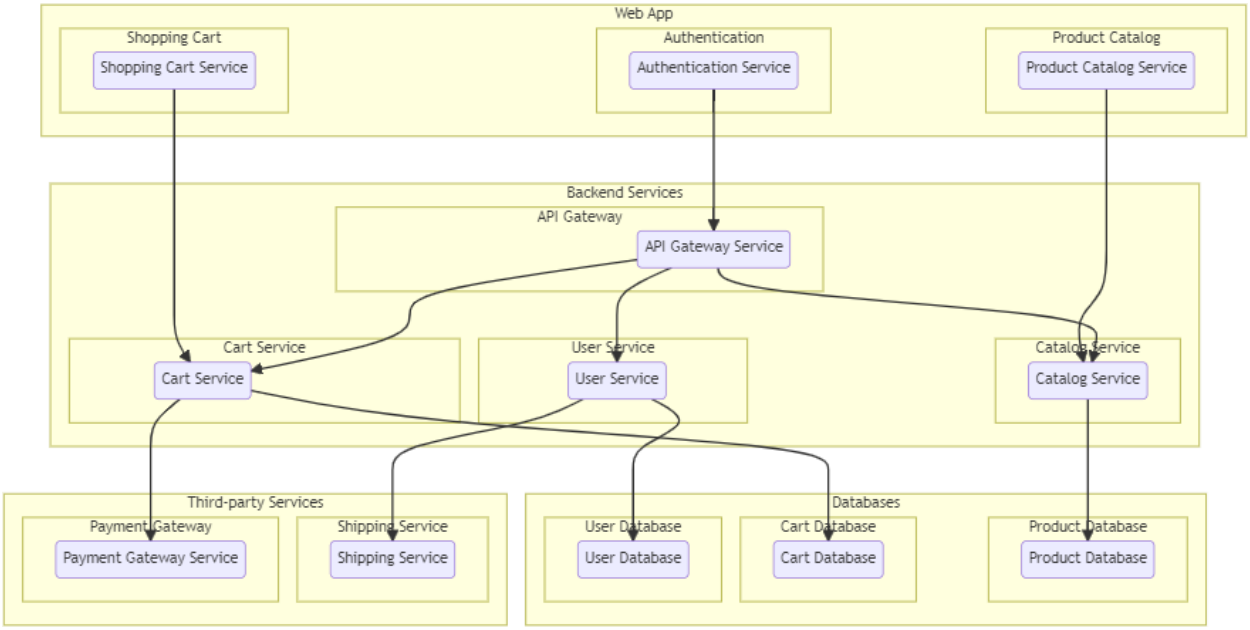
Welcome to our innovative e-commerce platform, where shopping becomes a seamless and delightful experience for everyone. Whether you're a tech enthusiast, a fashion-forward trendsetter, or a practical homemaker seeking everyday essentials, our app caters to all your needs. With a sleek and user-friendly 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.

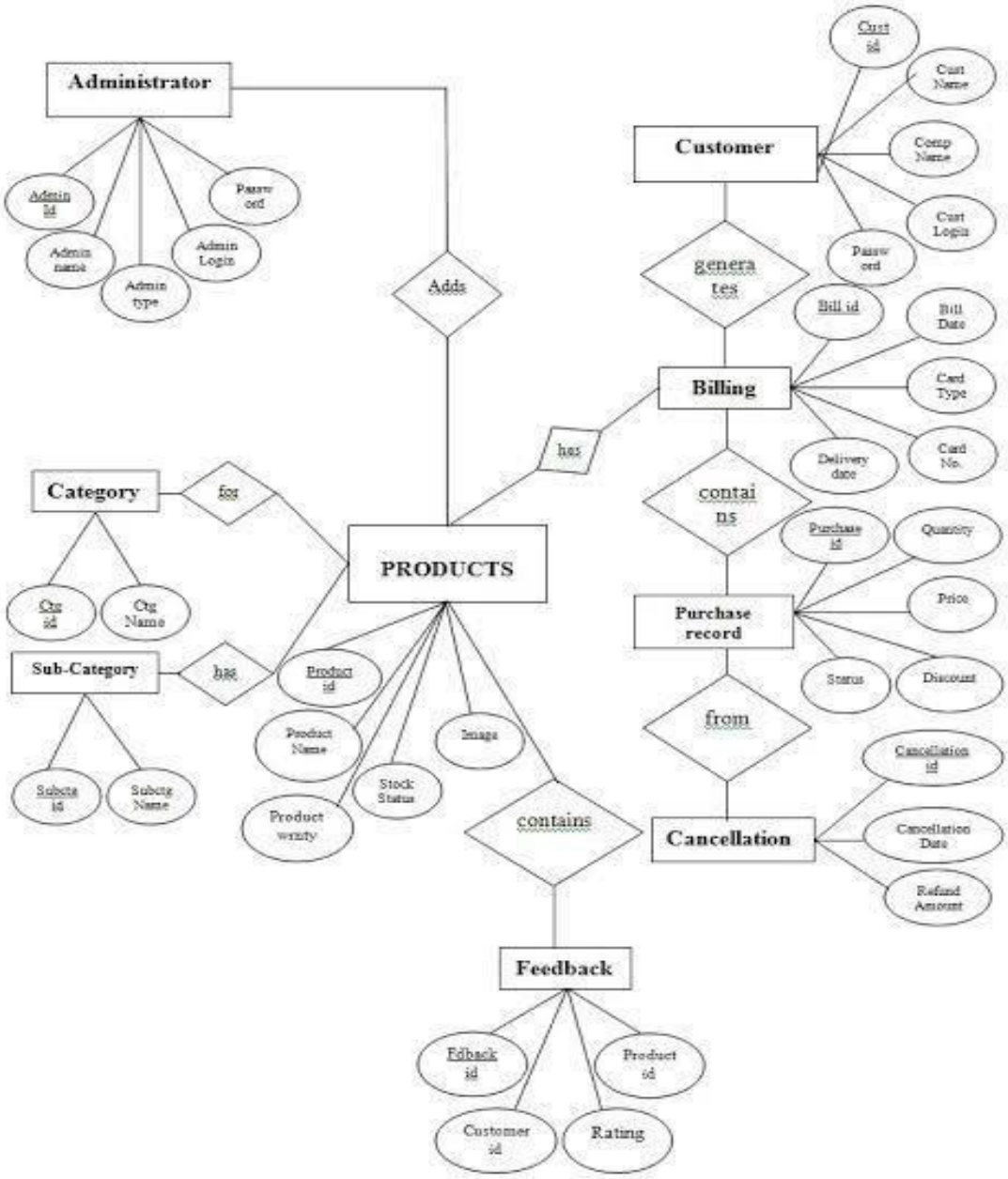
Thank you for joining us, and we look forward to providing you with an exceptional shopping experience. Happy shopping with our cutting-edge e-commerce app!

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 de pendencies

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 fol lowing 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 An- gular 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/Mahishkanna13/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-2I18Dl-7koG8/view?usp=drivesdk

Project Repository Link : https://github.com/Mahishkanna13/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:

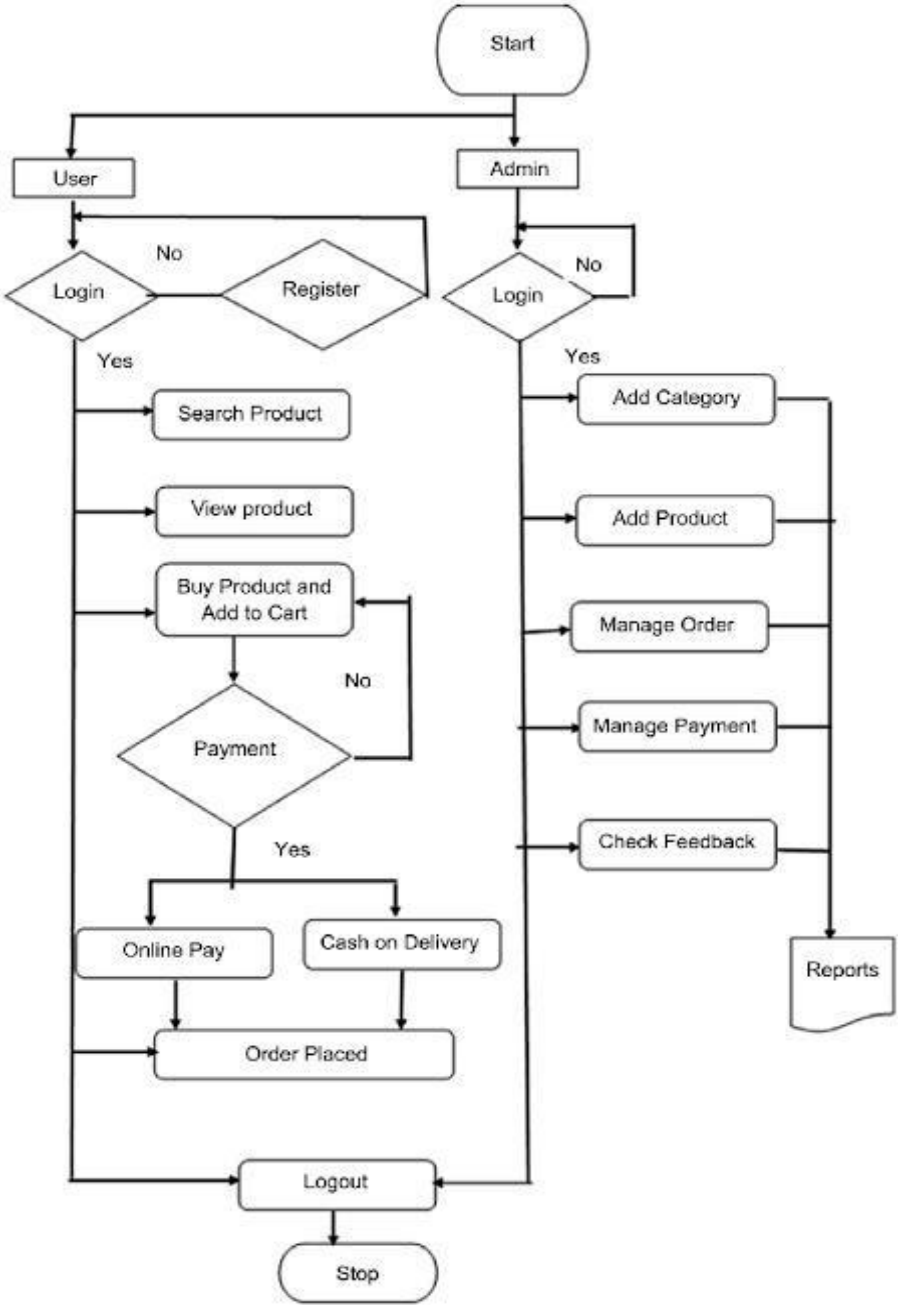
1. Create an account and log in to the system using their email and password.
2. Browse and search for products available on the platform.
3. View detailed product information, including description, price, and availability.
4. Add products to their cart for future purchase.
5. Proceed to checkout and 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.

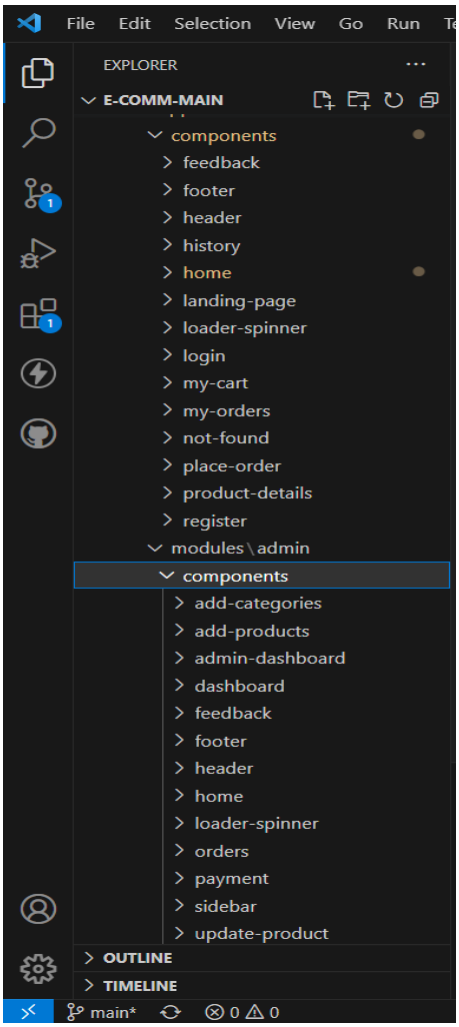
These roles and responsibilities are aimed at ensuring a smooth and efficient operation of the e-commerce app, providing a seamless experience for customers, sellers, and administrators.

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.

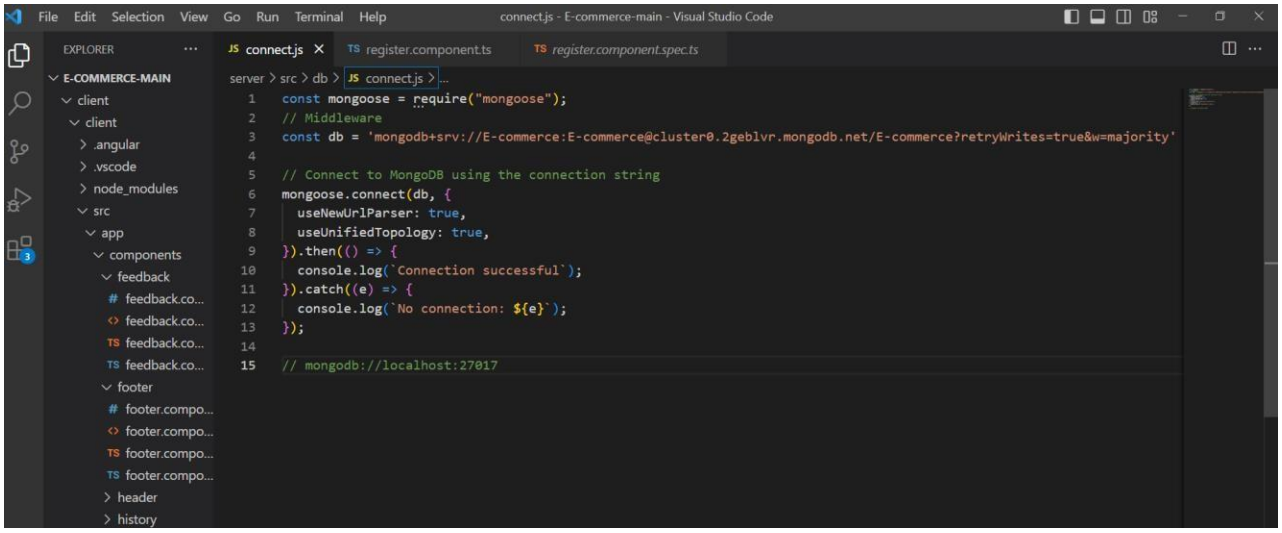
6. **Payment:**

- 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.

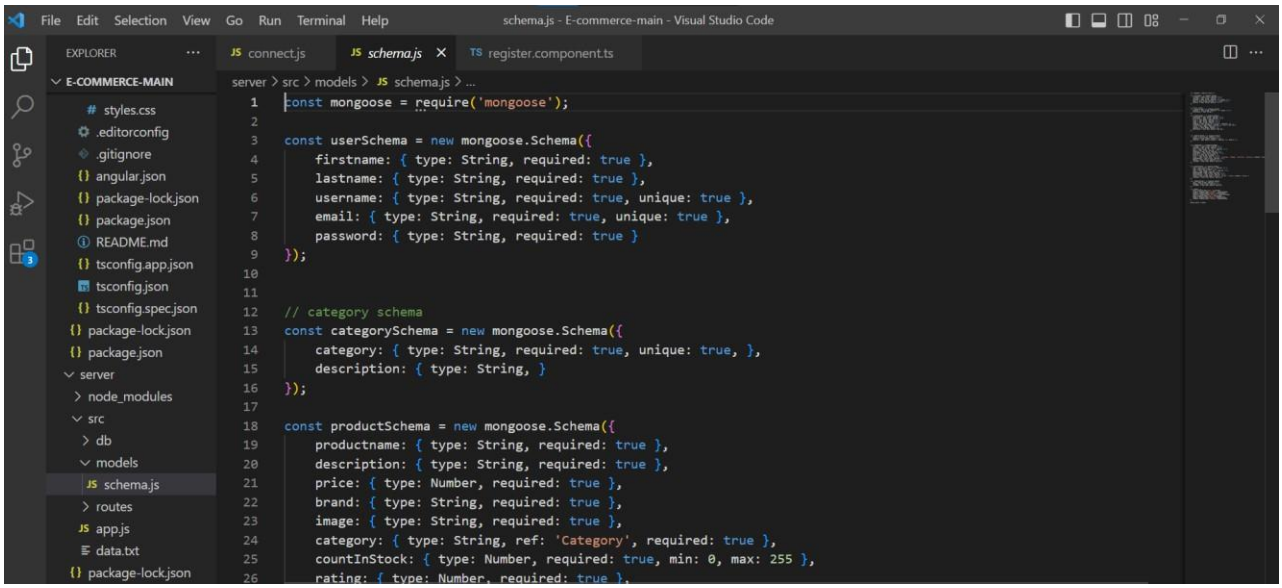
7. **Feedback:**

- 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:



Schemas:



schema.js - E-commerce-main - Visual Studio Code

server > src > models > schema.js > ...

```javascript
    dateCreated: { type: Date, default: Date.now }
});



const addToCartSchema = new mongoose.Schema({
    userId: { type: String, required: true },
    productId: { type: String, required: true },
    quantity: { type: Number, minimum: 1, required: true, default: 1 },
});


const orderSchema = new mongoose.Schema({
    firstname: { type: String, required: true },
    lastname: { type: String, required: true },
    user: { type: String, ref: 'User', required: true },
    phone: { type: String, required: true },
    productId: { type: String, required: true },
    productName: { type: String, required: true },
    quantity: { type: String, default: 1 },
    price: { type: String, required: true },
    status: { type: String, enum: ['Pending', 'Confirmed', 'Shipped', 'Delivered', 'Canceled',], default: 'Pending' }
    paymentMethod: { type: String, required: true },
    address: { type: String, required: true },
    createdAt: { type: Date, default: Date.now }
});
```

schema.js - E-commerce-main - Visual Studio Code

server > src > models > schema.js > ...

```javascript
    order: { type: String, ref: 'Order', required: true },
    amount: { type: Number, required: true },
    paymentMethod: { type: String, required: true },
    deliveryStatus: { type: String, required: true },
    status: { type: String, enum: ['Pending', 'Success', 'Failed'], default: 'Pending' },
    createdAt: { type: Date, default: Date.now }
});

const feedbackSchema = new mongoose.Schema({
    user: { type: String, required: true },
    message: { type: String, required: true },
    createdAt: { type: Date, default: Date.now }
});

const models = {
    Users: mongoose.model('User', userSchema),
    Category: mongoose.model('Category', categorySchema),
    Product: mongoose.model('Product', productSchema),
    AddToCart: mongoose.model('AddToCart', addToCartSchema),
    Order: mongoose.model('Order', orderSchema),
    Payment: mongoose.model('Payment', paymentSchema),
    Feedback: mongoose.model('Feedback', feedbackSchema)
};

module.exports = models;
```

API's for Add Product:

```
// Add a new product to the database and associate it with an existing category
app.post('/add-products', async (req, res) => {
    try {
        const { productname, description, price, brand, image, category, countInStock, rating, quantity } = req.body;

        if (!productname || !description || !price || !brand || !image || !category || !countInStock || !rating) {
            return res.status(400).send({ message: 'Missing required fields' });
        }
        const foundCategory = await models.Category.findOne({ category });
        if (!foundCategory) {
            return res.status(404).send({ message: 'Category not found' });
        }
        const product = new models.Product({
            productname,
            description,
            price,
            brand,
            image,
            category,
            countInStock,
            rating,
            quantity,
            dateCreated: new Date()
        });
```

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

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

```
// Endpoint for adding an item to the cart
app.post('/add-to-cart', async (req, res) => {
    const {userId, productId, productName, quantity = 1 } = req.body;
    const item = new models.AddToCart({userId, productId,productName, quantity });
    try {
        await item.save();
        res.status(200).json({ message: `Added ${quantity} of product ${productId} to cart` });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal server error' });
    }
});


app.delete('/remove-from-cart/:id', async (req, res) => {
    const id = req.params.id;
    try {
        const result = await models.AddToCart.deleteOne({ productId: id });
        if (result.deletedCount === 0) {
            res.status(404).json({ message: `Product with id ${id} not found in the cart` });
        } else {
            res.status(200).json({ message: `Removed product with id ${id} from cart` });
        }
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal server error' });
```

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 APIretrieves payment information for specific orders.

```
app.post('/orders', async (req, res) => {
    const { firstname, lastname, user, phone, productId, quantity, paymentMethod, address } = req.body;
    const product = await models.Product.findById(productId);
    const amount = product.price * quantity;
    try {
        const order = new models.Order({
            firstname,
            lastname,
            user,
            price: amount,
            phone,
            productId,
            productName:product.productname,
            quantity,
            paymentMethod,
            address
        });
        const newOrder = await order.save();
        const payment = new models.Payment({
            user,
            name:firstname+ " " +lastname,
            order: newOrder._id, // Associate the order with the payment
            amount,
            deliveryStatus: newOrder.status,
            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 APIretrieves orders specific to a user ID.

```
app.get('/orders', async (req, res) => {
    try {
        const order = await models.Order.find();
        if (!order) {
            return res.status(404).json({ message: 'Order not found' });
        }
        res.json(order);
    } catch (err) {
        res.status(500).json({ message: err.message });
    }
});

// Define a route for fetching orders by user ID
app.get('/my-orders/:id', async (req, res) => {
    const userId = req.params.id;
    try {
        const userOrders = await models.Order.find({ user: userId });
        if (userOrders.length === 0) {
            return res.status(404).json({ message: 'User orders not found' });
        }
        res.json(userOrders);
    } catch (err) {
        res.status(500).json({ message: err.message });
    }
```

API for Update order based on Id:

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

```
app.put('/orders/:id', async (req, res) => {
    try {
        const orderId = req.params.id;
        const { status } = req.body;

        const order = await models.Order.findById(orderId);
        if (!order) {
            return res.status(404).send('Order not found');
        }

        order.status = status; // Update the order status property
        order.createdAt = Date.now()
        const payment = await models.Payment.findOne({ order: orderId });
        if (!payment) {
            return res.status(404).send('Payment not found');
        }

        payment.deliveryStatus = status; // Update the payment status property
        if(status === 'Delivered'){
            payment.status = 'Success'
        }else{
            payment.status = "Pending"
        }
        payment.createdAt = Date.now()

        await payment.save();
```

API for cancel order based on Id:

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

```
app.put('/cancel-order/:id', async (req, res) => {
    try {
        const orderId = req.params.id;
        const { status } = req.body;

        const order = await models.Order.findById(orderId);
        console.log(order)
        if (!order) {
            return res.status(404).send('Order not found');
        }

        order.status = status;
        const payment = await models.Payment.findOne({ order: orderId });
        if (!payment) {
            return res.status(404).send('Payment not found');
        }

        payment.deliveryStatus = status;
        payment.status = "Failed"
        payment.createdAt = Date.now()


        await payment.save();
        const updatedOrder = await order.save();
        res.send(updatedOrder);
```

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.

```
app.get('/orders/:id', async (req, res) => {
    try {
        const order = await models.Order.findById(req.params.id);
        if (!order) {
            return res.status(404).json({ message: 'Order not found' });
        }
        res.json(order);
    } catch (err) {
        res.status(400).json({ message: err.message });
    }
});


// POST /payments
app.post('/payments', async (req, res) => {
    try {
        const payment = new models.Payment(req.body);
        const savedPayment = await payment.save();
        res.status(201).json(savedPayment);
    } catch (err) {
        console.error(err);
        res.status(500).send('Server Error');
    }
});
```

```
// Manage payment (admin only)
// Define the route for updating a payment
app.put('/payment/:id', async (req, res) => {
    console.log(req.body);
    try {
        const paymentId = req.params.id;

        const payment = await models.Payment.findById(paymentId);
        if (!payment) {
            return res.status(404).send('Payment not found');
        }
        const { amount, status } = req.body;
        if (!amount || !status) {
            return res.status(400).json({ message: 'Both amount and status are required' });
        }
        const updatedPayment = await models.Payment.findByIdAndUpdate(
            paymentId,
            { amount, status },
            { new: true, runValidators: true }
        );
        res.status(200).json({
            message: 'Payment updated successfully',
            payment: updatedPayment,
        });
```

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.

```
// Create feedback from user
app.post('/feedback', async (req, res) => {
    try {
        const { user, message } = req.body;
        const feedback = new models.Feedback({ user, message });
        const savedFeedback = await feedback.save();
        res.status(201).json(savedFeedback);
    } catch (err) {
        res.status(400).json({ message: err.message });
    }
});

// Check feedback (admin only)
app.get('/feedback', async (req, res) => {
    try {
        const feedback = await models.Feedback.find();
        res.status(200).send(feedback);
    } catch (error) {
        res.status(500).send('Server error');
        console.log(error);
    }
});
```

```
// user schema
app.post('/register', async (req, res) => {
    try {
        const { firstname, lastname, username, email, password } = req.body;
        const user = await models.Users.findOne({ email });

        if (user) {
            return res.status(400).send('User already exists');
        }
        const salt = await bcrypt.genSalt(10);
        const hashedPassword = await bcrypt.hash(password, salt);

        // Create a new user object
        const newUser = new models.Users({
            firstname,
            lastname,
            username,
            email,
            password: hashedPassword
        });

        // Save the new user to the database
        const userCreated = await newUser.save();
        console.log(userCreated, 'user created');
        return res.status(201).send('Successfully Registered');
    } catch (error) {
```

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.

```
// Get Products
// Define a function to query the database for all products
const getAllProducts = async () => {
    try {
        const products = await models.Product.find();
        return products;
    } catch (error) {
        console.log(error);
        return error;
    }
};

// Define a route for the "get products" API endpoint
app.get('/products', async (req, res) => {
    const products = await getAllProducts();
    res.json(products);
});


// Get a single product
app.get('/products/:id', async (req, res) => {
    try {
        const product = await models.Product.findById(req.params.id);
        if (!product) {
            return res.status(404).json({ message: 'Product not found' });
```

```
app.delete('/products/:id', async (req, res) => {
    try {
        const deletedProduct = await models.Product.findByIdAndDelete(req.params.id);
        if (!deletedProduct) {
            return res.status(404).json({ message: 'Product not found' });
        }
        res.status(200).json({ message: 'Product deleted' });
    } catch (error) {
        console.error(`Error deleting product with id ${req.params.id}`, error);
        res.status(500).json({ message: `Error deleting product with id ${req.params.id}` });
    }
});


app.put('/products/:id', async (req, res) => {
    try {
        const updatedProduct = await models.Product.findByIdAndUpdate(req.params.id, req.body, { new: true });
        if (!updatedProduct) {
            return res.status(404).json({ message: 'Product not found' });
        }
        res.status(200).json(updatedProduct);
    } catch (error) {
        console.error(`Error updating product with id ${req.params.id}`, error);
        res.status(500).json({ message: `Error updating product with id ${req.params.id}` });
    }
```

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.