RBNS – Ayurvedic Online Store

Project Report

Sri Lanka Institute of Information Technology

SE3020 – Distribted Systems

Y3.S1.WE.SE.01

IT21042324 - Reezan S.A

IT21078996 – Padmasiri P.G.S.M.

IT21102264 – Achchige R.A.N.R.

IT21104176 – Abeygunasekara P.W.A.B.N.

April 2023

Declaration

We confirm that the project report we have submitted is our own creation and does not involve any act of plagiarism from any other sources. We assure you that this report does not include any content that has been published or written by any third party, unless explicitly mentioned in the text.

Authors:

Author SID	Author name
IT21042324	Reezan S.A
IT21078996	Padmasiri P. G. S. M.
IT21102264	Achchige R.A.N.R.
IT21104176	Abeygunasekara P.W.A.B.N.

Date: 28/04/2023

Abstract

This report presents the development of a web-based microservices application created using the MERN (MongoDB, Express, React, Node.js) stack for the Distributed Systems module in Year 3 Semester 1. The application uses Docker and Kubernetes to connect and run its services efficiently. The backend has been divided into four main services: Product Service, User Service, Payment Service, and Product Service to enhance productivity. Robust security measures have been implemented to ensure the protection of the system from unauthorized access. These measures include a combination of authentication and authorization mechanisms and password encryption. All data is stored in a secure database that can only be accessed by authorized personnel. The system is accessible through the internet from anywhere at any time. The development process prioritized efficiency and reliability, resulting in a secure and efficient web-based application that provides an effective solution for customers.

Acknowledgment

The work described in this document was completed as our third year first-semester project for the module Distributed Systems 2023. As a group, we express our heartfelt gratitude to all those who provided guidance and necessary support to successfully complete our project task. Special thanks go to all the lecturers and instructors attached to the Distributed Systems (DS) module, particularly Dr. Dharshana Kasthurirathna and Ms. Hansi De Silva. Their advice and encouragement gave us the strength to begin and complete this project successfully.

We would also like to thank Mr. Sutharshan Shanmugarajah and Mr. Kasun Weerasekara, our lab instructors, for providing us with valuable information and support to complete our project. Their immense support helped us fulfill the requirements successfully. Finally, we express our gratitude to all the group members who gave their maximum effort and commitment to complete the project successfully. This website was the outcome of our hard work throughout this semester.

Table of Contents

Table Of Figures

1. Introduction

1.1. Background

We have been tasked with developing a collaborative shopping platform for Ayurvedic/Herbal medicines and supplements. The platform will serve as an online marketplace for customers to browse and purchase high-quality Ayurvedic and herbal products from a variety of trusted suppliers. We'll add comprehensive product descriptions, customer evaluations, and ratings to make sure buyers are happy with their purchases. We'll also provide a variety of payment methods and maintain the privacy of our clients' personal data. The platform should offer vendors a service to add, edit, and delete items as well as a web interface for buyers to browse the items supplied by sellers. Multiple-item purchases should be allowed, with an administrator personally validating and confirming orders. Buyers should have the opportunity to choose a delivery method and ask for a third-party delivery service (like DHL) after making a purchase. The platform should support credit card payments as well as those made through payment integration providers like PayPal. and send an email confirming the transaction. Customers must be able to check on the status of their orders and evaluate and rate sellers and products.

Our aim is to create a user-friendly and secure platform that provides customers with a hassle-free shopping experience. The website will provide a selection of goods at cheap costs, such as nutritional supplements, herbal cures, personal care items, and more.

1.1. Problems & Motivation

Traditional and alternative medicine, such as Ayurveda and herbal remedies, have become increasingly popular as people are turning to natural products for their healthcare needs. However, there is often a lack of reliable information, trusted suppliers, and convenient access to these products. Many people find it difficult to obtain high-quality Ayurvedic and herbal products due to limited availability in their area or a lack of awareness about which products to purchase.

The motivation behind developing a collaborative shopping platform for Ayurvedic/Herbal medicines and supplements is to provide a solution to these problems. The platform will serve as a centralized hub for customers to browse, purchase, and review a range of high-quality Ayurvedic and herbal products from trusted suppliers. By bringing together multiple suppliers, we aim to provide customers with a wide range of options at competitive prices, along with reliable product information, customer reviews, and ratings.

1.2. Structure of the report

The rest of the report will contain information on the implementation, design development, testing and evaluation process along with tools that were used to complete this project.

1.3. GitHub Link

https://github.com/IT21042324/DS-Assignment.git

1.4. Objectives

• Provide a user-friendly web interface where customers can easily browse and purchase Ayurvedic products.

• Provide a user-friendly seller dashboard where sellers can easily add, update, and delete Ayurvedic products.

• Provide a powerful search functionality for buyers to easily find Ayurvedic products based on various criteria such as product type, ingredients, price, and more.

• Allow buyers to purchase multiple items at once, and provide a streamlined checkout process for a hassle-free shopping experience.

• Ensure that orders are manually verified and confirmed by an administrator to prevent fraud and ensure the accuracy of orders.

• Provide a delivery service integration where buyers can select their preferred delivery option and track their order status from the seller to the final destination.

• Generate revenue from the platform by charging a commission on each sale, which includes the payment service fees.

• Provide secure payment options, such as credit card or Paypal/pay-here integration, with an easy-to-use payment gateway that securely processes transactions and protects customer data.

• Ensure that sensitive payment information, such as credit card number, CVC number, and cardholder name, is securely stored and transmitted using industry-standard encryption protocols.

• Provide a user-friendly interface for customers to track their order status, from order confirmation to delivery.

• Allow customers to rate and review Ayurvedic products and sellers to help other customers make informed purchasing decisions.

2. Solution Overview

2.1. User service

The user service handles user management and authentication within the Ayurvedic e-commerce platform. User registration and login are two of the user service's main functionalities. Customers need to their email address as a username and give the proper password and address, add a profile picture and contact number to create an account. these fields are required except add profile field. So, we could use some validations for email, password, and contact number. The user can’t register the already existing email address and the user should give the correct email pattern and password should be a minimum of 6 characters and the contact number should be 10 numbers and can't use letters and special characters. So, after the login user redirects to the product page. then the user can add products to the cart one at a time. Users can search for special Ayurvedic items and users can add some reviews as well. and From the account dashboard, they may modify their personal information and track orders. Merchants can also create and manage their accounts, enabling them to access a range of tools and features tailored to their business needs. Update their personal information, and change their passwords. The user service is essential in ensuring that user data is handled and maintained safely, which is an important priority for the Ayurvedic e-commerce platform. To protect user data and prevent unwanted access, the platform employs industry-standard security standards. Using Docker, the user service can be containerized into separate functionalities such as user authentication, authorization, and profile management, each packaged as a Docker image and deployed as a container. This allows for flexible scaling, where containers for specific functionalities can be scaled based on user demand. Kubernetes can be used to manage and scale the containers, ensuring high availability and security for the user service.

2.2. Product service

The product service manages the catalog of Ayurvedic products offered on the e-commerce platform. It includes features such as product listing, detailed product information, and product recommendations based on Ayurvedic principles. merchants can create, update, and manage their product listings, providing comprehensive information about the ingredients, benefits, and usage of each product. Customers can search, browse, and purchase Ayurvedic products based on their specific health needs and Ayurvedic doshas. To give clients access to high-quality Ayurvedic goods that are tailored to their specific health needs and Ayurvedic doshas, the product service for Ayurvedic products is essential. Using Docker, the product service can be containerized into separate functionalities such as product catalog management, product search, and inventory management, each packaged as a Docker image and deployed as a container. This allows for flexible scaling, where containers for specific functionalities can be scaled based on demand. Kubernetes can be used to manage and scale the containers, ensuring high availability and fault tolerance for the product service.

2.3. Store service

The store service provides an online storefront for merchants to showcase their products and brand. It includes features such as customizable templates, branding options, and online store management tools tailored for Ayurvedic businesses. This ensures that consumers will quickly know the brand and over time fosters loyalty and confidence in it. Customer can easily recognize their brand and they can get products from only that brand. Merchants can create their unique online store, customize the appearance and layout to reflect Ayurvedic principles, and manage various aspects such as product listings, orders, and customer interactions. The platform is intended to be versatile and user-friendly, with several features and tools to assist merchants in managing their online shop effectively. With Docker, the store service can be containerized into separate components such as web server, caching, and content delivery, each packaged as a Docker image and deployed as a container. This allows for horizontal scaling, where additional containers can be added or removed dynamically based on web traffic. Kubernetes can be used to manage and scale the containers, providing load balancing and automatic scaling capabilities for the store service.

2.4. Payment service

The payment service for Ayurvedic products is designed to provide a seamless and hassle-free payment experience to customers. With the increasing popularity of Ayurvedic products, it has become essential to have a payment service that is reliable, secure, and adheres to the principles of Ayurveda. The payment service offers various payment options such as credit cards, debit cards, and digital wallets. The payment service for Ayurvedic products also includes PayPal as a payment option. Customers can select the kind of payment that best satisfies their requirements and preferences. The payment service makes sure that every transaction is handled carefully and securely. To improve the entire payment experience, the payment service also has features like order tracking and secure payment information storage. With Docker, the payment service can be containerized into separate components such as payment gateways, transaction processors, and databases, each packaged as a Docker image and deployed as a container. This allows for horizontal scaling, where additional containers can be added or removed dynamically based on the payment processing load. Kubernetes can be used to manage and scale the containers across a cluster of Docker nodes, ensuring high availability and fault tolerance for the payment service.

3. Individual Contributions

3.1. Reezan S.A – IT21042324

User-Service Backend

In the backend, the user-service service was developed by me. Typically the users have been divided into three categories such as buyer, seller and admin. This service provides services such as user login and signup for all three of the users. Additionally, it also provides services for user management as well such as changing the user beign able to change his information and details.

Most important aspect that was developed in this service is the authentication and authorization that is provided to the system. The system routes have been protected based on the availability of a valid token by the user. If the token is accepted, then the system allows the user to access the services in the system if not he will be shown the authorization error. This feature was achieved by implementing the JSON Web token package in the service.

Email Service

I integrated the emailjs library to send emails to users based on signup, payments, and order confirmations. Emailjs is a third-party library that provides a simple API for sending emails without the need for setting up and configuring an email server.

Payment gateway Integration

I implemented payment gateway integration for both PayPal and credit card payment services. The implementation involved integrating PayPal and Stripe payment gateways into the front-end React application. Users can now securely make payments for their orders using either PayPal or credit card, which enhances the overall user experience of the application.

Kubernetes and Docker

In the development process, I was responsible for creating the Dockerfile for the user-service and other services. The Docker images were then pushed to Docker Hub and used in the Kubernetes deployment file. I was also responsible for writing the Kubernetes deployment file that deploys the services as separate containers and makes them scalable and highly available. Additionally, I also configured the Kubernetes service file for load balancing and managing traffic to the deployed services.

Frontend Contributions

In terms of my contributions to the front-end development, I was responsible for implementing the user authentication and authorization workflow using React components. This involved creating login and signup forms, along with the necessary validation and error handling. I ensured that the JSON Web Tokens received from the backend were stored in the browser's local storage for persistence. Additionally, I created the user profile page that displays the user's information and allows them to update it.

To ensure a seamless user experience, I used the React state management library, Redux, to manage states in the frontend. By leveraging Redux, I was able to ensure that the UI was consistent and didn't load every time a user switches from one section to another. This was achieved by providing a central store that holds the state of the application, which can be accessed and updated from any component in the application.

In addition, I also handled the mapping of items on the home page using Array.map functions, handling of making payments and orders placed by customers. I implemented the ability for customers to enter reviews for products as well as for sellers, with the restriction that customers could only add reviews once an order had been delivered. I also ensured that sellers were able to view, add, and update their products, while being able to manage their orders. Finally, I added the functionalities for admin that allowed him to view all the users in the system and accept orders.

I also added the functionality for the user to track the status of an order.

3.2. P. G. S. M. – IT21078996

Product-Service Backend

In the backend, the product service was developed by me. Typically the main users of this service are buyers and sellers. This service provides services such as creating new products and updating and deleting products for sellers to manage their stores. Also, this service has reviewing options provided. Buyers can review the service provided by the seller after a delivery is done. Buyers can post a review for a particular item they bought on the products page. Additionally, this service provides features to buyers to search items or stores using the search bar on the product page.

Frontend Contributions

In terms of my contributions to front-end development, I was responsible for implementing the product service interfaces using React components. This involved creating add-product forms and pop-up modals, along with the necessary validation and error handling. I ensured that the JSON Web Tokens received from the backend were stored in the browser's local storage for persistence. Additionally, I created a seller corner page that displays the product's information and allows them to update and delete it. To ensure a satisfactory experience, I used the React state management library, Redux, to manage states in the frontend. I have created the slider on the home page using Bootstrap components.

3.3. Achchige R.A.N.R-IT21102264

Store-Service Backend

For stores, the code exports a function to create a new store in the database. The code uses the mongoose to define the order and store models and interact with the MongoDB database it also uses the express.js framework to handle http request and response.

The store context which manages the state of items in a store. The ‘reducer’ function defines the actions that can modify the state such as adding, modifying, or deleting items. Each action is dispatched with an object that contains a type and a payload. The seller Order context manages the state orders and dashboard details for a seller. The 'reducer ' function define two actions adding order and dispatching order. The 'value' prop of the context provider is an object that contains the current state and the dispatch function the 'user Reducer' hook.

Frontend Contributions

So frontend I created the admin dashboard and store service functionality. So the store service functionality basically manage the stores and some order parts. So in here first seller have to click the become a seller button and redirect to the registration form. After give credentials. If account is successfully created, alert message will display and after redirect to the enter store details page and Seller can enter the store details and submit that. After the successfully submitted. Seller redirect to the seller corner dashboard. So the dashboard have the product list, how many orders are there for now?, Revenue, how many products are there? and seller can dispatch the order after click the dispatch order button display the alert message 'dispatch the order'. So this is front end part that I done.

3.4. Abeygunasekara P.W.A.B.N-IT21104176

Payment Service Backend

In our project I implemented the payment service, The customer who buy our products he or she wants to make a payment, so that all payment details are stored in here. And I input some various payment methods such as credit cards, debit cards, and PayPal also. This is very important service because we need more safety for this. In addition to providing a secure payment platform, the payment service also offers order tracking features. Customers can track the status of their orders and receive real-time updates on their delivery status. This helps to enhance the overall shopping experience and provides customers with peace of mind knowing that their orders are being processed and delivered on time. And all calculation parts are done in this service. And handles calculating the online store profit also.

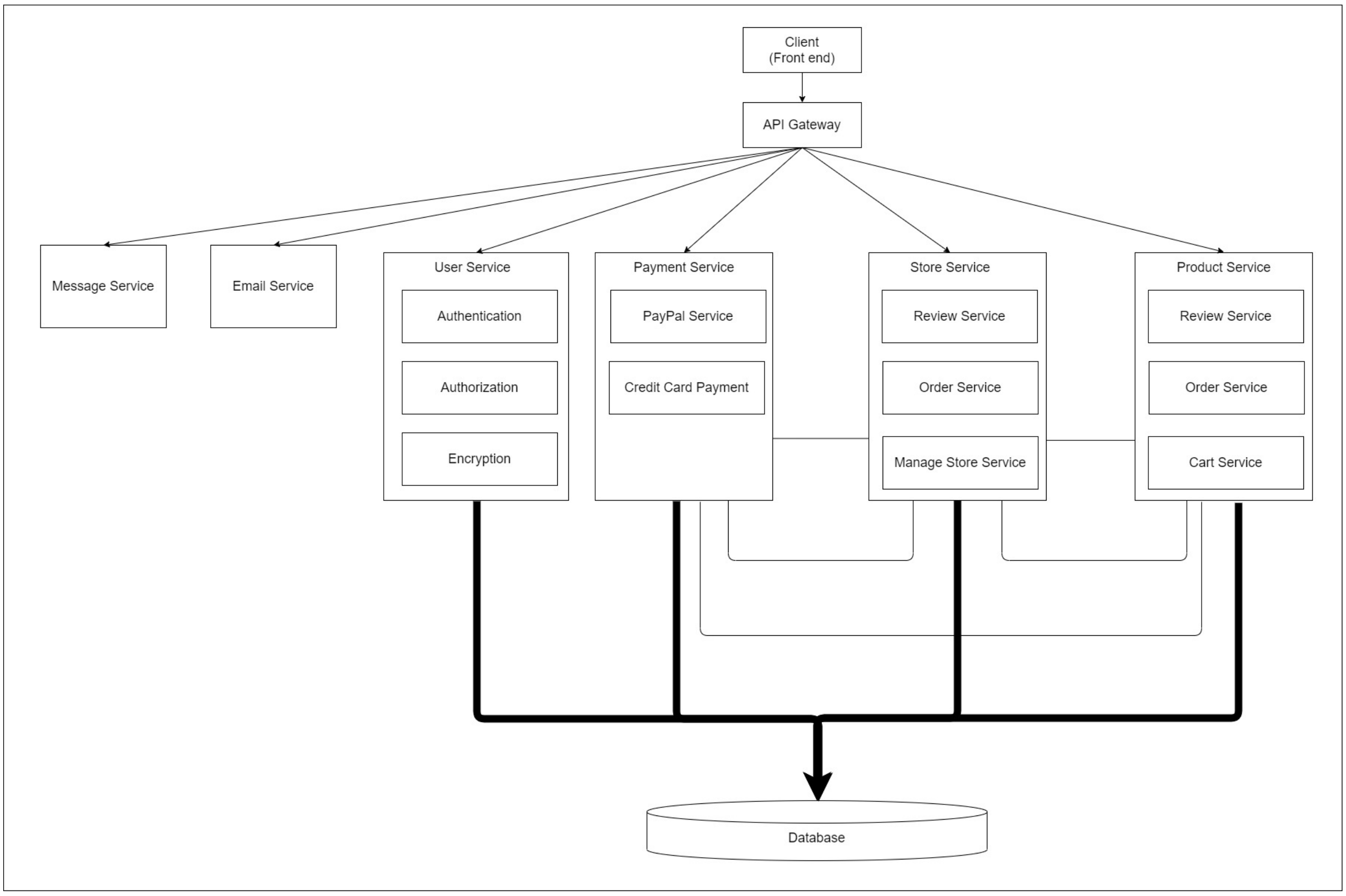
Frontend Contributions

As part of my contribution to front-end development I was responsible for implementing the cart functionality. I used some functionals to implement it. When customer add a product to cart it will be stored in the browser's local storage for persistence. And it will be deleted from that storage when customer buy it or cleared the cart. Also, I created the calculation function on this page according to the products taken by customers. and generate the total bill. By leveraging Redux, I was able to ensure that the UI was consistent and more responsive.

In addition, Customer can update, delete their products from the cart and also, they can make payment through our payment methods. Finally, admin can see all payments and all products who buy from our store, and he can check our products store also.

4. Design

4.1. High Level Architectural Diagram



4.1.Overview of architecture

4.2. Interface Explanations

Microservices architectural style has been implemented to achieve the requirements of this project. The backend of our system has been separated into four sections such as, User-service, payment-service, product-service, and store-service.

- User-service – The user-service is responsible for managing user authentication and authorization for all three types of users: buyers, sellers, and the admin. It exposes interfaces for users to log in and sign up. User authentication is handled using JSON Web Tokens (JWT), which are generated upon successful login and used to authorize subsequent requests. Passwords are encrypted and decrypted using BcryptJS with an appropriate salt value to ensure their security. The service also implements necessary secret keys for authentication and authorization purposes. Additionally, this service protects routes of other services by checking for a valid token on every request made by the user. The user service also integrates with an email service to send confirmation emails upon successful user registration.

- Product-Service – The product service is responsible for managing the products in the platform, including creating new products, adding reviews, and updating and deleting products or reviews. This service provides service for all three types of users: buyers, sellers, and the admin. It exposes interfaces for buyers to browse products and search for products. While sellers can manage products from the seller dashboard.

- Payment-Service – The payment service is responsible for managing all payments in stores and customers. It exposes interfaces for customer to buy their products and make payment. Customer can also make payment methods like Credit Card, Debit Card, and PayPal. And there are some features such as order tracking and secure storage of payment information. Customers can track the status of their orders and receive real-time updates on their delivery status.

- Store-Service – The store service is responsible for managing the store items and orders for the platform. It exposes interfaces for sellers to add store items, and for buyers to place orders for those items. So the seller can change the order status. The service implements an authentication middleware using JSON Web Tokens (JWT), which is used to verify the identity of the user making the request. The service also implements necessary secret keys for authentication and authorization purposes. The store service is designed to work with the user service to ensure the security and privacy of user data.

4.3. Justification of the usage of Docker and Kubernetes

The decision to use Docker and Kubernetes in this project was made with scalability and performance in mind. Each of the backend services (user, store, payment, and product) was developed as a separate Docker image. This approach allowed us to isolate each service and ensure its compatibility with the target environment, without worrying about system dependencies or other conflicts that might occur.

Moreover, Docker containers allow us to deploy these services consistently across different environments, whether it's our local development machine or a production server. This consistency eliminates potential issues with service dependencies, operating system compatibility, and other configuration problems that may arise during deployment.

In addition, Kubernetes was used to orchestrate these Docker containers and provide load balancing across them. By defining the Kubernetes yaml file, we were able to ensure that each service was deployed to multiple instances for fault tolerance and scalability. Kubernetes automatically detects and replaces failed containers, ensuring that the system is always available to handle incoming requests.

By utilizing Kubernetes, we also gained automatic scaling capabilities. With the increase in the number of incoming requests, Kubernetes will automatically scale the backend services by creating new containers and distributing the workload across them. This approach ensures that the system can handle sudden spikes in traffic without any downtime.

Finally, this approach allowed us to separate the front-end and backend applications, making them independently deployable and scalable. The frontend application, built with React, was served separately, and the backend services were orchestrated by Kubernetes. This separation of concerns made it easier to develop, test, and deploy the application as a whole.

In conclusion, the usage of Docker and Kubernetes in this project provided a more robust, scalable, and maintainable infrastructure that allowed us to deliver a high-performance and fault-tolerant application. By separating the frontend and backend applications and leveraging the containerization and orchestration capabilities of Docker and Kubernetes, we achieved better scalability, consistency, and flexibility in our development and deployment processes.

4.4. Authentication and authorization mechanism used.

To protect sensitive data and ensure secure access to our store service, we rely on the use of JSON web tokens (JWTs) for authentication and authorization. These tokens are generated when a user signs up or logs in and are used to validate the identity and access privileges of the user.

Whenever a store route is called, a middleware function is triggered on the backend of the store service to handle the incoming request. This middleware function acts as a gatekeeper and first receives the request before validating the associated JWT. The JWT is decoded and checked for its validity, ensuring that it hasn't been tampered with or expired. If the token is valid, the middleware function allows the request to proceed to the intended resource. However, if the token is not valid, the middleware function will throw an error with the message "User is not authenticated", preventing unauthorized access to the store's resources.

By using this approach, we can ensure that only authorized users can access our store's resources and prevent unauthorized access. Additionally, by implementing JWTs, we can avoid the need for frequent re-authentication and reduce the load on our authentication servers, resulting in a more efficient and secure system.

4.5. Development Tools and Technologies

- Frontend – ReactJS, Bootstrap, CSS
- Backend – Express JS, Node JS, MongoDB, Postman (for testing)
- Tools – GitHub, VSCode, Kanban Flow, Docker Desktop

4.6. Testing Methods

Throughout the development process, our team employed several testing methods to ensure the quality and reliability of our project.

During Development

1. **Unit testing** – We conducted unit testing, which involved each member of our team using a set of predefined test cases to test their respective components. This enabled us to identify and resolve any issues early on, simplifying the debugging process in later stages.

2. **Integration testing** – Following the completion of unit testing, we carried out integration testing. This was done by each member of our team to ensure that the various services developed by our team worked seamlessly together.
3. **System testing -** System testing was also performed to ensure that the entire system functioned as a cohesive unit. This testing method allowed us to evaluate the integrated system against the predetermined requirements.

After Development

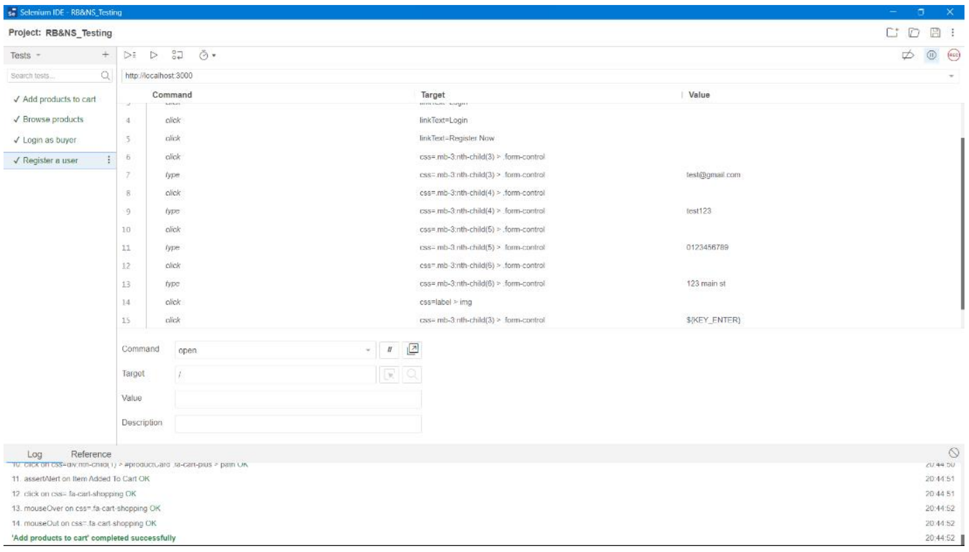
Usability Testing – After the development phase, we conducted usability testing to ensure that our microservices were user-friendly and easy to use. This testing approach helped us ensure that our services were accessible and intuitive for users.

Performance testing – We also conducted performance testing to assess the response of our microservices to different workloads. This testing was crucial for our microservices architecture project, as the system is designed to handle various tasks and user requests.

Security testing – Finally, we carried out security testing to ensure that our microservices were secure and protected user data. We tested concepts such as integrity, confidentiality, and availability to verify that our microservices met the required security standards based on the authentication, authorization and password encryption mechanisms developed for this system.

5. Testing using Selinium IDE

This test is done using the selinium ide for automated testing. In here we have tested four test case add products to cart, browse products, login as buyer and registration a user.

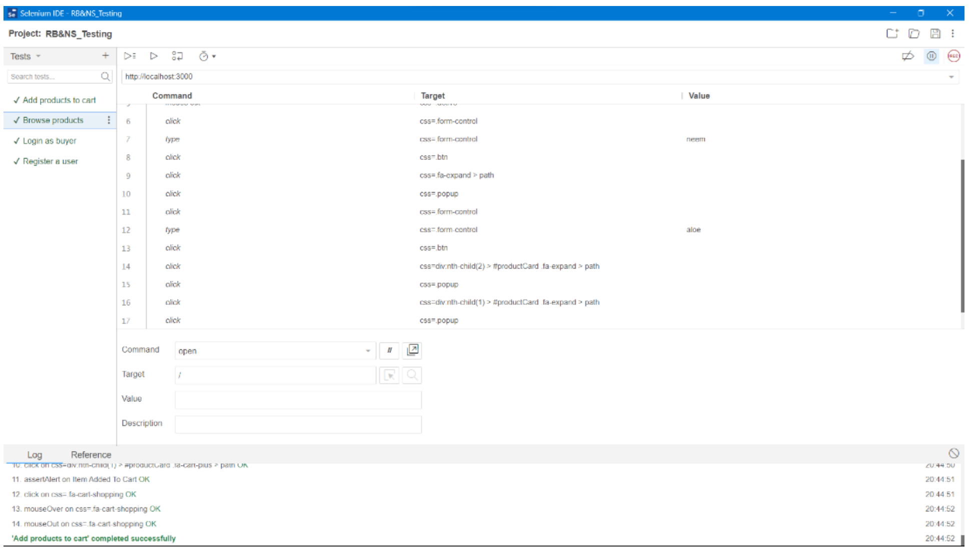


✓ Add products to cart
✓ Browse products
✓ Login as buyer
✓ Register a user

	Command	Target	Value
4	click	linkText=Login	
5	click	linkText=Register Now	
6	click	css=.mb-3:nth-child(3) > .form-control	
7	type	css=.mb-3:nth-child(3) > .form-control	test@gmail.com
8	click	css=.mb-3:nth-child(4) > .form-control	
9	type	css=.mb-3:nth-child(4) > .form-control	test123
10	click	css=.mb-3:nth-child(5) > .form-control	
11	type	css=.mb-3:nth-child(5) > .form-control	0123456789
12	click	css=.mb-3:nth-child(6) > .form-control	
13	type	css=.mb-3:nth-child(6) > .form-control	123 main st
14	click	css=label > img	
15	click	css=.mb-3:nth-child(3) > .form-control	${KEY_ENTER}

Command: open
Target: /
Value:
Description:

```
11. assertAlert on Item Added To Cart OK                              20:44:51
12. click on css=.fa-cart-shopping OK                                 20:44:51
13. mouseOver on css=.fa-cart-shopping OK                             20:44:52
14. mouseOut on css=.fa-cart-shopping OK                              20:44:52
'Add products to cart' completed successfully                        20:44:52
```



Selenium IDE - RB&NS_Testing

Project: RB&NS_Testing

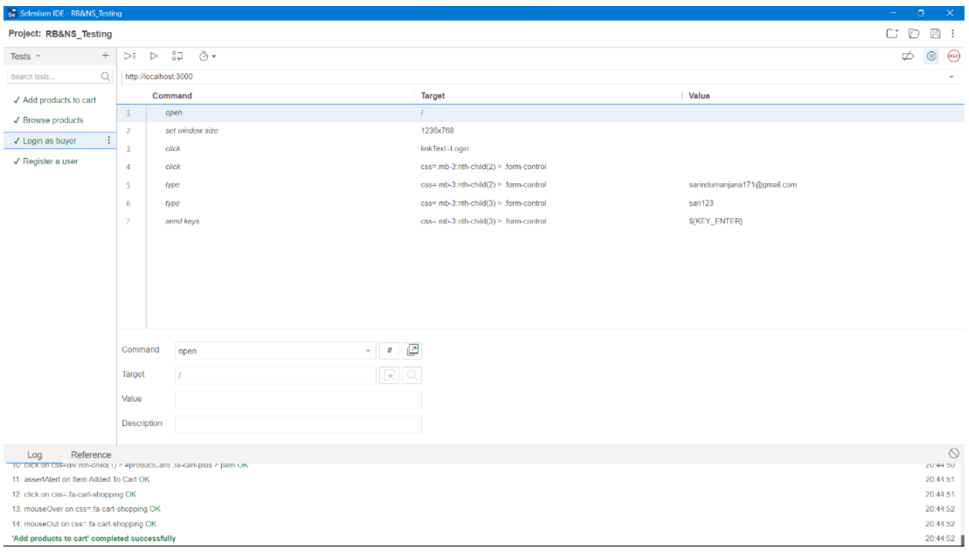
Tests: ✓ Add products to cart · ✓ Browse products · ✓ Login as buyer · ✓ Register a user

http://localhost:3000

	Command	Target	Value
6	click	css=.form-control	
7	type	css=.form-control	neem
8	click	css=.btn	
9	click	css=.fa-expand > path	
10	click	css=.popup	
11	click	css=.form-control	
12	type	css=.form-control	aloe
13	click	css=.btn	
14	click	css=div:nth-child(2) > #productCard .fa-expand > path	
15	click	css=.popup	
16	click	css=div:nth-child(1) > #productCard .fa-expand > path	
17	click	css=.popup	

Command: open
Target: /
Value:
Description:

```
10. click on css=div:nth-child(1) > #productCard .fa-cart-plus > path OK     20:44:50
11. assertAlert on Item Added To Cart OK                                     20:44:51
12. click on css=.fa-cart-shopping OK                                        20:44:51
13. mouseOver on css=.fa-cart-shopping OK                                    20:44:52
14. mouseOut on css=.fa-cart-shopping OK                                     20:44:52
'Add products to cart' completed successfully                                20:44:52
```



Selenium IDE - RB&NS_Testing

Project: RB&NS_Testing

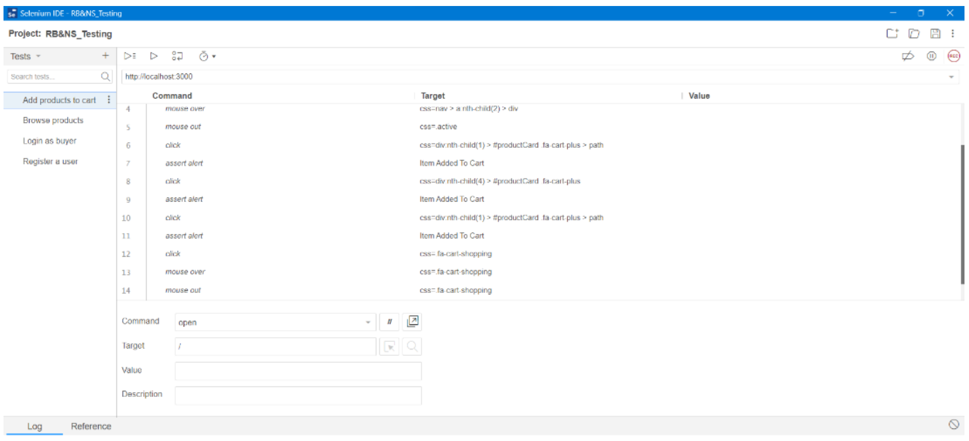
Tests: ✓ Add products to cart · ✓ Browse products · ✓ Login as buyer · ✓ Register a user

http://localhost:3000

	Command	Target	Value
1	open	/	
2	set window size	1236x768	
3	click	linkText=Login	
4	click	css=.mb-3:nth-child(2) > .form-control	
5	type	css=.mb-3:nth-child(2) > .form-control	sarindumanjana171@gmail.com
6	type	css=.mb-3:nth-child(3) > .form-control	sari123
7	send keys	css=.mb-3:nth-child(3) > .form-control	${KEY_ENTER}

Command: open
Target: /
Value:
Description:

```
10. click on css=div:nth-child(1) > #productCard .fa-cart-plus > path OK     20:44:50
11. assertAlert on Item Added To Cart OK                                     20:44:51
12. click on css=.fa-cart-shopping OK                                        20:44:51
13. mouseOver on css=.fa-cart-shopping OK                                    20:44:52
14. mouseOut on css=.fa-cart-shopping OK                                     20:44:52
'Add products to cart' completed successfully                                20:44:52
```



Selenium IDE - RB&NS_Testing

Project: RB&NS_Testing

Tests: Add products to cart · Browse products · Login as buyer · Register a user

http://localhost:3000

	Command	Target	Value
4	mouse over	css=nav > a:nth-child(2) > div	
5	mouse out	css=.active	
6	click	css=div:nth-child(1) > #productCard .fa-cart-plus > path	
7	assert alert	Item Added To Cart	
8	click	css=div:nth-child(4) > #productCard .fa-cart-plus	
9	assert alert	Item Added To Cart	
10	click	css=div:nth-child(1) > #productCard .fa-cart-plus > path	
11	assert alert	Item Added To Cart	
12	click	css=.fa-cart-shopping	
13	mouse over	css=.fa-cart-shopping	
14	mouse out	css=.fa-cart-shopping	

Command: open
Target: /
Value:
Description:

Appendix

REST API

User Service

User.js

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
const validator = require("validator");

const bcrypt = require("bcryptjs");

const userSchema = new Schema({
  userName: {
    type: String,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
  contact: {
    type: String,
    required: true,
  },
  address: {
    type: String,
    required: true,
  },
  image: {
    type: String,
  },
  role: {
    type: String,
    required: true,
  },
  storeID: String,
});

//Creating User schema functions
userSchema.statics.signup = async function (
  userName,
  password,
  contact,
  address,
  image,
  role
) {
  const exist = await this.find({ userName });

  if (!userName || !password || !contact || !address)
    throw Error("Please fill all fields");
  if (!validator.isEmail(userName)) throw Error("Email is invalid");
  if (exist.length === 1)
    if (exist[0].role == role) throw Error("Email is already in use");

  if (exist.length > 1) throw Error("Email is already in use");

  const salt = await bcrypt.genSalt(10);
  const hash = await bcrypt.hash(password, salt);

  const singedUser = await this.create({
    userName,
    password: hash,
    contact,
    address,
    image,
    role,
  });

  return singedUser; //To return a signedup new user object
};

userSchema.statics.login = async function (userName, password, role) {
  if (!userName || !password) throw Error("Please fill all fields");

  const user = await this.findOne({ userName, role });
  if (!user) throw Error("User Name doesn't exist");

  const match = await bcrypt.compare(password, user.password); //returns true
or false

  if (!match) throw Error("Incorrect Password");

  return user;
};

module.exports = mongoose.model("User", userSchema);
```

userController.js

```
const userModel = require("../models/User");
const jwt = require("jsonwebtoken");

//To generate a token
const createToken = (id) => {
  return jwt.sign({ id }, process.env.SECRET, { expiresIn: "3d" });
  //1st argument->object for payload
  //2nd argument-> secret string only know for our server (.env file)
  //3rd argument
};
const userLogin = async (req, res) => {
  try {
    // Get userName, password, and role from request body
    const { userName, password, role } = req.body;

    // Authenticate user using userModel's login method
    const user = await userModel.login(userName, password, role);

    // Create JWT for authenticated user
    const token = createToken(user._id);

    // Send JWT and user data in response
    res.json({ ...user.toObject(), token });
  } catch (err) {
    console.log(err.message);
    res.json({ err: err.message });
  }
};

const userSignUp = async function (req, res) {
  // Get user details from request body
  const { userName, password, contact, address, role, image } = req.body;

  try {
    // Create new user using userModel's signup method
    const user = await userModel.signup(
      userName,
      password,
      contact,
      address,
      image,
      role
    );

    // Create JWT for new user
    const token = createToken(user._id);

    // Send JWT and user data in response
    res.json({ ...user.toObject(), token });
  } catch (err) {
    console.log(err.message);
    res.json({ err: err.message });
  }
};

const getAllUsers = async function (req, res) {
  try {
    // Get all users from MongoDB database using Mongoose
    const users = await userModel.find();

    // Send users and user count in response
    res.json({ users, userCount: users.length });
  } catch (err) {
    res.send(err.message);
  }
};

const updateUser = async function (req, res) {
  // Get userId, userName, and image from request body
  const { userId, userName, image } = req.body;

  try {
    // Update user in MongoDB database using Mongoose
    const user = await userModel.findOneAndUpdate(
      { _id: userId },
      { userName, image },
      { new: true }
    );

    // Send updated user data in response
    return res.json(user);
  } catch (err) {
    console.log(err.message);
  }
};

const deleteUser = async (req, res) => {
  try {
    // Delete user from MongoDB database using Mongoose
    const data = await userModel.findByIdAndDelete(req.params.id);

    console.log(data);
    res.json(data);
  } catch (err) {
    console.log(err.message);
    res.send(err.message);
  }
};

const getOneUser = async function (req, res) {
  // Get id and role from request params
  const { id, role } = req.params;

  try {
    // Get user from MongoDB database using Mongoose
    const user = await userModel.find({ _id: id, role });

    // Send user data in response
    res.status(200).json(user);
  } catch (err) {
    console.log(err.message);
  }
};

const updateUserStore = async (req, res) => {
  // Get userID and storeID from request body
  const { userID, storeID } = req.body;

  try {
    // Update user's store in MongoDB database using Mongoose
    const updatedUser = await userModel.findOneAndUpdate(
      { _id: userID },
      { storeID }
    );

    console.log(updateUser);

    // Send updated user data in response
    res.json(updatedUser);
  } catch (err) {
    console.log(err);
    res.json(err);
  }
};

const getUserCount = async (req, res) => {
  try {
    // Get all users from MongoDB database using Mongoose
    const data = await userModel.find();
    // Send user count in response
    res.json({ userCount: data.length });
  } catch (err) {
    res.send(err.message);
  }
};

// Export functions for use in other files
module.exports = {
  userSignUp,
  userLogin,
  getAllUsers,
  updateUser,
  deleteUser,
  getOneUser,
  updateUserStore,
  getUserCount,
};
```

user.js

```
const router = require("express").Router();

// Import controller functions
const {
  userLogin,
  userSignUp,
  updateUser,
  getOneUser,
  updateUserStore,
  getUserCount,
  getAllUsers,
  deleteUser,
} = require("../controller/userController");

// User login route
router.post("/login", userLogin);

// User sign up route
router.post("/signup", userSignUp);

// Get all users route
router.get("/", getAllUsers);
```

```
// Update user route
router.patch("/update", updateUser);

// Get one user by ID route
router.get("/:id/:role", getOneUser);

// Update user store route
router.patch("/updateUserStore", updateUserStore);

// Get user count for admin route
router.get("/getUserCountForAdmin", getUserCount);

// Delete user by ID route
router.delete("/deleteUser/:id", deleteUser);

module.exports = router;
```

server.js

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
require("dotenv").config();

const userRouter = require("./routes/user");

//Creating an express app
const app = express();

// Configure middleware functions
app.use(express.json({ limit: "100mb" }));
app.use(express.urlencoded({ limit: "100mb", extended: true }));
app.use(cors());

// Get port number and database URI from environment variables
const PORT = process.env.PORT;
const URI = process.env.URI;

// Connect to MongoDB database and start server
mongoose
  .connect(URI)
  .then(() => {
    console.log("Connection to MongoDB successful");
    app.listen(PORT, () => {
      console.log(`Server is running on ${PORT}`);
    });
  })
  .catch((err) => {
    console.log(err.message);
  });

// Set up route for handling requests to /api/user endpoint
app.use("/api/user", userRouter);
```

Product Service

Item.js

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const itemsSchema = new Schema({
  itemName: {
    type: String,
    required: true,
  },
  description: {
    type: String,
    required: true,
  },
  image: {
    type: String,
    required: true,
  },
  storeID: {
    type: String,
    required: true,
  },
  storeName: {
    type: String,
    required: true,
  },
  price: {
    type: Number,
    required: true,
  },
  totalPrice: {
    type: Number,
    required: true,
  },
  quantity: {
    type: Number,
    required: true,
  },
  discount: {
    type: Number,
    required: true,
  },
  reviews: {
    type: Array,
    default: [],
  }, //retings are taken as an overall rating from each reviewer
});

module.exports = mongoose.model("Items", itemsSchema);
```

itemController.js

```
const itemModel = require("../models/Item");

// Get all items
const getAllItems = async (req, res) => {
  try {
    const data = await itemModel.find();
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

// Add a new item
const postItem = async (req, res) => {
  // Get item details from request body
  const {
    itemName,
    image,
    storeName,
    description,
    category,
    price,
    quantity,
    discount,
    storeID,
  } = req.body;

  // Calculate total price after discount
  const totalPrice = price - (price * discount) / 100;

  try {
    // Create a new item model
    const ItemModel = new itemModel({
      itemName,
      description,
      image,
      category,
      price,
      quantity,
      discount,
      totalPrice,
      storeName,
      storeID,
    });

    // Save the new item to the database
    const data = await ItemModel.save();
    res.json(data);
  } catch (err) {
    res.json(err.message);
  }
};

// Get one item by ID
const getOneItem = async (req, res) => {
  // Get item ID from request body
  const { itemID } = req.body;

  try {
    // Find the item in the database using its ID
    const fetchedItem = itemModel.findOne({ _id: itemID });

    res.json(fetchedItem);
  } catch (err) {
    res.json(err.message);
  }
};

// Update an item
const updateItem = async (req, res) => {
  // Get item information from request body
  const itemInfo = req.body;

  try {
    let updatedInfo;
    if (itemInfo.redQuantity) {
      // Reduce item quantity if redQuantity is provided
      const { quantity } = await itemModel.findById(
        itemInfo.itemID,
        "quantity"
      );

      if (quantity < itemInfo.redQuantity) {
        throw new Error("Not enough stock available");
      }

      updatedInfo = await itemModel.findByIdAndUpdate(
        itemInfo.itemID,
        { $inc: { quantity: -itemInfo.redQuantity } },
        { new: true }
      );
    } else {
      // Calculate total price after discount
      itemInfo.totalPrice =
        itemInfo.price - (itemInfo.price * itemInfo.discount) / 100;

      // Update item details in the database
      updatedInfo = await itemModel.findByIdAndUpdate(
        itemInfo.itemID,
        itemInfo,
        { new: true }
      );
    }

    return res.json(updatedInfo);
  } catch (err) {
    res.json(err.message);
  }
};

// Delete an item by ID
const deleteItem = async (req, res) => {
  const id = req.params.id;

  try {
    // Find the item in the database and delete it
    const deletedRecord = await itemModel.findByIdAndDelete(id);
    res.json(deletedRecord);
  } catch (err) {
    res.json(err.message);
  }
```

```
};

//add a review for an item
const addReview = async (req, res) => {
  //to this data is just passed through the body (all of them)
  const { review, itemID, userID, userName, rating } = req.body; //_id is userID

  try {
    const insertReview = async (callback) => {
      const item = await itemModel.findOne({ _id: itemID });
      if (item) await callback(item.reviews); //item.reviews is an array
    };

    await insertReview(callBack);

    async function callBack(descArr) {
      //an array is passed in the parameter

      descArr.push({ userID, userName, rating, review });

      const data = await itemModel.findOneAndUpdate(
        { _id: itemID },
        { reviews: descArr }
      );
      res.json(data);
    }
  } catch (err) {
    res.json(err.message);
  }
};

//update a review for an item
const modifyReview = async (req, res) => {
  //to this data is just passed as normal text. all of them
  const { review, itemID, userID, userName, rating } = req.body; //_id is userID

  try {
    const removeReview = async (callback) => {
      const item = await itemModel.findOne({ _id: itemID });
      if (item) await callBack(item.reviews); //item.reviews is an array
    };

    removeReview();
  } catch (err) {
    res.json(err.message);
  }
  async function callBack(descArr) {
    //an item review array is passed in the parameter

    descArr = descArr.filter((obj) => {
      return obj.userID != userID;
    });

    descArr.push({ userID, userName, rating, review });

    const data = await itemModel.findOneAndUpdate(
      { _id: itemID },
      { reviews: descArr }
    );
    res.json({ updatedInfo: data });
  }
};

//delete a review for an item
const deleteReview = (req, res) => {
  //to this data is just passed as normal text. all of them
  const { itemID, userID } = req.body; //_id is userID

  try {
    const removeReview = async (callback) => {
      const item = await itemModel.findOne({ _id: itemID });
      if (item) await callBack(item.reviews); //item.reviews is an array
    };

    removeReview();
  } catch (err) {
    res.json(err.message);
  }

  async function callBack(descArr) {
    // item review array is passed in the parameter

    descArr = descArr.filter((obj) => {
      return obj.userID != userID;
    });

    const data = await itemModel.findOneAndUpdate(
      { _id: itemID },
      { reviews: descArr }
    );
    res.json({ updatedInfo: data });
  }
};
```

```
//delete all store items
const deleteAllItemsFromStore = async function (req, res) {
  try {
    const data = await itemModel.deleteMany({ storeID: req.params.id });
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

module.exports = {
  postItem,
  addReview,
  getAllItems,
  getOneItem,
  deleteItem,
  modifyReview,
  deleteReview,
  updateItem,
  deleteAllItemsFromStore,
};
```

item.js

```
const router = require("express").Router();
const {
  postItem,
  addReview,
  getAllItems,
  modifyReview,
  deleteReview,
  updateItem,
  getOneItem,
  deleteItem,
  deleteAllItemsFromStore,
} = require("../controller/itemController");

// Route for adding a new item
router.post("/addItem", postItem);

// Route for adding a new review to an item
router.patch("/addReview", addReview);

// Route for modifying an existing review for an item
```

```
router.patch("/modifyReview", modifyReview);

// Route for deleting a review for an item
router.patch("/deleteReview", deleteReview);

// Route for deleting an item
router.delete("/deleteItem/:id", deleteItem);

// Route for getting all items
router.get("/", getAllItems);

// Route for getting a specific item by ID
router.get("/findOne", getOneItem);

// Route for updating an item
router.patch("/updateItem", updateItem);

// Route for deleting all items for a specific store by store ID
router.delete("/deleteStoreItems/:id", deleteAllItemsFromStore);

module.exports = router;
```

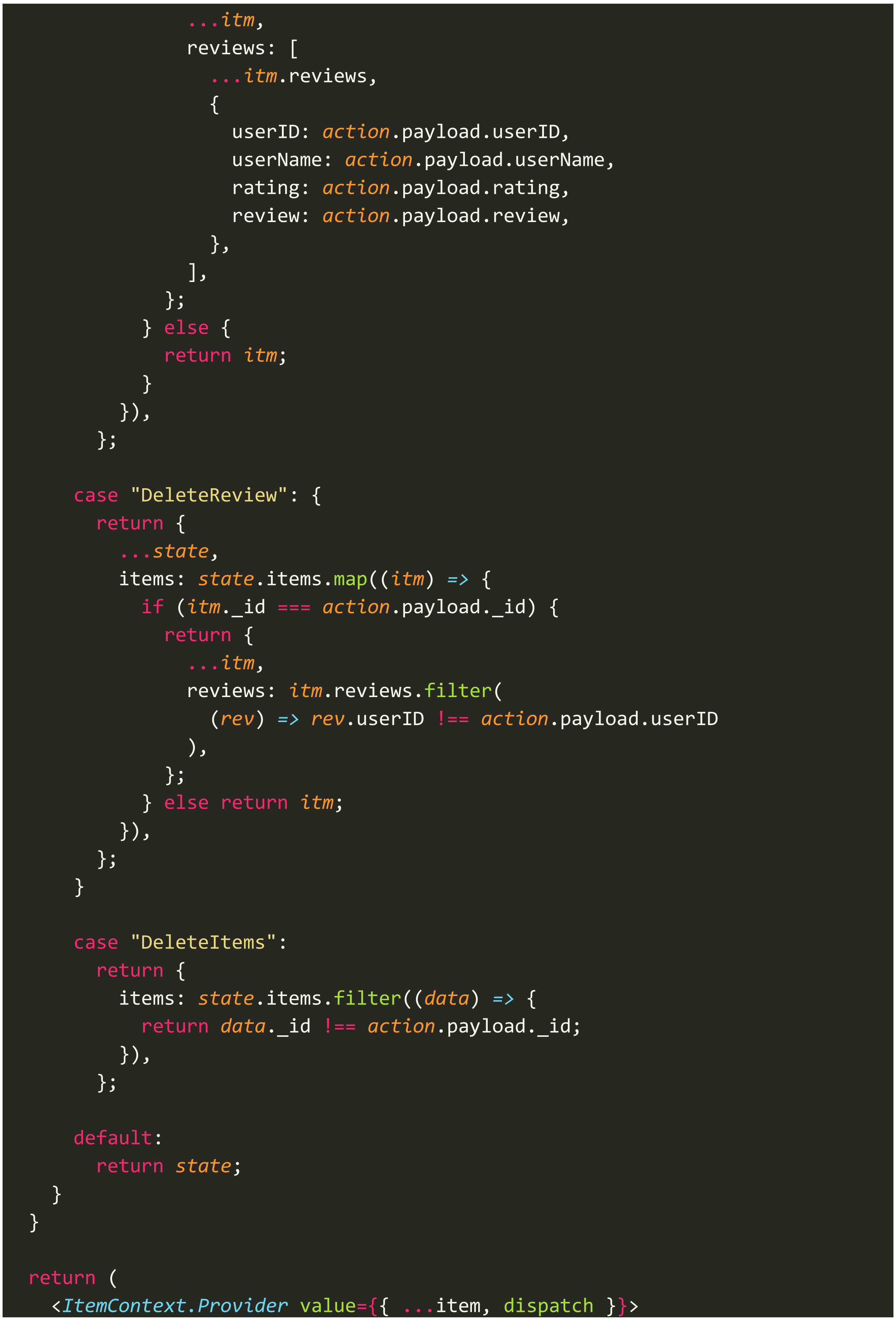
server.js

```
// Import necessary packages
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
require("dotenv").config(); // Load environment variables from .env file

const itemRouter = require("./routes/item"); // Import router for item-related
endpoints

// Create an Express app
const app = express();

// Middleware to parse JSON data and urlencoded data
app.use(express.json({ limit: "100mb" }));
app.use(express.urlencoded({ limit: "100mb", extended: true }));

// Enable CORS
app.use(cors());

const PORT = process.env.PORT; // Get port number from environment variables
const URI = process.env.URI; // Get MongoDB URI from environment variables
```

```
// Connect to MongoDB database and start server
mongoose
  .connect(URI, { useUnifiedTopology: true })
  .then(() => {
    console.log("Connection to MongoDB successful");
    app.listen(PORT, () => {
      console.log(`Server is running on ${PORT}`);
    });
  })
  .catch((err) => {
    console.log(err.message);
  });

app.use("/api/product",  itemRouter);  //  Mount  itemRouter  at  /api/product
endpoint
```

itemContext.js

```
import { createContext, useReducer } from "react";
import React from "react";

export const ItemContext = createContext();

export const ItemContextProvider = (props) => {
  const [item, dispatch] = useReducer(reducer, {
    items: [],
  });

  function reducer(state, action) {
    switch (action.type) {
      case "CreateItem":
        return { items: [action.payload, ...state.items] };

      case "SetItems":
        return { items: action.payload };

      case "AddReview":
        //[{userID, userName, rating, review},...{}] what a review contains
        //the payload struture {_id (item), userID, userName, rating, review}
        return {
          ...state,
          items: state.items.map((itm) => {
            if (itm._id === action.payload._id) {
              return {
```



```
              ...itm,
              reviews: [
                ...itm.reviews,
                {
                  userID: action.payload.userID,
                  userName: action.payload.userName,
                  rating: action.payload.rating,
                  review: action.payload.review,
                },
              ],
            };
          } else {
            return itm;
          }
        }),
      };

    case "DeleteReview": {
      return {
        ...state,
        items: state.items.map((itm) => {
          if (itm._id === action.payload._id) {
            return {
              ...itm,
              reviews: itm.reviews.filter(
                (rev) => rev.userID !== action.payload.userID
              ),
            };
          } else return itm;
        }),
      };
    }

    case "DeleteItems":
      return {
        items: state.items.filter((data) => {
          return data._id !== action.payload._id;
        }),
      };

    default:
      return state;
  }
}

return (
  <ItemContext.Provider value={{ ...item, dispatch }}>
```

```
      {props.children}
    </ItemContext.Provider>
  );
};
```

useItemContext.js

```
import { useContext, useEffect } from "react";
import { ItemContext } from "./itemContext";
import axios from "axios";

export const UseItemContext = () => {
  const itemContext = useContext(ItemContext);
  const { dispatch, items } = itemContext;

  useEffect(() => {
    async function fetchData() {
      try {
        const { data } = await axios.get("http://localhost:8081/api/product/");
        dispatch({
          type: "SetItems",
          payload: data,
        });
      } catch (err) {
        console.log(err);
      }
    }
    fetchData();
  }, []);

  function hasUserReviewedItem(itemId, userId) {
    const item = items.find((item) => item.id === itemId);

    const hasReviewed = item.reviews.some((review) => review.userId === userId);

    return hasReviewed;
  }
  return { itemContext, dispatch, items, hasUserReviewedItem };
};
```

Store Service

Order.js

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const ordersSchema = new Schema({
  userID: {
    type: String,
    required: true,
  },
  storeID: {
    type: String,
    required: true,
  },
  paymentID: {
    type: String,
    required: true,
  },
  address: {
    type: String,
    required: true,
  },
  orderedDate: {
    type: Date,
    default: Date.now,
    required: true,
  },
  status: {
    type: String,
    required: true,
    default: "Pending",
  },
  deliveredDate: {
    type: Date,
  },
  itemList: {
    type: Array,
  },
  reviewed: { type: Boolean, default: false },
});

module.exports = mongoose.model("Order", ordersSchema);
```

Store.js

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
const storesSchema = new Schema({
  merchantID: {
    type: String,
    required: true,
  },
  storeName: {
    type: String,
    required: true,
  },
  location: {
    type: String,
    required: true,
  },
  storeItem: {
    type: Array,
  },
  description: {
    type: String,
  },
  reviews: {
    type: Array,
    default: [],
  },
});

module.exports = mongoose.model("Store", storesSchema);
```

orderController.js

```
let Order = require("../models/Order");

// Create a new order
const createOrder = async (req, res) => {
  const { userID, storeID, paymentID, address, itemList } = req.body;

  const newOrder = new Order({
    userID,
    paymentID,
    address,
    storeID,
    itemList,
  });

  try {
    const data = await newOrder.save(); // Save the new order to the database
    res.json(data); // Send a JSON response containing the newly created order
data
  } catch (err) {
    res.json(err.message); // Send a JSON response with the error message if
there was an error saving the order to the database
  }
};

// Get all orders
const getAllOrder = async (req, res) => {
  try {
    const data = await Order.find(); // Find all orders in the database
    res.json(data); // Send a JSON response containing all the orders
  } catch (err) {
    res.send(err.message); // Send a response with the error message if there
was an error getting the orders
  }
};

// Update an order
const updateOrder = async (req, res) => {
  const { orderID, status } = req.body;

  const updateStore = {
    status,
  };
  const update = await Order.findById(orderID, updateStore) // Find the order
by ID and update its status
    .then(() => {
      res.status(200).send({ Status: "Order updated", order: update }); // Send
a success response with the updated order data
    })
    .catch((err) => {
      res.status(500).send({ status: "Error with updating data" }); // Send an
error response if there was an error updating the order
    });
};

// Get a single order by ID
const getOneOrder = async (req, res) => {
  await Order.findById(req.params.id) // Find the order by ID
    .then((order) => {
      res.status(200).send(order); // Send a JSON response with the order data
    })
    .catch((err) => {
      res
        .status(500)
        .send({ status: "Error Fetching Order", error: err.message }); // Send
an error response if there was an error getting the order
    });
};

// Get all orders for a specific store
const getAllOrderPerStore = async (req, res) => {
  try {
    const orders = await Order.find({ storeID: req.params.id }); // Find all
orders for the specified store

    const result = orders.map((order) => ({
      ...order.toObject(),
      totalAmount: order.itemList.reduce(
        (total, item) => total + item.itemPrice * item.itemQuantity,
        0
      ),
    }));

    res.json(result); // Send a JSON response containing all the orders for the
specified store
  } catch (error) {
    res.status(500).json({ error: "Failed to get orders for store" }); // Send
an error response if there was an error getting the orders for the store
  }
};

// This function updates the status of an order based on the orderID
const updateOrderStatus = async (req, res) => {
  const { orderID, status } = req.body;

  try {
    const data = await Order.findByIdAndUpdate(
      orderID,
      { status },
      { new: true }
    );
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

// This function retrieves the count of all orders for the admin dashboard
const getOrderCountForAdmin = async (req, res) => {
  try {
    const orderCount = await Order.countDocuments();
    res.json({ orderCount });
  } catch (err) {
    res.send(err.message);
  }
};

// This function retrieves all orders for all stores
const getAllStoreOrders = async (req, res) => {
  try {
    const data = await Order.find();
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

// This function retrieves all orders for a particular user
const getAllUserOrders = async (req, res) => {
  try {
    const data = await Order.find({ userID: req.params.id });
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

// This function sets the reviewed status of an order to true
const setReviewStatus = async (req, res) => {
  try {
    const data = await Order.findByIdAndUpdate(req.params.id, {
      reviewed: true,
    });
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

module.exports = {
  createOrder,
  getAllOrder,
  updateOrder,
  getOneOrder,
  getAllUserOrders,
  getAllStoreOrders,
  getAllOrderPerStore,
  updateOrderStatus,
  setReviewStatus,
  getOrderCountForAdmin,
};
```

storeController.js

```
// Importing Store model
let Store = require("../models/Store");

// Creating a new store in the database
const createStore = async (req, res) => {
  const { storeName, location, merchantID } = req.body;

  // Creating a new Store object with the provided data
  const newStore = new Store({
    storeName,
    merchantID,
    location,
  });

  // Saving the new store to the database
  await newStore
    .save()
    .then(() => {
      // Sending the newly created store object as response
      res.json(newStore);
    })
    .catch((err) => {
       // If there is an error, logging the error message and sending it as
response
      console.log(err.message);
      res.send(err.message);
    });
};

// Getting all stores from the database
const getAllStore = async (req, res) => {
  await Store.find()
    .then((store) => {
      // Sending all store objects as response
      res.json(store);
    })
    .catch((err) => {
      // If there is an error, sending the error message as response
      res.send(err.message);
    });
};

// Updating basic store info details
const updateStore = async (req, res) => {
  const { storeName, location, storeID } = req.body;

  // Creating an object with the updated values
  const updateStore = {
    storeName,
    location,
  };

  try {
    // Finding the store by the given ID and updating the store details with
the new values
    const updatedStore = await Store.findOneAndUpdate(
      { _id: storeID },
      updateStore,
      { new: true }
    ); // Sending the updated store object as response
    res.send(updatedStore);
  } catch (err) {
    // If there is an error, sending the error message as response
    res.send(err.message);
  }
};

// Deleting a store from the database
const deleteStore = async (req, res) => {
  try {
    // Finding the store by the given ID and deleting it from the database
    const data = await Store.findByIdAndDelete(req.params.id);
    // Sending the deleted store object as response
    res.json(data);
  } catch (err) {
    // If there is an error, sending the error message as response
    res.send(err.message);
  }
};

// Getting a store by ID
const getOneStore = async (req, res) => {
  const id = req.params.id;

  try {
    // Finding the store by the given ID and sending it as response
    const data = await Store.findById(id);
    res.json(data);
  } catch (err) {
    // If there is an error, sending the error message as response
    res.send(err.message);
  }
};

// Getting the description of a store by ID
const getStoreDescription = async (req, res) => {
  try {
    // Finding the store by the given ID and selecting the 'description' field
    const data = await Store.findById(req.params.id, { description });
    // Sending the store's description as response
    res.json(data);
  } catch (err) {
    // If there is an error, sending the error message as response
    res.send(err.message);
  }
};

//get the itemCount from store
const getStoreItemCount = async (req, res) => {
  const storeID = req.params.id;

  try {
    const data = await Store.findOne({ _id: storeID });

    res.json({ itemCount: data.storeItem.length });
  } catch (err) {
    res.send(err.message);
  }
};

//add items to store
const addStoreItem = async (req, res) => {
  const { item, storeID } = req.body;

  try {
    const store = await Store.findOne({ _id: storeID });

    var itemArray = store.storeItem;

    itemArray.push(item);

    const updatedStore = await Store.findOneAndUpdate(
      { _id: storeID },
      { storeItem: itemArray },
      { new: true }
    );

    res.send(updatedStore);
  } catch (err) {
    res.send(err.message);
  }
};

//modify the items in the store
const modifyStoreItem = async (req, res) => {
  const { item, storeID } = req.body;

  try {
    const store = await Store.findOne({ _id: storeID });

    var itemArray = store.storeItem;

    var itemArray = itemArray.map((itm) => {
      if (itm._id === item._id) {
        // Replace elements in itm with elements from item
        return Object.assign({}, itm, item);
      } else {
        // Return original object
        return itm;
      }
    });

    const updatedStore = await Store.findOneAndUpdate(
      { _id: storeID },
      { storeItem: itemArray },
      { new: true }
    );

    res.send(updatedStore);
  } catch (err) {
    res.send(err.message);
  }
};

//delete item from store
const deleteStoreItem = async (req, res) => {
  const { storeID, itemID } = req.body;

  try {
    const store = await Store.findOne({ _id: storeID });
    const itemArray = store.storeItem;

    var newArray = itemArray.filter((itm) => itm._id !== itemID);

    const updatedStore = await Store.findOneAndUpdate(
      { _id: storeID },
      { storeItem: newArray },
      { new: true }
    );

    res.send(updatedStore);
  } catch (err) {
    res.send(err.message);
  }
};

//add store review
const addReview = async (req, res) => {
  //to this data is just passed through the body (all of them)
  const { review, storeID, userID, userName, rating } = req.body; //_id is
userID

  try {
    const insertReview = async (callback) => {
      const store = await Store.findOne({ _id: storeID });
      if (store) await callback(store.reviews); //item.reviews is an array
    };

    await insertReview(callBack);

    async function callBack(descArr) {
      //an array is passed in the parameter

      descArr.push({ userID, userName, rating, review });

      const data = await Store.findOneAndUpdate(
        { _id: storeID },
        { reviews: descArr }
      );
      res.json(data);
    }
  } catch (err) {
    res.json(err.message);
  }
};
```

```
//exporting necessary functions to be used in the route file
module.exports = {
  createStore,
  getAllStore,
  updateStore,
  addReview,
  deleteStore,
  getOneStore,
  getStoreItemCount,
  addStoreItem,
  deleteStoreItem,
  modifyStoreItem,
};
```

order.js

```
const router = require("express").Router();

// Import order controller functions
const {
  createOrder,
  getAllOrder,
  updateOrder,
  getOneOrder,
  getAllOrderPerStore,
  updateOrderStatus,
  getOrderCountForAdmin,
  getAllStoreOrders,
  getAllUserOrders,
  setReviewStatus,
} = require("../controller/orderController");

// Route for creating a new order
router.post("/add", createOrder);

// Route for getting all orders
router.get("/", getAllOrder);

// Route for updating an existing order
router.patch("/update/", updateOrder);

// Route for getting a single order by ID
router.get("/get/:id", getOneOrder);

// Route for getting all orders for a specific store
```

```
router.get("/getStoreOrder/:id", getAllOrderPerStore);

// Route for updating the status of an existing order
router.patch("/updateOrderStatus", updateOrderStatus);

//Route for getting the total order count to display in the admin dashboard
router.get("/getOrderCountForAdmin", getOrderCountForAdmin);

//Route for getting all the orders
router.get("/getAllStoreOrders", getAllStoreOrders);

//Route for getting all the orders for a particular user
router.get("/getAllStoreOrders/:id", getAllUserOrders);

//Route to set review status as true once user submites a store review
router.patch("/setReviewStatus/:id", setReviewStatus);

module.exports = router;
```

store.js

```
const router = require("express").Router();
const requireAuth = require("../middleware/requireAuth");

const {
  createStore,
  getAllStore,
  updateStore,
  getOneStore,
  deleteStore,
  addStoreItem,
  deleteStoreItem,
  getStoreItemCount,
  modifyStoreItem,
  addReview,
} = require("../controller/storeController");

// router.use(requireAuth);
//create store
router.post("/add", createStore);
//display
router.get("/", getAllStore);

//to get the item count in a store
router.get("/getStoreItemCount/:id", getStoreItemCount);
```

```
//update store info
router.put("/update/", updateStore);

//add a review for a store item
router.patch("/addReview", addReview);

//delete a store
router.delete("/delete/:id", deleteStore);

//get one store
router.get("/get/:id", getOneStore);

router.patch("/updateItem", addStoreItem); //to add item to store item Array
router.patch("/modifyItem", modifyStoreItem); //to modify the item in Store
item array
router.patch("/deleteStoreItem", deleteStoreItem); //to delete the item from
store item array

module.exports = router;
```

requireAuth.js

```
const jwt = require("jsonwebtoken");
const axios = require("axios");

const requireAuth = async (req, res, next) => {
  const { authorization, role } = req.headers;

  // Check if authorization token is provided in the request header
  if (!authorization) {
    return res.status(401).json({ error: "Authorization token not found" });
  }

  // Extract the token from the authorization header
  const token = authorization.split(" ")[1];

  try {
    // Verify the token using the secret key
    const { id } = jwt.verify(token, process.env.SECRET);

    // Retrieve user data from API using the id and role from the token
    const { data } = await axios.get(
      `http://localhost:8080/api/user/${id}/${role}`
    );
    // Attach user data to the request object
    req.user = data;

    // Call next middleware function
    next();
  } catch (error) {
    console.log(error);
    res.status(401).json({ error: "Unauthorized Request" });
  }
};

module.exports = requireAuth;
```

server.js

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
require("dotenv").config();

const storeRouter = require("./routes/store");
const orderRouter = require("./routes/order");

//Creating an express app
const app = express();

// Configure middleware functions
app.use(express.json({ limit: "100mb" }));
app.use(express.urlencoded({ limit: "100mb", extended: true }));
app.use(cors());

// Get port number and database URI from environment variables
const PORT = process.env.PORT;
const URI = process.env.URI;

// Connect to MongoDB database and start server
mongoose
  .connect(URI, { useUnifiedTopology: true })
  .then(() => {
    console.log("Connection to MongoDB successful");
    app.listen(PORT, () => {
      console.log(`Server is running on ${PORT}`);
    });
  })
  .catch((err) => {
    console.log(err.message);
  });

// Set up routes for handling requests to /api/store and /api/order endpoints
app.use("/api/store", storeRouter);
app.use("/api/order", orderRouter);
```

Payment Service

Payment.js

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const paymentSchema = new Schema({
  amount: {
    type: Number,
    required: true,
  },
  itemList: {
    type: Array,
    default: [],
  },
  userID: { type: String, required: true },
  status: {
    type: String,
    default: "Processing",
    required: true,
  },
  date: {
    type: Date,
    default: Date.now,
    required: true,
  },
});

module.exports = mongoose.model("Payment", paymentSchema);
```

paymentController.js

```
// Importing the Payment model
let Payment = require("../models/Payment");

// This function creates a new payment and saves it to the database
```

```
const createPayment = async (req, res) => {
  const amount = Number(req.body.amount);
  const { itemList, userID, storeID } = req.body;

  // Creating a new payment object
  const newPayment = new Payment({
    amount,
    itemList,
    userID,
    storeID,
  });

  // Saving the payment object to the database

  try {
    const data = await newPayment.save();
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

// This function retrieves all payments from the database
const getAllPayment = async (req, res) => {
  await Payment.find()
    .then((payment) => {
      res.json(payment);
    })
    .catch((err) => {
      res.send(err.message);
    });
};

// This function updates a payment's status
const updatePayment = async (req, res) => {
  const { paymentID, status } = req.body;

  // Creating an object to update the payment's status
  const updatePayment = {
    status,
  };

  // Finding and updating the payment object in the database
  const update = await Payment.findOneAndUpdate(
    { _id: paymentID },
    updatePayment,
    { new: true }
  )
    .then(() => {
      res.status(200).json(update);
    })
    .catch((err) => {
      res.status(500).send({ status: "Error updating data" });
    });
};

// This function deletes a payment from the database
const deletePayment = async (req, res) => {
  const { paymentID } = req.body;

  // Finding and deleting the payment object from the database
  await Payment.findByIdAndDelete(paymentID)
    .then(() => {
      res.status(200).send({ status: "Payment Deleted" });
    })
    .catch((err) => {
      res.status(500).send({ status: "Error...." });
    });
};

// This function calculates the total payment amount for a particular store and
the number of orders
const getTotalPaymentPerStore = async (req, res) => {
  const storeID = req.params.id;
  try {
    const results = await Payment.aggregate([
      { $match: { "itemList.storeID": storeID } },
      { $unwind: "$itemList" },
      { $match: { "itemList.storeID": storeID } },
      {
        $group: {
          _id: "$_id",
          totalAmount: {
            $sum: {
              $multiply: ["$itemList.itemPrice", "$itemList.itemQuantity"],
            },
          },
        },
      },
      {
        $group: {
          _id: null,
          totalAmount: { $sum: "$totalAmount" },
          orderCount: { $sum: 1 },
        },
      },
    ]);

    if (results.length > 0) {
      const { totalAmount, orderCount } = results[0];
      res.send({ total: totalAmount, orderCount });
    } else {
      res.send({ total: 0, orderCount: 0 });
    }
  } catch (err) {
    res.json(err.message);
  }
};

const updatePaymentStatus = async (req, res) => {
  const { paymentID, status } = req.body;

  try {
    const data = await Payment.findByIdAndUpdate(
      paymentID,
      { status },
      { new: true }
    );
    res.json(data);
  } catch (err) {
    res.send(err.message);
  }
};

const getTotalPaymentForAdmin = async (req, res) => {
  try {
    const data = await Payment.find();
    const amountForStore =
      data.reduce((acc, dat) => acc + dat.amount, 0) * 0.15;

    res.json({ amountForStore });
  } catch (err) {
    res.send(err.message);
  }
};

// Exporting the functions to be used in other modules
module.exports = {
  createPayment,
  getAllPayment,
  updatePayment,
  deletePayment,
  getTotalPaymentPerStore,
  updatePaymentStatus,
  getTotalPaymentForAdmin,
};
```

payment.js

```
const router = require("express").Router(); // Importing the Router class from
the Express package

const {
  createPayment,
  getAllPayment,
  deletePayment,
  updatePayment,
  getTotalPaymentPerStore,
  updatePaymentStatus,
  getTotalPaymentForAdmin,
} = require("../controller/paymentController"); // Importing the controller
functions from '../controller/paymentController'

//create a new payment
router.post("/add", createPayment); // Handles POST requests to create a new
payment using the createPayment() function

//get all payments
router.get("/", getAllPayment); // Handles GET requests to get all payments
using the getAllPayment() function

//update all payments
router.put("/update/", updatePayment); // Handles PUT requests to update
payments using the updatePayment() function

//delete a payment
router.delete("/delete/", deletePayment); // Handles DELETE requests to delete
a payment using the deletePayment() function

//To get the total payments done to a certain store
router.get("/getStoreTotal/:id", getTotalPaymentPerStore); // Handles GET
requests to get the total payments made to a specific store using the
getTotalPaymentPerStore() function

router.patch("/updatePaymentStatus", updatePaymentStatus); //Handles updating
of payment status
```

```
router.get("/getAdminTotal",  getTotalPaymentForAdmin);  //Handles  calculating
the online store profit

module.exports = router; // Exports the router instance for use in the app.
```

server.js

```
const express = require("express"); // Express.js framework
const mongoose = require("mongoose"); // MongoDB object modeling tool
const cors = require("cors"); // Cross-Origin Resource Sharing middleware
require("dotenv").config(); // Loads environment variables from a .env file

// Importing route modules
const paymentRouter = require("./routes/payment");

// Creating an Express.js app
const app = express();

// Using middleware to parse incoming requests
app.use(express.json({ limit: "100mb" })); // Parses incoming JSON data
app.use(express.urlencoded({  limit:  "100mb",  extended:  true  }));  //  Parses
incoming URL-encoded data
app.use(cors()); // Enables Cross-Origin Resource Sharing for all routes

//  Defining  the  server  port  and  the  MongoDB  database  URI  using  environment
variables
const PORT = process.env.PORT;
const URI = process.env.URI;

// Connecting to the MongoDB database and starting the server
mongoose
  .connect(URI, { useUnifiedTopology: true })
  .then(() => {
    console.log("Connection to MongoDB successful");
    app.listen(PORT, () => {
      console.log(`Server is running on ${PORT}`);
    });
  })
  .catch((err) => {
    console.log(err.message);
  });

// Adding a route for handling payment-related requests
app.use("/api/payment", paymentRouter);
```