Sri Lanka Institute of Information Technology

Distributed Systems

SE – 3020

Assignment 1

Report

Name	IT number
Lekamge L.R.S.T	IT20159030
Weerawarna N.V	IT21110948
Samaranayake N.P.T.D	IT21047824
Dissanayake D.M.J.C.B	IT21151392

Contents

Introduction

“RedStream Herbal Store” is an ecommerce site for buying and selling herbal products online. The web application is primarily developed using MERN stack (MongoDB, ExpressJs, React and NodeJS). Asynchronous web client is developed using ReactJS and Tailwind CSS. Redux is used to manage the altering states of the application. The backend of the application is developed according to the microservices architecture. The web application consists of several services.

- User service
- Product service
- Order service
- Admin service
- Email service
- SMS service
- Payment service –PayPal
- Delivery service – Dummy service

The backend is deployed using Docker and Kubernetes.

The Web application provides 3 main User Interfaces to interact with the system.

- StoreFront
- Seller Dashboard
- Admin Dashboard

The Store Front is used by customers to buy products. The Seller Dashboard is used by the sellers to add products and manage their store. The Admin Dashboard is used by the administrators to manage the e-commerce platform. These User Interfaces provide the platform users a comprehensive functionality to use and navigate the web application effectively. In-depth explanations are given in the coming sections of the report.

Contributions

Name	IT number	Contribution	
		Frontend	**Backend**
Lekamge L.R.S.T	IT20159030	StoreFront - User management - Shop management - Search and filter - Order Tracker - Product Reviewing - Dummy delivery service	User Service
Weerawarna N.V	IT21110948	StoreFront - Shopping cart - Product Detail - Checkout - Place order - Shipping service - Payment service - SMS service	Order service SMS service

Samaranayake N.P.T.D	IT21047824	Admin Dashboard
		- Statistics
		  - Sales
		  - Orders
		- Manage Orders
		- Manage Products
		- Manage User
		- Configuration
			Admin service
			Email service

Dissanayake D.M.J.C.B	IT21151392	Seller Dashboard - Add Product - Manage Product - Manage Orders - Manage Shop	Product service

System Requirements

- Sellers should be able to upload new items.
- Buyers should be able to purchase items.
- Buyers should be able to add items to cart and purchase multiple items.
- Buyers should be able to search for items.
- Once a buyer makes the purchase, an administrator must confirm the order before the seller can ship the item.
- Buyers can select a delivery option which will send a request to a 3rd party delivery service.
- A commission must be charged from each purchase.
- Buyers should be able to make payments using PayPal or credit card.
- Once the payment is made, the buyer should be notified by SMS and email.
- An interface should be available to see the status of the order.
- Users should be able to review items and sellers.

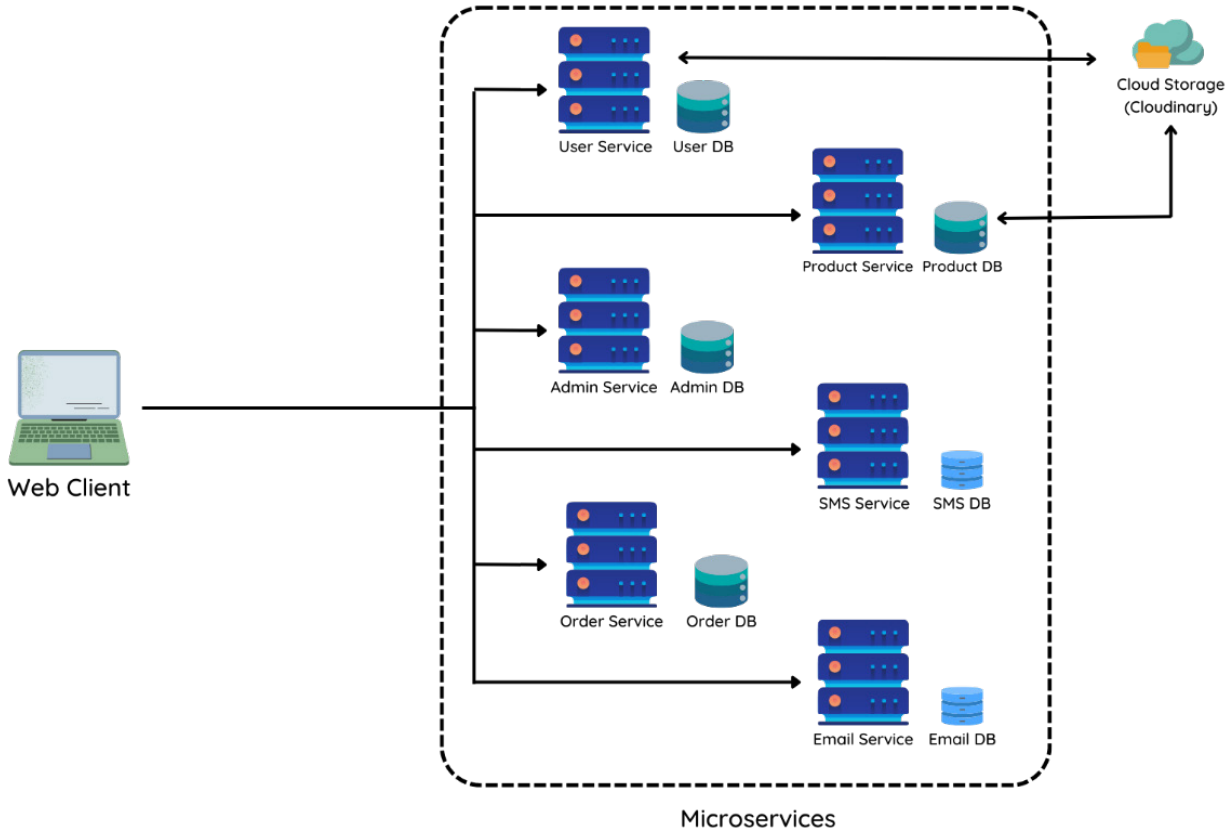
System Architecture

For this web application, we have implemented the microservices architecture in the backend. This platform consists of User service, Product service, Order service, Admin service, Email service and SMS service. Each service in this architecture performs a specific business function and interacts with other services via well-defined interfaces.

By decomposing the application into smaller services, it is easier to make modifications to specific services without affecting the system in its entirety. As a result, we may work on several application services concurrently, and adding or removing new functionality is relatively easier. This method offers several benefits including increased scalability, quicker development and deployment, simpler maintenance, and higher fault tolerance.

Each microservice must be packaged as a Docker image and deployed as a container to use Docker and Kubernetes to deploy them. These containers are scaled and deployed by Kubernetes, which also makes sure they are operational in perfect condition. Additionally, Kubernetes has capabilities that make managing and scaling microservices in a dynamic and distributed environment easier such as rolling updates, automatic scaling, load balancing, and self-healing. In our web application microservices include admin, order, product, user, email, and SMS services.

High level Architectural Diagram



User Service

The User Service manages user registration, authentication, and authorization, thus becoming an essential component of our web application. This service is responsible of ensuring that sure that only users who have registered can access specific areas of the web application and conduct specific functions. It provides APIs for registering users, updating, and deleting user accounts. For authentication, it utilizes the JSON Web Token (JWT) mechanism. The bcrypt.js library, which is a popular library for password hashing and comparison, is used to encrypt the user's password. The User Service implements middleware for managing incoming HTTP requests in addition to authentication and authorization. Requests are intercepted by this middleware, which then performs several types of checks on them, including confirming that the user is authenticated and permitted to carry out the requested action. This makes ensuring that only approved users are allowed to perform specific tasks, such updating their account information or accessing specific sections of the web application. A database layer is also implemented by User Service for the purpose of securely and effectively storing user account data. This database layer makes use of a NoSQL database, such MongoDB, which enables quick and effective retrieval of user data.

Product Service

In our web-based application, the products are managed by the Product Service. In addition to managing product inventory and pricing data, it offers APIs for adding, retrieving, updating, and removing products. This service is connected with a third-party service called cloudinary to store images of the products. The Product Service interacts with other microservices, including the User Service, Order Service, Email Service, SMS Service, and Admin Service and this communication is facilitated through APIs. The Product Service maintains a collection of product related information in the MongoDB database, which is accessible through the service's APIs for retrieval and updating.

Order Service

The Order Service is an essential component of our web-based application that controls the complete workflow of order processing. It is the responsibility of this service to offer a collection of APIs that enable users to make new orders, modify existing orders, and, if necessary, delete orders. Additionally, it ensures that each order is tracked and processed in a prompt and effective manner. The Order Service's main responsibilities include coordinating with many other services and elements of the web app. This involves corresponding with the User Service to verify the identity of the user placing the order, the Admin Service to coordinate and oversee the order fulfilment process, the Email Service to send customers order confirmation emails, and the SMS Service to send SMS updates of the order. Additionally using a MongoDB database, the Order Service maintains and manages all records pertaining to orders. This includes details about the order, payment, shipment, and tracking information. Middleware such JWT protection is developed to authenticate and authorize users and admins to make requests to the service to ensure the security of the Order Service. The Order Service interfaces with PayPal's API for payment processing to securely process payments from customers. Additionally, the service utilizes the Shippo API to interface with third-party shipping services and determine shipping costs in order to manage the delivery process.

Admin Service

The admin service is responsible for monitoring the metrics and statistics related to sales and orders. It uses MongoDB for the database and communicates with the User Service, Order Service, and Product Service. The Admin Service includes APIs for calculating daily, monthly, and yearly sales as well as retrieving information about sales and orders. These APIs are used to create charts that display the system's sales trends. Charts displaying the system's sales trends are produced using these APIs. Additionally, the Admin Service offers APIs for calculating and obtaining order statistics. These statistics provide data on the quantity of orders, the number of products ordered, and the overall revenue. Additionally, an API can be used for commission configuration through the admin service.

Email Service

The Email Service is implemented in Node.js, which integrates with mailgen, nodemailer, and Gmail SMTP to offer a dependable and effective method for delivering emails, notifying users via email. Sending users order confirmation emails is the Email Service's primary feature. The Email Service receives notification when a user completes a transaction on the system, and it uses mailgen to create a personalized order confirmation email. The email includes all pertinent information regarding the order, including the items ordered and the total amount. The Email Service subsequently sends the email via Gmail SMTP using nodemailer, ensuring that it reaches the user's inbox.

SMS Service

The SMS service integrates with notify.lk, a third-party SMS service provider that offers dependable rapid SMS delivery services. The Kubernetes container orchestration platform is used for the deployment of the Node.js-based SMS Service. To facilitate communication between the web application and the SMS provider, the SMS Service offers a set of APIs. These APIs' minimal weight, scalability, and fault tolerance make it possible for customers to get SMS notifications on time and with reliability. The SMS Service's ability to interface with other microservices in the architecture, such as the Order Service and User Service, is one of its essential characteristics. For instance, when a user places an order, the Order Service communicates with the SMS Service to send an SMS notification to the user confirming the order.

Security Mechanisms

In microservice architecture, it is important to have a robust and secure authentication system in place. In our project, JSON Web Tokens (JWT) were used for authentication, along with bcrypt.js for password hashing.

A reliable and secure authentication system is essential for microservice architecture. In our project, we used bcrypt.js for password hashing and JSON Web Tokens (JWT) for authentication. Here's how we generate the token:

```
import jwt from 'jsonwebtoken';
const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    expiresIn: '30d',
  });
};
```

The user service uses bcrypt.js to hash and compare the passwords of logged-in users to ensure that their credentials are valid. If the credentials are legitimate, the user service creates a JWT token and returns it to the client. The user ID contained in this token, which is utilized for further requirements to the backend, is user-specific data. This is how a password is hashed with bcrypt:

```
userSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password);
};
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) {
    next();
  }
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
});
```

An auth middleware was created to ensure that only authenticated users may access the microservices. This middleware examines the request headers for the presence of a valid JWT token and, if user is identified, permits the request to continue. An unauthorized error is returned by the middleware if the token is invalid. This user middleware is used by all system microservices to guarantee that only authorized users may access the APIs. By ensuring that only authorized users can interact with the backend, this additionally increases the security of the system.

Service Interfaces

Admin Service

- /v1/commission
    - GET – Get commission rate.
    - POST – Update current commission.
- /v1/order-list
    - GET – Call order service API and get a list of all orders.
- /v1/get-order
    - POST – Call order service API and get a specific order.
- /v1/update-order
    - POST – Call order service API and update a specific order.
- /v1/query-order
    - POST – Call order service API and get list of queried orders.
- /v1/product-list
    - GET – Call product service API and get list of all products.
- /v1/user-list
    - GET – Call user service API and get list of all users.
- /v1/get-user
    - POST – Call user service API and get a specific user.
- /v1/update-user
    - POST – Call user service API and update a specific user.
- /v1/sales/daily/:year/:month
    - GET – Get daily sales statistics for a specified month.
- /v1/sales/daily
    - POST – Calculate daily sales statistics for a specified month.
- /v1/sales/monthly
    - POST – Calculate monthly sales statistics for a specified year.
- /v1/sales/monthly/:year
    - GET – Get monthly sales statistics for a specified year.
- /v1/sales/yearly
    - GET – Get yearly sales statistics for all orders.
    - POST – Calculate yearly sales statistics for all orders.
- /v1/orders/monthly
    - POST – Calculate monthly order statistics for a specified year.
- /v1/orders/:year/:month
    - GET – Get monthly order statistics for a specified year.

User Service

- /api/users/
    - POST - Call user service API and register new users.
    - GET - Call user service API and get all users.

- /api/users/login'
    - authUser - Call user service API and authenticate user.
- /api/users/account'
    - PUT - Call user service API and update user account.
- /api/users/:id'
    - DELETE - Call user service API and delete user account by user Id.
    - GET - Call user service API and get user information by user Id.
- /api/shops/
    - POST - Call user service API and create shop.
    - GET - Call user service API and retrieve shops.
- /api/shops/user/:id
    - GET - Call user service API and get shop by user ID.
- /api/shops/all
    - GET - Call user service API and get all shops.
- /api/shops/:id
    - GET - Call user service API and get shop by shop ID.
    - PUT - Call user service API and update shop details by shop ID.
    - DELETE - Call user service API and delete shop by shop ID.

Product Service

- /api/products
    - GET – Call product service API and get all products.
    - POST – Call product service API and create new product.
- /api/products/user/:id
    - GET - Call product service API and get sellers product.
- /api/search/:searchTerm
    - GET - Call product service API and get product by search.
- /api/products/top
    - GET - Call product service API and get top rated products.
- /api/products/:id/reviews
    - GET - Call product service API and get reviews.
    - POST - Call product service API and add new review.
- /api/products/:id
    - GET - Call product service API and get product by id.
    - DELETE - Call product service API and delete a product by id.
    - PUT - Call product service API and update a product by id.

Order Service

- /api/orders
    - GET – Call order service API and get all orders.
    - POST - Call order service API and create new orders.
- /api/orders/:id
    - GET – Call order service API and get order by id.

- /api/orders/:id/pay
    - PUT – Call order service API and update the order as paid.
- /api/orders/:id/confirm
    - PUT – Call order service API and update the order as confirm.
- /api/orders/:id/reject
    - PUT – Call order service API and update the order as reject.
- /api/orders/:id/deliver
    - PUT – Call order service API and update the order as delivered.
- /api/orders/query
    - POST – Call order service API and get queried list of orders.
- /api/orders/user/:id
    - GET – Call order service API and get orders by user id.
- /api/orders/seller/products/:id
    - GET – Call order service API and get orders for seller.
- /api/orders/query/:id/shipped
    - PUT – Call order service API and update the order as shipped.
- /api/config/paypal
    - GET – Send PayPal client id to frontend.
- /api/config/shippo
    - PUT – Send shippo shipping service api key to frontend.

Email Service

- /v1/send
    - POST – Send an email with the specified order information, to a specified email.

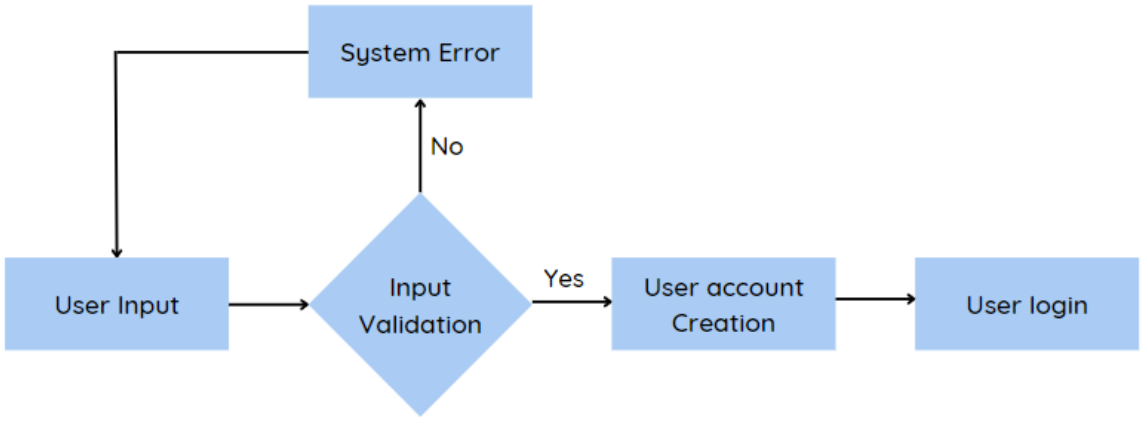
SMS Service

- /api/sms/send
    - POST – Send an SMS with the specified order information, to a specified SMS.

Workflows

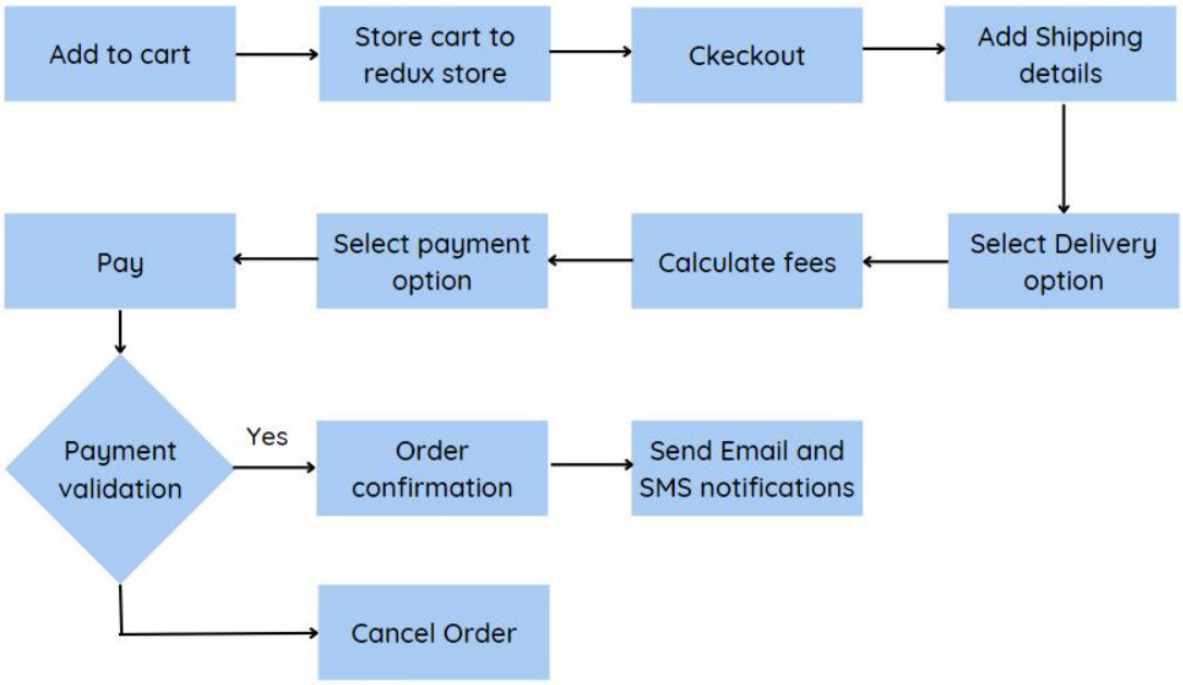
User Registration workflow

1. User inputs Registration details
2. System validates the input details.
3. If invalid details are provided, the system gives an error.
4. If valid details are provided, the system creates a new user account.
5. The system logs in the user.



Order Processing workflow

1. User adds products to the shopping cart.
2. Shopping cart updates the products from react redux store.
3. User proceeds to checkout page and enter the shipping information.
4. User selects a shipping method and the total value is calculated including product items cost, commission and shipping cost.
5. User selects a payment method, such as PayPal or credit/debit card, and enter payment details.
6. Once the payment is successful, an order confirmation email and SMS notification are sent to the customer.


Add to cart
Store cart to redux store
Ckeckout
Add Shipping details
Select Delivery option
Calculate fees
Select payment option
Pay
Payment validation
Yes
Order confirmation
Send Email and SMS notifications
Cancel Order

Appendix

User Service

Controllers

User controller

```
import User from '../models/userModel.js';
import generateToken from '../utils/generateToken.js';
import asyncHandler from 'express-async-handler';

// @desc    Auth user & get token
// @route   POST /api/users/login
// @access  Public
const authUser = asyncHandler(async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (user && (await user.matchPassword(password))) {
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
      isSeller: user.isSeller,
      profilePic: user.profilePic,
      firstName: user.firstName,
      lastName: user.lastName,
      phone: user.phone,
      number: user.shippingInfo.number,
      line1: user.shippingInfo.line1,
      line2: user.shippingInfo.line2,
      city: user.shippingInfo.city,
      state: user.shippingInfo.state,
      zip: user.shippingInfo.zip,
      country: user.shippingInfo.country,
      token: generateToken(user._id),
    });
  } else {
    res.status(401).json({
      error: 'Invalid email and password',
    });
  }
});
```

```
// @desc    Register a new user
// @route   POST /api/users
// @access  Public
const registerUser = asyncHandler(async (req, res) => {
  const { name, email, password } = req.body;

  const userExists = await User.findOne({ email });

  if (userExists) {
    res.status(400);
    throw new Error('User already exists');
  }

  const user = await User.create({
    name,
    email,
    password,
  });

  if (user) {
    res.status(201).json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
      isSeller: user.isSeller,
      profilePic: user.profilePic,
      firstName: user.firstName,
      lastName: user.lastName,
      phone: user.phone,
      number: user.shippingInfo.number,
      line1: user.shippingInfo.line1,
      line2: user.shippingInfo.line2,
      city: user.shippingInfo.city,
      state: user.shippingInfo.state,
      zip: user.shippingInfo.zip,
      country: user.shippingInfo.country,
      token: generateToken(user._id),
    });
  } else {
    res.status(400);
    throw new Error('Invalid user data');
  }
```

```
});

// @desc    Get all users
// @route   GET /api/users
// @access  Private/Admin
const getUsers = asyncHandler(async (req, res) => {
  const users = await User.find({});
  res.json(users);
});

// @desc    Delete user
// @route   DELETE /api/users/:id
// @access  Private/Admin
const deleteUser = asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id);

  if (user) {
    await user.remove();
    res.json({ message: 'User removed' });
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});

// @desc    Get user by ID
// @route   GET /api/users/:id
// @access  Private/Admin
const getUserById = asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id).select('-password');

  if (user) {
    res.json(user);
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});

// desc    Update user
// route   PUT /api/users/account
// access  Private/
// make it handle authorization token
```

```
const updateUser = asyncHandler(async (req, res) => {
  const user = await User.findById(req.user._id);
  const {
    name,
    email,
    isAdmin,
    isSeller,
    profilePic,
    firstName,
    lastName,
    phone,
    number,
    line1,
    line2,
    city,
    state,
    zip,
    country,
  } = req.body;

  if (user) {
    user.name = name || user.name;
    user.email = email || user.email;
    user.isAdmin = isAdmin || user.isAdmin;
    user.isSeller = isSeller || user.isSeller;
    user.profilePic = profilePic || user.profilePic;
    user.firstName = firstName || user.firstName;
    user.lastName = lastName || user.lastName;
    user.phone = phone || user.phone;
    user.shippingInfo.number = number || user.shippingInfo.number;
    user.shippingInfo.line1 = line1 || user.shippingInfo.line1;
    user.shippingInfo.line2 = line2 || user.shippingInfo.line2;
    user.shippingInfo.city = city || user.shippingInfo.city;
    user.shippingInfo.state = state || user.shippingInfo.state;
    user.shippingInfo.zip = zip || user.shippingInfo.zip;
    user.shippingInfo.country = country || user.shippingInfo.country;

    const updatedUser = await user.save();

    res.json({
      _id: updatedUser._id,
      name: updatedUser.name,
      email: updatedUser.email,
      profilePic: updatedUser.profilePic,
```

```
      isAdmin: updatedUser.isAdmin,
      isSeller: updatedUser.isSeller,
      firstName: updatedUser.firstName,
      lastName: updatedUser.lastName,
      phone: updatedUser.phone,
      number: updatedUser.shippingInfo.number,
      line1: updatedUser.shippingInfo.line1,
      line2: updatedUser.shippingInfo.line2,
      city: updatedUser.shippingInfo.city,
      state: updatedUser.shippingInfo.state,
      zip: updatedUser.shippingInfo.zip,
      country: updatedUser.shippingInfo.country,
      token: generateToken(updatedUser._id),
    });
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});

export {
  authUser,
  registerUser,
  getUsers,
  deleteUser,
  getUserById,
  updateUser,
};
```

Shop controller

```
import asyncHandler from 'express-async-handler';
import Shop from '../models/shopModel.js';

// @desc    Create a new shop
// @route   POST /api/shops
// @access  Private
const createShop = asyncHandler(async (req, res) => {
  const { shopName, shopEmail, shopAddress, shopPhone, shopDescription } = req.body;

  const shop = new Shop({
    user: req.user._id,
    shopDetails: {
      shopName,
      shopEmail,
      shopAddress,
      shopPhone,
      shopDescription
    },
  });

  const createdShop = await shop.save();
  res.status(201).json(createdShop);
});

// @desc    Fetch shop by id
// @route   GET /api/shops/:id
// @access  Public
const getshopById = asyncHandler(async (req, res) => {

  const shop = await Shop.findById(req.params.id);

  if (shop) {
    res.json(shop);
  } else {
    res.status(404);
    throw new Error('Shop not found');
  }
});

// @desc    Fetch all shops
// @route   GET /api/shops
// @access  Public

const getshops = asyncHandler(async (req, res) => {
```

```
  const shops = await Shop.find({});
  res.json(shops);
  console.log(shops);
});

// @desc    Fetch  shop to specific user
// @route   GET /api/products/shops/:id
// @access  Public
const getShopByUser = asyncHandler(async (req, res) => {
  const shop = await Shop.find({ user: req.params.id });

  res.json(shop);
});

// @desc    Update a shop
// @route   PUT /api/shops/:id
// @access  Private/Admin
const updateshop = asyncHandler(async (req, res) => {
  try{
    const id = req.params.id;
    const shop = req.body;
    await Shop.findByIdAndUpdate(id, shop);
    res.json({message: 'Shop updated'});
  } catch (error) {
    res.status(500).json({message: error.message});
  }
});

// @desc    Delete a shop
// @route   DELETE /api/shops/:id
// @access  Private/Admin
const deleteshop = asyncHandler(async (req, res) => {
  const shop = await Shop.findByIdAndDelete(req.params.id);

  if (shop) {
    res.json({ message: 'Shop removed' });
  } else {
    res.status(404);
    throw new Error('Shop not found');
  }
});

const getAllShops = asyncHandler(async (req, res) => {
  const shops = await Shop.find({})
```

```
  if (shops) {
    res.json(shops)
    console.log(shops)
  } else {
    res.status(404)
  }
});

export { getshopById, getshops, updateshop, createShop, getAllShops, deleteshop,
getShopByUser };
```

Config

```
import mongoose from 'mongoose';

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, {});
    // db connection

    console.log(
      `MongoDB database connection established successfully: ${conn.connection.host}`
        .cyan.underline
    );
  } catch (error) {
    console.log(`Error: ${error.message}`.red.underline.bold);
    process.exit(1);
  }
};

export default connectDB;
```

Middleware

```
import jwt from 'jsonwebtoken';
import asyncHandler from 'express-async-handler';
import User from '../models/userModel.js';

// to protect routes
const protect = asyncHandler(async (req, res, next) => {
  let token;

  if (
    req.headers.authorization &&
    req.headers.authorization.startsWith('Bearer')
  ) {
    try {
      token = req.headers.authorization.split(' ')[1];

      const decoded = jwt.verify(token, process.env.JWT_SECRET);

      req.user = await User.findById(decoded.id).select('-password');

      next();
    } catch (error) {
      console.error(error);
      res.status(401);
      throw new Error('Not authorized, token failed');
    }
  }

  if (!token) {
    res.status(401);
    throw new Error('Not authorized, no token');
  }
});

// to check admin
const admin = (req, res, next) => {
  if (req.user && req.user.isAdmin) {
    next();
  } else {
    res.status(401);
    throw new Error('Not authorized as an admin');
  }
};
```

```
// to check admin or seller
const adminSeller = (req, res, next) => {
  if ((req.user && req.user.isAdmin) || req.user.isSeller) {
    next();
  } else {
    res.status(401);
    throw new Error(
      `Not authorized as an ${req.user.isAdmin ? 'admin' : 'seller'}`
    );
  }
};

export { protect, admin, adminSeller };
```

Routes

User routes

```
import express from 'express';
import {
  authUser,
  registerUser,
  getUsers,
  deleteUser,
  getUserById,
  updateUser,
} from '../controllers/userController.js';
import { protect, admin } from '../middleware/authMiddleware.js';

const router = express.Router();

router.route('/').post(registerUser).get(protect, admin, getUsers);
router.post('/login', authUser);
router.route('/account').put(protect, updateUser);
router.route('/:id').delete(protect, deleteUser).get(protect, getUserById);

export default router;
```

Shop routes

```
import express from 'express';
import {
  getshopById,
  getshops,
  updateshop,
  createShop,
  deleteshop,
  getShopByUser,
  getAllShops,
} from '../controllers/shopcontroller.js';
import { protect, adminSeller } from '../middleware/authMiddleware.js';

const router = express.Router();


router.route('/').post(protect, adminSeller, createShop);
router.route('/').get(getshops);
router.route('/user/:id').get(getShopByUser);
router.route('/all').get(getAllShops);
router
  .route('/:id')
  .get(getshopById)
  .put(protect, adminSeller, updateshop)
  .delete(protect, adminSeller, deleteshop);

export default router;
```

Models

User model

```
import mongoose from 'mongoose';
import bcrypt from 'bcryptjs';

const userSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
      pattern : "^[a-zA-Z0-9_]*$",
    },
    email: {
      type: String,
      required: true,
      unique: true,
    },
    password: {
      type: String,
      required: true,
    },
    isAdmin: {
      type: Boolean,
      required: true,
      default: false,
    },
    isSeller: {
      type: Boolean,
      required: true,
      default: false,
    },
    profilePic: {
      type: String,
      default: 'https://images.unsplash.com/photo-1633332755192-
727a05c4013d?ixlib=rb-
4.0.3&ixid=MnwxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8&auto=format&fit=crop&w=580&q=
80',
    },
    firstName: {
      type: String,
      pattern : "^[a-zA-Z]*$",
      default: '',
    },
    lastName: {
      type: String,
      pattern : "^[a-zA-Z]*$",
```

```
  default: '',
},
phone: {
  type: String,
  pattern : "^[0-9]*$",
  default: '',
},
shippingInfo : {
  number : {
    type: String,
    pattern : "^[a-zA-Z0-9/-]*$",
    default: '',
  },
  line1 : {
    type: String,
    pattern : "^[a-zA-Z]*$",
    default: '',
  },
  line2 : {
    type: String,
    pattern : "^[a-zA-Z]*$",
    default: '',
  },
  city : {
    type: String,
    pattern : "^[a-zA-Z]*$",
    default: '',
  },
  state : {
    type: String,
    pattern : "^[a-zA-Z]*$",
    default: '',
  },
  country : {
    type: String,
    pattern : "^[a-zA-Z]*$",
    default: '',
  },
  zip : {
    type: String,
    pattern : "^[0-9]*$",
    default: '',
  },
},
```

```
  },
  {
    timestamps: true,
  }
);

userSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password);
};

userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) {
    next();
  }

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

const User = mongoose.model('User', userSchema);

export default User;
```

Shop model

```
import mongoose from 'mongoose';

const shopSchema = mongoose.Schema(
  {
    user: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'User',
    },
    shopDetails: {
      shopName: {
        type: String,
        required: true,
      },
      shopEmail: {
        type: String,
        required: true,
        unique: true,
      },
      shopAddress: {
        type: String,
        required: true,
      },
      shopPhone: {
        type: String,
        required: true,
        unique: true,
      },
      shopImage: {
        type: String,
        default: 'https://images.unsplash.com/photo-1472851294608-
062f824d29cc?ixlib=rb-
4.0.3&ixid=MnwxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8&auto=format&fit=crop&w=1170&q
=80',
      },
      shopDescription: {
        type: String,
        required: true,
      }
    },
  },
  {
    timestamps: true,
  }
```

```
);

const Shop = mongoose.model('Shop', shopSchema);

export default Shop;
```

Utils

```
import jwt from 'jsonwebtoken';

const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    expiresIn: '30d',
  });
};

export default generateToken;
```

Dockerfile

```
FROM node:alpine

# Create app directory
WORKDIR /app

# Install app dependencies
COPY package*.json ./

# If you are building your code for production
RUN npm install

# Bundle app source
COPY . .

# Expose port 9120
EXPOSE 9120
```

Kubernetes

user-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: user-service
  labels:
    app: user-service
spec:
  selector:
    app: user-service
  ports:
    - protocol: TCP
      port: 9120
      targetPort: 9120
  type: LoadBalancer
```

user-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  labels:
    app: user-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
        - name: user-service
          image: user-service:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 9120
```

Order Service

Controllers

Order Controller

```
import asyncHandler from 'express-async-handler';
import Order from '../models/orderModel.js';

// @desc    Create new order
// @route   POST /api/orders
// @access  Private
const addOrderItems = asyncHandler(async (req, res) => {
  const {
    orderItems,
    shippingDetails,
    phone,
    price,
    shippingMethod,
    commission,
    shippingPrice,
    totalPrice,
    seller,
  } = req.body;

  if (orderItems && orderItems.length === 0) {
    res.status(400);
    throw new Error('No order items');
    return;
  } else {
    const order = new Order({
      orderItems,
      user: req.user._id,
      shippingDetails,
      phone,
      price,
      shippingMethod,
      commission,
      shippingPrice,
      totalPrice,
      seller,
    });

    const createdOrder = await order.save();

    res.status(201).json(createdOrder);
  }
});

// @desc    Get order by ID
```

```
// @route   GET /api/orders/:id
// @access  Private
const getOrderById = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);

  if (order) {
    res.json(order);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

// @desc    Update order to paid
// @route   GET /api/orders/:id/pay
// @access  Private
const updateOrderToPaid = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);

  if (order) {
    order.isPaid = true;
    order.paidAt = Date.now();
    order.paymentResult = {
      id: req.body.id,
      status: req.body.status,
      update_time: req.body.update_time,
      email_address: req.body.payer.email_address,
    };

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

// @desc    Update order to delivered
// @route   GET /api/orders/:id/deliver
// @access  Private/Admin
const updateOrderToDelivered = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);
```

```
  if (order) {
    order.isDelivered = true;
    order.deliveredAt = Date.now();

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

// @desc    Get logged in user orders
// @route   GET /api/orders/myorders
// @access  Private
const getMyOrders = asyncHandler(async (req, res) => {
  const orders = await Order.find({ user: req.user._id });
  res.json(orders);
});

// @desc    Get all orders
// @route   GET /api/orders
// @access  Private/Admin
const getOrders = asyncHandler(async (req, res) => {
  const orders = await Order.find({});
  res.json(orders);
});

const updateOrderToConfirm = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);

  if (order) {
    order.isConfirmed = true;
    order.confirmedAt = Date.now();

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});
```

```
const updateOrderToDeliver = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);

  if (order) {
    order.isDelivered = true;
    order.deliveredAt = Date.now();

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

const updateOrderToReject = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);

  if (order) {
    order.isRejected = true;
    order.rejectedAt = Date.now();

    if (req.body.rejectReason) order.rejectReason = req.body.rejectReason;

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

const queryOrders = asyncHandler(async (req, res) => {
  const { start, end } = req.body;

  //get queried list of orders from db
  const queryData = await Order.find({
    createdAt: {
      $gte: new Date(start),
      $lt: new Date(end),
    },
```

```
  });

  if (queryData) {
    res.status(200).json(queryData);
  } else {
    res.status(404);
    throw new Error('No orders found');
  }
});

// get orders by user id and only output the order id and order status for each order

const getOrdersByUserId = asyncHandler(async (req, res) => {
  const orders = await Order.find({ user: req.params.id });
  if (!orders) {
    res.status(404);
    throw new Error('No orders found');
  }
  const orderIds = orders.map((order) => {
    return {
      _id: order._id,
      date: order.paidAt ? order.paidAt : order.createdAt,
      amount: order.totalPrice,
      isConfirmed: order.isConfirmed,
      isRejected: order.isRejected,
      isPaid: order.isPaid,
      isDelivered: order.isDelivered,
    };
  });
  res.status(200).json(orderIds);
});

//get products for seller to ship

const getOrdersforSeller = asyncHandler(async (req, res) => {
  const orders = await Order.find({
    orderItems: { $elemMatch: { seller: req.params.id } },
  });

  if (!orders) {
    res.status(404);
    throw new Error('No orders found');
  }
```

```
  return res.status(200).json(orders);
});

//update order to shipped
const updateOrderProductsToShipped = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);

  if (order) {
    order.isShipped = true;
    order.shippedAt = Date.now();

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

export {
  addOrderItems,
  getOrderById,
  updateOrderToPaid,
  updateOrderToDelivered,
  getMyOrders,
  getOrders,
  updateOrderToConfirm,
  updateOrderToReject,
  queryOrders,
  getOrdersByUserId,
  updateOrderToDeliver,
  getOrdersforSeller,
  updateOrderProductsToShipped,
};
```

Config

```
import mongoose from 'mongoose';

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, {});

    console.log(
      `MongoDB database connection established successfully: ${conn.connection.host}`
        .cyan.underline
    );
  } catch (error) {
    console.log(`Error: ${error.message}`.red.underline.bold);
    process.exit(1);
  }
};

export default connectDB;
```

Middleware

Auth Middleware

```
import jwt from 'jsonwebtoken';
import asyncHandler from 'express-async-handler';
import axios from 'axios';
import dotenv from 'dotenv/config';

const protect = asyncHandler(async (req, res, next) => {
  let token;

  if (
    req.headers.authorization &&
    req.headers.authorization.startsWith('Bearer')
  ) {
    try {
      token = req.headers.authorization.split(' ')[1];

      const USER = process.env.USER_PORT || 9120;

      const decoded = jwt.verify(token, process.env.JWT_SECRET);

      const response = await axios.get(
        `http://localhost:${USER}/api/users/${decoded.id}`,
        {
          headers: {
            Authorization: req.headers.authorization,
          },
        }
      );

      req.user = response.data;
      next();
    } catch (error) {
      console.error(error);
      res.status(401);
      throw new Error('Not authorized, token failed');
    }
  }

  if (!token) {
    res.status(401);
    throw new Error('Not authorized, no token');
  }
});

const admin = (req, res, next) => {
```

```
  if (req.user && req.user.isAdmin) {
    next();
  } else {
    res.status(401);
    throw new Error('Not authorized as an admin');
  }
};

export { protect, admin };
```

Error Middleware

```
const notFound = (req, res, next) => {
  const error = new Error('Not Found - ${req.originalUrl}');
  res.status(404);
  next(error);
};

const errorHandler = (err, req, res, next) => {
  const statusCode = res.statusCode === 200 ? 500 : res.statusCode;
  res.status(statusCode);
  res.json({
    message: err.message,
  });
};

export { notFound, errorHandler };
```

Routes

Order Routes

```
import express from 'express';
import {
  addOrderItems,
  getOrderById,
  updateOrderToPaid,
  getOrders,
  updateOrderToConfirm,
  updateOrderToReject,
  queryOrders,
  getOrdersByUserId,
  updateOrderToDeliver,
  getOrdersforSeller,
  updateOrderProductsToShipped
} from '../controllers/orderController.js';
import { protect, admin } from '../middleware/authMiddleware.js';

const router = express.Router();

router.route('/').post(protect, addOrderItems).get(getOrders);
router.route('/:id').get(getOrderById);
router.route('/:id/pay').put(protect, updateOrderToPaid);
router.route('/:id/confirm').put(updateOrderToConfirm);
router.route('/:id/reject').put(updateOrderToReject);
router.route('/:id/deliver').put(updateOrderToDeliver);
router.route('/query').post(queryOrders);
router.route('/user/:id').get(getOrdersByUserId);
router.route('/seller/products/:id').get(getOrdersforSeller);
router.route('/:id/shipped').put(updateOrderProductsToShipped);

export default router;
```

Models

```
import mongoose from 'mongoose';

const orderSchema = mongoose.Schema(
  {
    user: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'User',
    },
    orderItems: [
      {
        name: {
          type: String,
          required: true,
        },
        quantity: {
          type: Number,
          required: true,
        },
        image: {
          type: String,
          required: true,
          default: '',
        },
        price: {
          type: Number,
          required: true,
        },
        product: {
          type: mongoose.Schema.Types.ObjectId,
          required: true,
          ref: 'Product',
        },
        seller: {
          type: mongoose.Schema.Types.ObjectId,
          required: false,
          ref: 'User',
        },
      },
    ],
    shippingDetails: {
      firstName: {
        type: String,
```

```
    required: true,
  },
  lastName: {
    type: String,
    required: true,
  },
  address: {
    type: String,
    required: true,
  },
  apartment: {
    type: String,
    required: false,
  },
  state: {
    type: String,
    required: true,
  },
  city: {
    type: String,
    required: true,
  },
  country: {
    type: String,
    required: true,
  },
  postalCode: {
    type: String,
    required: true,
  },
  phone: {
    type: String,
    required: true,
  },
},
shippingMethod: {
  type: String,
  required: true,
  default: 'Standard',
},
paymentResult: {
  id: { type: String },
  status: { type: String },
  update_time: { type: String },
```

```
  email_address: { type: String },
},
commission: {
  type: Number,
  required: true,
  default: 0.0,
},
shippingPrice: {
  type: Number,
  required: true,
  default: 0.0,
},
totalPrice: {
  type: Number,
  required: true,
  default: 0.0,
},
isPaid: {
  type: Boolean,
  required: true,
  default: false,
},
paidAt: {
  type: Date,
},
isConfirmed: {
  type: Boolean,
  required: true,
  default: false,
},
confirmedAt: {
  type: Date,
},
isRejected: {
  type: Boolean,
  required: true,
  default: false,
},
rejectedAt: {
  type: Date,
},
rejectReason: {
  type: String,
  required: false,
```

```
    },
    isDelivered: {
      type: Boolean,
      required: true,
      default: false,
    },
    deliveredAt: {
      type: Date,
    },
    isShipped: {
      type: Boolean,
      required: true,
      default: false,
    },
    shippedAt: {
      type: Date,
    }
  },
  {
    timestamps: true,
  }
);

const Order = mongoose.model('Order', orderSchema);

export default Order;
```

Dockerfile

```
FROM node:alpine

# Create app directory
WORKDIR /app

# Install app dependencies
COPY package*.json ./

# If you are building your code for production
RUN npm install

# Bundle app source
COPY . .

# Expose port 9124
EXPOSE 9124
```

Kubernets

order-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: order-service
  labels:
    app: order-service
spec:
  selector:
    app: order-service
  ports:
    - protocol: TCP
      port: 9124
      targetPort: 9124
  type: LoadBalancer
```

order-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
  labels:
    app: order-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
        - name: order-service
          image: order-service:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 9124
```

Product Service

Controllers

```
// @desc    Fetch all products
// @route   GET /api/products
const getProducts = asyncHandler(async (req, res) => {
  const products = await Product.find({});
  res.json(products);
});

// @desc    Fetch single product
// @route   GET /api/products/:id
const getProductById = asyncHandler(async (req, res) => {
  const product = await Product.findById(req.params.id);
const getShopById = async () => {
    const response = await axios.get(
      `http://localhost:9120/api/shops/${product.shop}`);
    return response.data;};
  if (product) {
    const shop = await getShopById(product.shop._id);
    const productWithShop = {
      ...product.toObject(),
      shop: {
        shopDetails: shop.shopDetails,
      },};
    res.json(productWithShop);
  } else {
    res.status(404);
    throw new Error('Product not found');
  }});

// @desc    Fetch all products to specific user
// @route   GET /api/products/user/:id
const getProductsByUser = asyncHandler(async (req, res) => {
  const products = await Product.find({ user: req.params.id });
  res.json(products);
});
// @desc    Delete a product
// @route   DELETE /api/products/:id
const deleteProduct = asyncHandler(async (req, res) => {
  const product = await Product.findByIdAndDelete(req.params.id);

  if (product) {
    res.json({ message: 'Product removed' });
```

```
  } else {
    res.status(404);
    throw new Error('Product not found');
  }
});

// @desc    Create a product
// @route   POST /api/products
// @access  Private/Admin
const createProduct = asyncHandler(async (req, res) => {
  const {
    name,
    images,
    category,
    brand,
    detail,
    description,
    uses: [],
    ingredients,
    price,
    countInStock,
  } = req.body;

  const imageUrls = images.map((image) => ({ url: image.url }));

  const product = new Product({
    user: req.user._id,
    name,
    images: imageUrls,
    category,
    brand,
    detail,
    description,
    uses: [],
    ingredients,
    price,
    countInStock,
  });

  const createdProduct = await product.save();
  res.status(201).json(createdProduct);
});

// @desc    Update a product
```

```
// @route   PUT /api/products/:id
// @access  Private/Admin
const updateProduct = asyncHandler(async (req, res) => {
  const {
    name,category,brand,detail,description,
    uses,
    ingredients,
    reviews,
    rating,
    numReviews,
    price,
    countInStock,
  } = req.body;

  const product = await Product.findById(req.params.id);

  if (product) {
    product.name = name;
    product.category = category;
    product.brand = brand;
    product.detail = detail;
    product.description = description;
    product.uses = uses;
    product.ingredients = ingredients;
    product.reviews = reviews;
    product.rating = rating;
    product.numReviews = numReviews;
    product.price = price;
    product.countInStock = countInStock;

    const updatedProduct = await product.save();
    res.json(updatedProduct);
  } else {
    res.status(404);
    throw new Error('Product not found');
  }
});

// @desc    Get top rated products
// @route   GET /api/products/top
// @access  Public
const getTopProducts = asyncHandler(async (req, res) => {
  const products = await Product.find({}).sort({ rating: -1 }).limit(3);
res.json(products);
```

```
});

// get products by search query
// @desc    Fetch all products
// @route   GET /api/products/search/:query
const getProductsBySearch = asyncHandler(async (req, res) => {
  const { searchTerm } = req.params;
  const products = await Product.find({
    $or: [
      { name: { $regex: searchTerm, $options: 'i' } },
      { brand: { $regex: searchTerm, $options: 'i' } },
    ],});
console.log('products', products);
res.json(products);
});

// @desc    Create product review
// @route   POST /api/products/:id/reviews
const createProductReview = asyncHandler(async (req, res) => {
  const { rating, comment } = req.body;

  const product = await Product.findById(req.params.id);

  if (product) {
    const alreadyReviewed = product.reviews.find(
      (r) => r.user.toString() === req.user._id.toString()
    );

    if (alreadyReviewed) {
      res.status(400);
      throw new Error('Product already reviewed');
    }

    const review = {
      name: req.user.name,
      rating: Number(rating),
      comment,
      user: req.user._id,
    };

    product.reviews.push(review);

    product.numReviews = product.reviews.length;
```

```
    product.rating =
      product.reviews.reduce((acc, item) => item.rating + acc, 0) /
      product.reviews.length;

    await product.save();
    res.status(201).json({ message: 'Review added' });
  } else {
    res.status(404);
    throw new Error('Product not found');
  }
});
```

Middleware

```
const notFound = (req, res, next) => {
  const error = new Error('Not Found - ${req.originalUrl}');
  res.status(404);
  next(error);
};

const errorHandler = (err, req, res, next) => {
  const statusCode = res.statusCode === 200 ? 500 : res.statusCode;
  res.status(statusCode);
  res.json({
    message: err.message,
  });
};

export { notFound, errorHandler };
```

Routes

```
const router = express.Router();
router.route('/').get(getProducts).post(protect, adminSeller, createProduct);
router.get('/user/:id', getProductsByUser);
router.get('/top', getTopProducts);
router.get('/search/:searchTerm', getProductsBySearch);
router.route('/:id/reviews').post(protect, createProductReview);
router
  .route('/:id')
  .get(getProductById)
  .delete(protect, adminSeller, deleteProduct)
  .put(protect, adminSeller, updateProduct);
export default router;
```

Models

```
const productSchema = mongoose.Schema(
  {
    user: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'User',
    },
    name: {
      type: String,
      required: true,
    },
    images: [imageSchema],
    category: {
      type: String,
      required: true,
      default: '',
    },
    brand: {
      type: String,
      required: true,
      default: '',
    },
    detail: {
      type: String,
      required: true,
      default: '',
    },
    description: {
      type: String,
```

```
      required: true,
      default: '',
    },
    uses: [String],
    ingredients: {
      type: String,
      required: true,
      default: '',
    },
    reviews: [reviewSchema],
    rating: {
      type: Number,
      required: true,
      default: 0,
    },
    numReviews: {
      type: Number,
      required: true,
      default: 0,
    },
    price: {
      type: Number,
      required: true,
      default: 0,
    },
    countInStock: {
      type: Number,
      required: true,
      default: 0,
    },
    shop: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'Shop',
      default: '64458bd232c4bea2a2cbd9ef',
    },
  },
  {
    timestamps: true,
  }
);
```

Docker File

```
FROM node:alpine

# Create app directory
WORKDIR /app

# Install app dependencies
COPY package*.json ./

# If you are building your code for production
RUN npm install

# Bundle app source
COPY . .

# Expose port 9121
EXPOSE 9121
```

Admin Service

Controllers

1. ConfigServices.js

```
const Config = require("../models/ApplicationConfig");

const DOC_ID = "6431b70d546809a792408963"

const getCommission = async (req, res) => {
  const commission = await Config.findById(DOC_ID);

  if(!commission) {
    res.status(400).json({msg: 'No commission found'});
} else {
    res.status(200).json(commission);
}
};

const updateCommission = async (req, res) => {
    const {commission} = req.body;
    const updatedCommission = await Config.findByIdAndUpdate(
        DOC_ID,
        commission,
        {
            new: true,
            runValidators: true
        }
    );

    if(!updatedCommission) {
        res.status(400).json({msg: 'Update failed!'});
    } else {
        res.status(200).json(updatedCommission);
    }
};

module.exports = {
    getCommission,
    updateCommission
};
```

2. OrderService.js

```
const getOrderById = async (id) => {};
const getOrders = async () => {

    //call order service
    const orders = await axios.get('http://order-service:9124/api/orders');

    if (orders) {
        return orders.data;
    } else {
        return null;
    }

};

const updateOrder = async (order) => {};

const queryOrderAsync = async (dateRange) => {
    //get queried list of orders from db
    //correctly format the query
    let query = {};

    if(dateRange) {
        query = {
            start: moment(dateRange.start).startOf('day').toDate(),
            end: moment(dateRange.end).endOf('day').toDate()
        }
    } else if(query.createdAt) {
        query = {
            start: moment(createdAt).startOf('day').toDate(),
            end: moment(createdAt).endOf('day').toDate()
        }
    }

    const queryData = await axios.post('http://order-service:9124/api/orders/query',
query);

    if (queryData) {
        return queryData.data;
    } else {
        return null;
    }
};

const queryOrders = async (req, res) => {
```

```
    const { query } = req.body;

    //correctly format the query
    let newQuery = {};

    if(query.dateRange) {
        newQuery = {
            start: moment(query.dateRange.start).startOf('day').toDate(),
            end: moment(query.dateRange.end).endOf('day').toDate()
        }
    } else if(query.createdAt) {
        newQuery = {
            start: moment(query.createdAt).startOf('day').toDate(),
            end: moment(query.createdAt).endOf('day').toDate()
        }
    }

    //call order service
    const queryData = await axios.post('http://order-service:9124/api/orders/query',
{query: newQuery});

    if(queryData) {
        res.status(200).json(queryData);
    } else {
        res.status(404);
        throw new Error('No orders found');
    }
};

module.exports = {
    getOrderById,
    getOrders,
    updateOrder,
    queryOrders,
    queryOrderAsync
};
```

3. ProductServices.js

```
const getProducts = async (req, res) => {};

module.exports = {
    getProducts
};
```

4. StatisticsServices.js

```
const {
    DailySalesStat,
    MonthlySalesStat,
    YearlySalesStat,
} = require("../models/SalesStatModel");
const OrderStat = require("../models/OrderStatModel");
const {queryOrderAsync, getOrders} = require("../controllers/OrderServices");

const getDailySalesStats = async (req, res) => {
    const { year, month } = req.params;
    try {
        const stats = await DailySalesStat.find({ year, month });
        res.status(200).json(stats);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};

const calculateDailySales = async (req, res) => {
    const { year, month } = req.body;

    try {
        const orders = await queryOrderAsync({
            start: new Date(year, month - 1, 1),
            end: new Date(year, month, 0),
        });

        let dailySales =
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];

        for (let i = 0; i < orders.length; i++) {
            const order = orders[i];
            const orderDate = new Date(order.createdAt);
            dailySales[orderDate.getDate() - 1] += order.totalPrice;
        }

        const saveStats = await DailySalesStat.find({ year, month });

        if (saveStats.length > 0) {
            saveStats[0].sales = dailySales;
            await saveStats[0].save();
        } else {
            const stats = new DailySalesStat({
                year,
```

```
                month,
                sales: dailySales,
            });
            await stats.save();
        }

        res.json({
            message: "Daily sales calculated successfully",
            dailySales,
        });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }

};

const getMonthlySalesStats = async (req, res) => {
    const { year } = req.params;
    try {
        const stats = await MonthlySalesStat.find({ year });
        res.status(200).json(stats);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};

const calculateMonthlySales = async (req, res) => {
    const { year } = req.body;

    try {
        const orders = await queryOrderAsync({
            start: new Date(year, 0, 1),
            end: new Date(year, 11, 31),
        });

        const monthlySales = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

        for (let i = 0; i < orders.length; i++) {
            const order = orders[i];
            const orderDate = new Date(order.createdAt);
            monthlySales[orderDate.getMonth()] += order.totalPrice;
        }

        const saveStats = await MonthlySalesStat.find({ year });
```

```
        if (saveStats.length > 0) {
            saveStats[0].sales = monthlySales;
            await saveStats[0].save();
        } else {
            const stats = new MonthlySalesStat({
                year,
                sales: monthlySales,
            });
            await stats.save();
        }

        res.json({
            message: "Monthly sales calculated successfully",
            monthlySales,
        });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};

const getYearlySalesStats = async (req, res) => {
    try {
        const stats = await YearlySalesStat.find();
        res.status(200).json(stats);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};

const calculateYearlySales = async (req, res) => {

    try {
        const orders = await getOrders();

        //calculate sales figures for each year
        let yearlyStats = {};  // {year: totalSales}

        for (let i = 0; i < orders.length; i++) {
            const order = orders[i];
            const orderDate = new Date(order.createdAt);
            const year = orderDate.getFullYear();
            if (yearlyStats[year]) {
                yearlyStats[year] += order.totalPrice;
```

```
    } else {
        yearlyStats[year] = order.totalPrice;
    }
}

//save sales figures to database
const saveStats = await YearlySalesStat.find();

if (saveStats.length > 0) { //yearly sales stats already exist in database

    for(let [year, totalSales] of Object.entries(yearlyStats)) {
        for(let i = 0; i <= saveStats.length; i++){
            //if year already exists in database, update the sales figure
            if(i < saveStats.length){
                if(saveStats[i].year === year){
                    console.log("updating entry")
                    saveStats[i].sales = totalSales;
                    await saveStats[i].save();
                    break;
                }
            }

            //if year does not exist in database, create a new entry
            if(i === saveStats.length){
                const newStat = new YearlySalesStat({
                    year,
                    sales: totalSales,
                });
                newStat.save();
                break;
            }
        }
    };

} else { //no yearly sales stats in database
    //create new entries for each year
    for(let [year, totalSales] in yearlyStats) {
        const newStat = new YearlySalesStat({
            year,
            sales: totalSales,
        });
        newStat.save();
    };
}
```

```
        res.status(200).json({
            message: "Yearly sales calculated successfully",
            yearlyStats,
        });

    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};

const calculateOrderStats = async (req, res) => {
    const { year, month } = req.body;

    try {
        const orders = await queryOrderAsync({
            start: new Date(year, month - 1, 1),
            end: new Date(year, month, 0),
        });

        const orderStats = {
            year,
            month,
            stats : {
                pending: 0,
                unpaid: 0,
                confirmed: 0,
                rejected: 0,
                delivered: 0
            }
        };

        for (let i = 0; i < orders.length; i++) {
            const order = orders[i];

            let orderStatus = "unpaid";
            if (order.isPaid) {
                orderStatus = "pending";
            } else if (order.isDelivered) {
                orderStatus = "delivered";
            } else if (order.isConfirmed) {
                orderStatus = "confirmed";
            } else if (order.isRejected) {
                orderStatus = "rejected";
```

```
            }

            orderStats.stats[orderStatus] += 1;
        }

        //save stats to database
        const orderStat = await OrderStat.find({ year, month });

        if(orderStat.length > 0) {//if stats already exist in database, update them
            orderStat[0].stats = orderStats.stats;
            await orderStat[0].save();
        } else {//if stats do not exist in database, create new entry
            const newStat = new OrderStat(orderStats);
            await newStat.save();
        }

        res.status(200).json({
            message: "Order stats calculated successfully",
            orderStats,
        });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
}

const getOrderStats = async (req, res) => {
    const { year, month } = req.params;

    try {
        const stat = await OrderStat.find({ year, month });
        res.status(200).json(stat);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};

module.exports = {
    getDailySalesStats,
    getMonthlySalesStats,
    getYearlySalesStats,
    calculateYearlySales,
    calculateMonthlySales,
    calculateDailySales,
    getOrderStats,
```

```
    calculateOrderStats,
};
```

5. UserServices.js

```
const mongoose = require('mongoose');

const getUserbyId = async (req, res) => {};

const updateUser = async (req, res) => {};

const getUsers = async (req, res) => {};

module.exports = {
    getUserbyId,
    updateUser,
    getUsers
};
```

Middleware

Routes

1. AdminConfig.js

```
const router = require('express').Router();
const {
    getCommission,
    updateCommission,
} = require("../controllers/ConfigServices");

router.route('/commission')
    .get(getCommission)
    .post(updateCommission);

module.exports = router;
```

2. AdminOrderList.js

```
const router = require('express').Router();
const {
    getOrders,
    queryOrders,
    getOrderById,
    updateOrder,
} = require("../controllers/OrderServices");

router.route('/order-list').get(getOrders);

router.route('/get-order').post(getOrderById);
router.route('/update-order').post(updateOrder);
router.route('/query-order').post(queryOrders);

module.exports = router;
```

3. AdminProductLIst.js

```
const router = require('express').Router();
const {getProducts} = require('../controllers/ProductServices');

router.route('/product-list').get(getProducts);

module.exports = router;
```

4. AdminUserList.js

```
const router = require('express').Router();
const {
    getUserbyId,
    updateUser,
    getUsers,
} = require("../controllers/UserServices");

router.route('/user-list').get(getUsers);

router.route('/get-user').post(getUserbyId);
router.route('/update-user').post(updateUser);

module.exports = router;
```

5. Statistic.js

```
const router = require('express').Router();
const {
    calculateDailySales,
    calculateMonthlySales,
    calculateYearlySales,
    getDailySalesStats,
    getMonthlySalesStats,
    getYearlySalesStats,
    getOrderStats,
    calculateOrderStats,
} = require('../controllers/StatisticServices');

router.route('/sales/daily/:year/:month').get(getDailySalesStats);
router.route('/sales/daily').post(calculateDailySales);

router.route('/sales/monthly').post(calculateMonthlySales);
router.route('/sales/monthly/:year').get(getMonthlySalesStats);

router.route('/sales/yearly')
    .get(getYearlySalesStats)
    .post(calculateYearlySales);

router.route('/orders/monthly').post(calculateOrderStats);
router.route('/orders/:year/:month').get(getOrderStats);

module.exports = router;
```

Models

1. ApplicationConfig.js

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

const configSchema = new mongoose.Schema({
    commission: {
        type: Number,
        required: true
    }
});

const Config = mongoose.model("Config", configSchema);

module.exports = Config;
```

2. OrderStatModel.js

```
const mongoose = require('mongoose');

const OrderStatSchema = new mongoose.Schema({
    year: {
        type: Number,
        required: true,
    },
    month: {
        type: Number,
        required: true,
    },
    stats: {
        pending: {
            type: Number,
            required: true,
        },
        unpaid: {
            type: Number,
            required: true,
        },
        confirmed: {
            type: Number,
            required: true,
        },
        rejected: {
            type: Number,
            required: true,
        },
        delivered: {
            type: Number,
            required: true,
        }
    }
});

const OrderStat = mongoose.model('OrderStat', OrderStatSchema);

module.exports = OrderStat;
```

3. SalesStatModel.js

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

const dailyStatsSchema = new mongoose.Schema({
    year: {
        type: Number,
        required: true
    },
    month: {
        type: Number,
        required: true
    },
    sales: {
        type: Array,
        required: false,
        default: []
    }
});
```

```
const monthlyStatsSchema = new mongoose.Schema({
    year: {
        type: Number,
        required: true
    },
    sales: {
        type: Array,
        required: false,
        default: [0,0,0,0,0,0,0,0,0,0,0,0]
    }
});

const yearlyStatsSchema = new mongoose.Schema({
    year: {
        type: Number,
        required: true
    },
    sales: {
        type: Number,
        required: false,
        default: 0
    }
});

const DailySalesStat = mongoose.model("dailyStatistic", dailyStatsSchema);
const MonthlySalesStat = mongoose.model("monthlyStatistic", monthlyStatsSchema);
const YearlySalesStat = mongoose.model("yearlyStatistic", yearlyStatsSchema);

module.exports = {
    DailySalesStat,
    MonthlySalesStat,
    YearlySalesStat
};
```

Docker File

```
FROM node:alpine

WORKDIR /app

COPY ./package*.json ./
RUN npm install

COPY . .

EXPOSE 9122
```

SMS Service

Controllers

```
import axios from 'axios';

// dotenv.config({ path: findConfig('.env.sms') });
// @desc Send a SMS
// @route POST /api/sms
// @access Public
const sendSms = async (req, res) => {
  try {
    const { to, message } = req.body;

    const { data } = await axios.post(

`https://app.notify.lk/api/v1/send?user_id=${process.env.user_id}&api_key=${process.e
nv.SMS_API_KEY}&sender_id=${process.env.sender_id}&to=${to}&message=${message}`
    );

    // Return a success response to the client
    res.status(200).json({ message: data.message });
  } catch (error) {
    // Return an error response to the client
    res.status(500).json({ message: error.message });
  }
};

export { sendSms };
```

Routes

```
import express from 'express';
import { sendSms } from '../controllers/smsController.js';

const router = express.Router();

router.post('/send', sendSms);

export default router;
```

Dockerfile

```
FROM node:alpine


# Create app directory

WORKDIR /app
```

```
# Install app dependencies
COPY package*.json ./

# If you are building your code for production
RUN npm install

# Bundle app source
COPY . .

# Expose port 9125
EXPOSE 9125
```

Kubernets

sms-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: sms-service
  labels:
    app: sms-service
spec:
  selector:
    app: sms-service
  ports:
    - protocol: TCP
      port: 9125
      targetPort: 9125
  type: LoadBalancer
```

sms-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sms-service
  labels:
    app: sms-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sms-service
  template:
    metadata:
      labels:
        app: sms-service
    spec:
      containers:
        - name: sms-service
          image: sms-service:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 9125
```

Email Service

Controllers

Email Service

```
const nodemailer = require('nodemailer');
const getAccount = require('../config/HostAccount');

let transporter;

const getGmailTransporter = async () => {

    if(!transporter){
        const account = await getAccount();

        const config = {
            service: 'gmail',
            auth: {
                user: account.user,
                pass: account.pass
            }
        }

        transporter = nodemailer.createTransport(config);
    }

    return transporter;
};

module.exports = getGmailTransporter;
```

Mail Generator

```
const Mailgen = require('mailgen');

const mailGenerator = new Mailgen({
    theme: 'default',
    product: {
        name: 'RedStream Herbal',
        link: 'https://redstream.sliit.com/',//TODO: Change this to the actual link
    },
});

module.exports = mailGenerator;
```

Config

```
//RedStream email host account credentials
let account;

const getAccount = async () => {
    if (!account) {
        // const nodemailer = require('nodemailer');
        // account = await nodemailer.createTestAccount();

        account = {
            user: "redstream.sliit@gmail.com",
            pass: "amkcomtpghwrabyt"
        };
    }

    return account;
};

module.exports = getAccount;
```

Routes

```
const router = require('express').Router();
const getGmailTransporter = require('../controllers/EmailService');
const getAccount = require('../config/HostAccount');
const mailGenerator = require('../controllers/MailGenerator');

router.route('/send').post(async (req, res) => {
    // Get email data from request body
    const { to, subject, mail } = req.body;
    const account = await getAccount();
    const transporter = await getGmailTransporter();

    //prepare content for Mailgen
    const data = {
        body: {
            name: mail.header,
            intro: mail.intro,
            outro: mail.outro
        }
    };

    if(mail.tableData) {
        data.body.table = {
            data: mail.tableData
        }
    }

    if(mail.action) {
        data.body.action = {
            instructions: mail.action.instructions,
            button: {
                text: mail.action.buttonText,
                link: mail.action.buttonLink
            }
        }
    }

    //generate mail html
    const html = mailGenerator.generate(data);

    // Send email
    const result = await transporter.sendMail(
        {
            from: `"RedStream" <${account.user}>`,
```

```
            to,
            subject,
            html,
        },
        (err, info) => {
            if (err) {
                res.status(400).send({ error: err, msg: "Email not sent!" });
            } else {
                res.status(200).send({ msg: "Email sent!", info });
            }
        }
    );

});

module.exports = router;
```

Dockerfile

```
FROM node:alpine

WORKDIR /app

COPY ./package*.json ./
RUN npm install

COPY . .

EXPOSE 9123
```

Kubernetes

email-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: email-service
  labels:
    app: email-service
spec:
  selector:
    app: email-service
  ports:
    - protocol: TCP
      port: 9123
      targetPort: 9123
  type: LoadBalancer
```

email-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: email-service
  labels:
    app: email-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: email-service
  template:
    metadata:
      labels:
        app: email-service
    spec:
      containers:
        - name: email-service
          image: email-service:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 9123
```