Sri Lanka Institute of Information Technology

Distributed Systems (SE3020) – Semester 1 2023

Assignment 1 – Programming project

RareAyur – The collaborative shopping platform for Ayurvedic medicines and supplements

Project Report

Group details

Batch: Y3.S1.WE.SE.01

Group number: GRP-05

Description	Student ID	Name
Group Leader	IT21028878	Kulasekara D.A.M. N
Member 2	IT21034404	Nipun P.G. I
Member 3	IT21039140	Dewasurendra S. V
Member 4	IT21071034	Wanniarachchi T. T

Table of Contents

1) Introduction

1. Background

In recent years, interest in natural health and wellness products such as Ayurvedic and medicinal products and supplements has increased. In response to this development, the MERN stack is being used to create the RareAyur collaborative shopping platform.

For clients interested in Ayurvedic and herbal medications and supplements, RareAyur is trying to offer a convenient and user-friendly purchase experience. The platform will provide a massive selection of products from different brands. The ability for people to engage in the items they are interested in is one of RareAyur's key characteristics.

For this project, the MERN stack was selected because of its flexibility, scalability, and simplicity. The stack consists of Node.js for the server-side runtime environment, Express for the server-side framework, React for the client-side framework, and MongoDB for the database. Developers may construct web apps that are quick, reliable, and highly responsive using this technology combo.

The RareAyur platform is being created using both the MERN stack and microservice architecture, which enables the creation of loosely connected services that can be created, used, and scaled independently.

The platform is being set up with Kubernetes and Docker containers to accommodate this design. While Kubernetes handles the setup and management of these containers, Docker is used for building and operating each of the microservices in isolated containers.

A few benefits of using microservices and containerization are increased reliability, scalability, and fault tolerance. The program can be divided into smaller, easier-to-manage services, allowing developers to work more quickly and release upgrades and new features without affecting other system components.

Finally, RareAyur will give customers a useful tool for locating and buying natural wellness and health products. By giving users a forum to share knowledge and decide intelligently on the things they are interested in, collaborative tools will improve the shopping experience.

2. Purpose

The purpose of RareAyur is to provide customers looking for Ayurvedic and herbal supplements and drugs with a collaborative buying platform. RareAyur seeks to offer a convenient and user-friendly purchasing experience by offering a variety of products from various companies.

To achieve this, the MERN stack, which offers scalability, adaptability, and simplicity, is being used in the development of RareAyur. Microservice architecture is also being used in the platform's design, and Kubernetes and Docker containers are being used for deployment; these technologies provide greater stability, scaling, and fault tolerance.

3. System overview

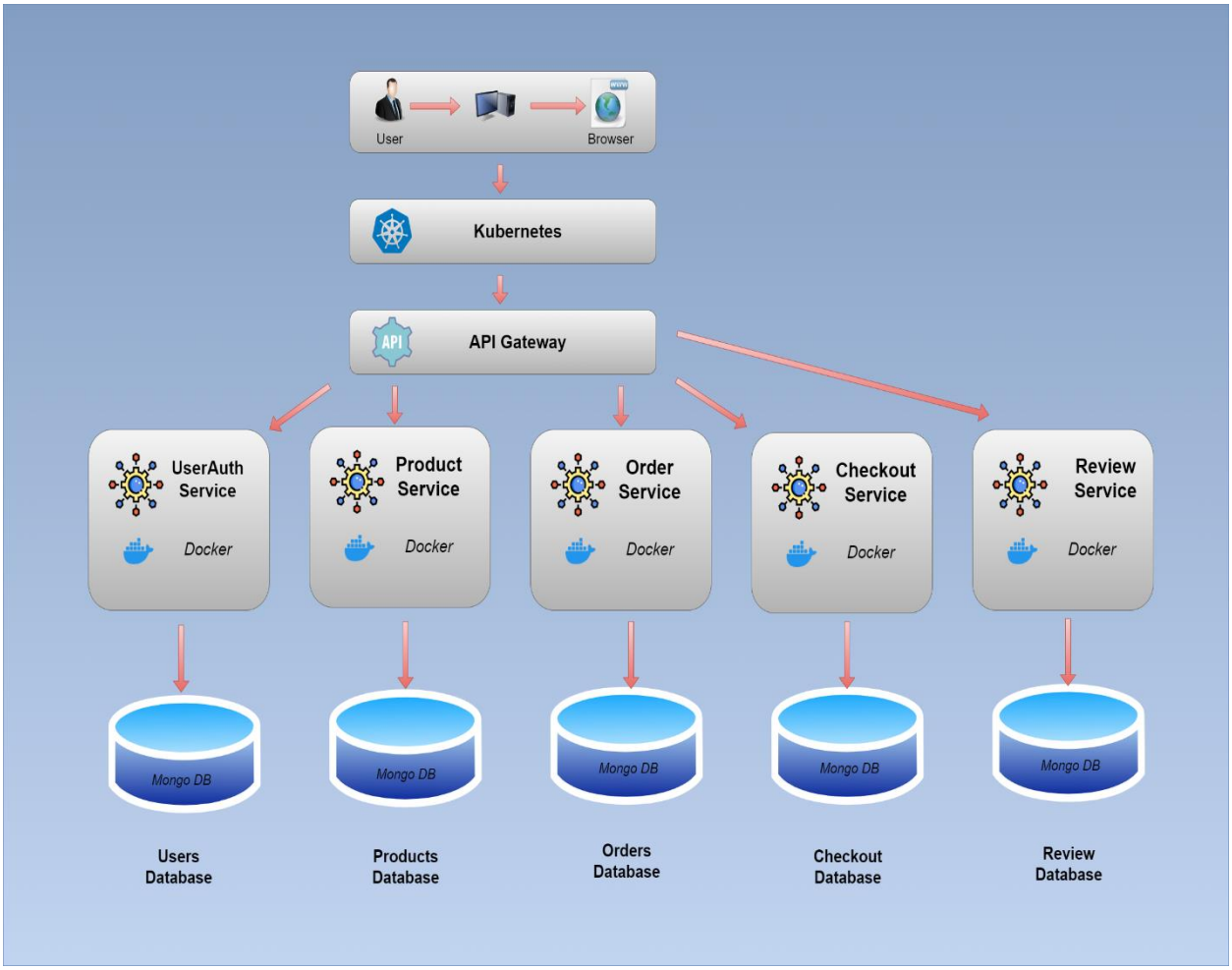


*Figure **Error! Use the Home tab to apply 0 to the text that you want to appear here.**.1.1 – High Level System Architecture*

The diagram displays the various system parts, such as the user's web browser that they use to access the platform and the load balancer (Kubernetes) that divides up the incoming traffic among several instances of the API gateway. Requests are forwarded to the relevant microservices by the API gateway.

User data is managed by the user service, including registrations and profile management. Products, along with their descriptions, pictures, and inventory data, are managed by the product service. The management of orders, including handling of payments, is under the control of the order service. Payment data managed by the payment service. Review data managed by the Review service. Using Docker to containerize each microservice and Kubernetes to manage them all.

Each microservice has its own MongoDB.

The front end is likewise deployed in the Kubernetes cluster and containerized using Docker. The front-end and back-end are developed using the MERN stack, which consists of React as the user interface library, Express.js as the web application framework for Node.js, and MongoDB as the database.

Overall, this graphic gives a clear picture of the high-level architecture of the platform, highlighting the various microservices and their interconnections as well as the building and deployment tools, like Docker and Kubernetes, that were utilized to create the system. Each microservice has its own database, which ensures data isolation and enables independent scaling of each service.

2) Overall Description

1. Product Perspective

Ayurvedic and Herbal medications and supplements can be purchased at RearAyur, a collaborative shopping platform created to offer clients a quick and personalized buying experience. UserAuth, Product, Order, and Checkout are the platform's four primary services, which together make up its microservice architecture. The MERN stack is used to build the services, while Docker and Kubernetes are used to deploy them.

Market Analysis

The market for Ayurvedic and Herbal medications and supplements is expanding quickly due to the rising desire for herbal and natural alternative healthcare options. Health-conscious consumers looking for a variety of Ayurvedic and Herbal medications and supplements make up the intended audience for RearAyur.

Competitor Analysis

Other e-commerce sites and physical pharmacies that provide Ayurvedic and Herbal medications and supplements are RearAyur's primary rivals. RearAyur, on the other hand, sets itself apart by providing a customized shopping experience, with an emphasis on making product recommendations to customers that are specific to their health requirements and tastes.

Stakeholders

Buyers, who will use the platform to browse and buy products, product suppliers(sellers), who will use the system to sell their products, and internal stakeholders, such as the development team, marketing team, and operational team, make up the main stakeholders for RearAyur.

Dependencies

The MERN stack for development, Docker for containerization, and Kubernetes for orchestration are just a few of the technologies and services that RearAyur depends on to run. The platform also uses third-party APIs for shipping and processing payments.

Strategic Fit:

RearAyur supports the organization's overall goal by offering clients a distinctive and cutting-edge platform to buy Ayurvedic and Herbal medications and supplements. The platform's emphasis on natural health and personalized shopping experience is consistent with the goals and values of the company.

Ayurvedic and Herbal medications and supplements can be purchased through RearAyur's creative and distinctive platform, which offers customers a quick and personalized shopping experience. Its microservice architecture, usage of Docker and Kubernetes, and emphasis on customization and natural health set it apart from rivals in the market while also making it scalable and simple to operate.

2. Product Features

UserAuth Service:

User registration: Allows new users to create an account on the platform by providing their personal details and creating a password.

User authentication: Validates user credentials to ensure that only authorized users can access the platform.

User profile management: Allows users to manage their personal information.

Role-based access control: Enables administrators to assign different levels of access to users based on their roles and responsibilities.

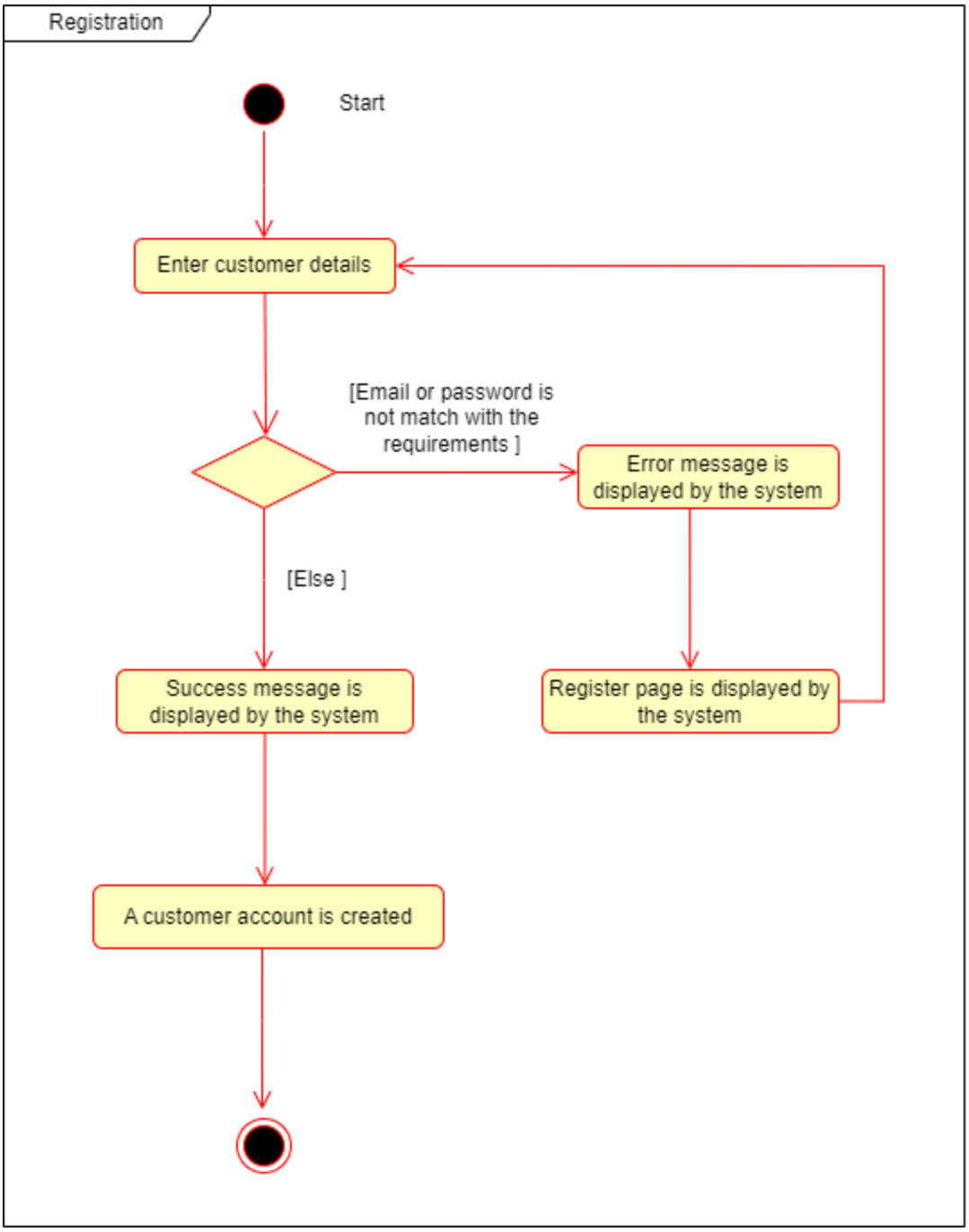


Figure ***Error! Use the Home tab to apply 0 to the text that you want to appear here.****..2.1 – Activity Diagram - Register*

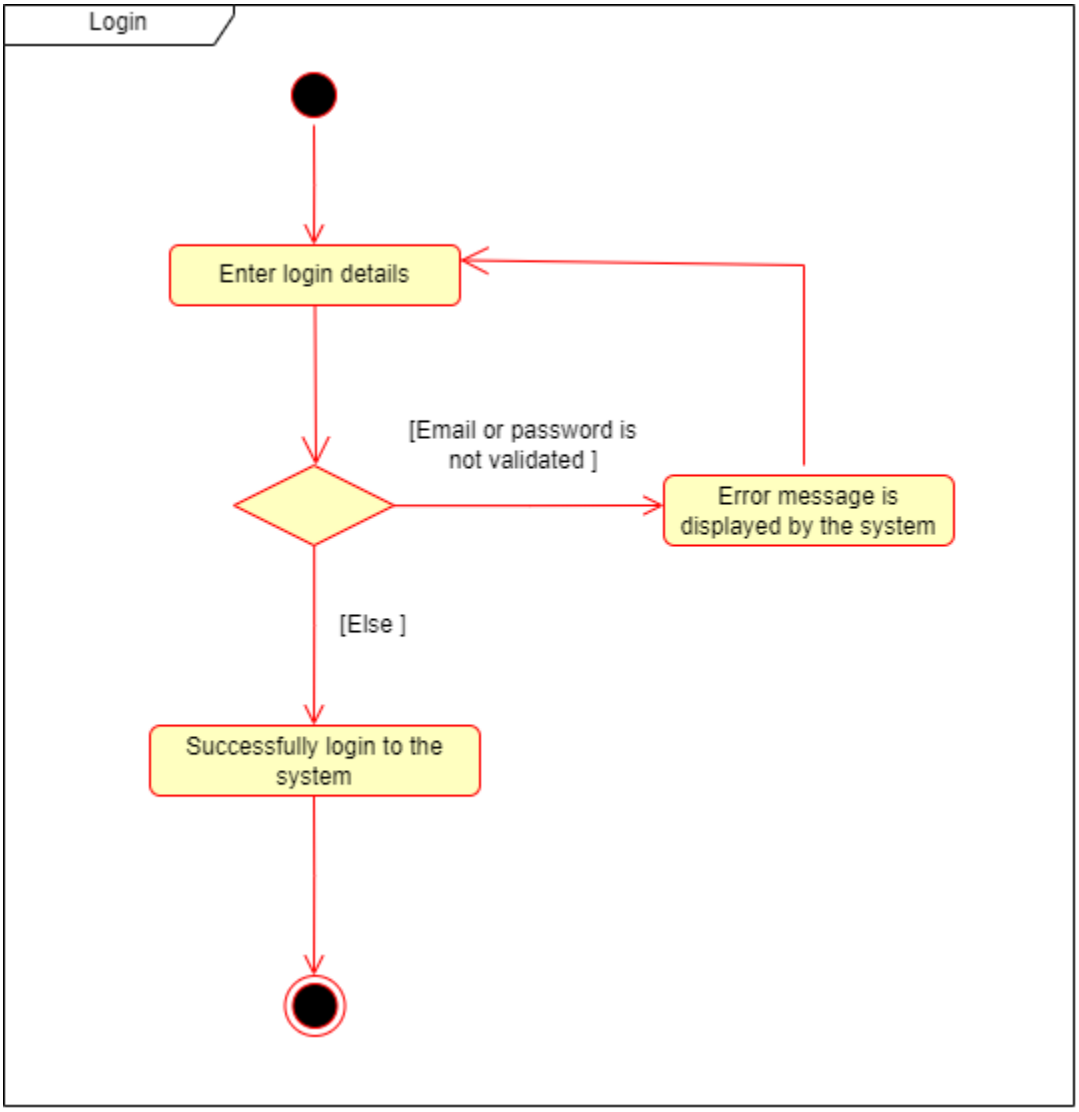


Figure 2.2 – Activity Diagram - Login

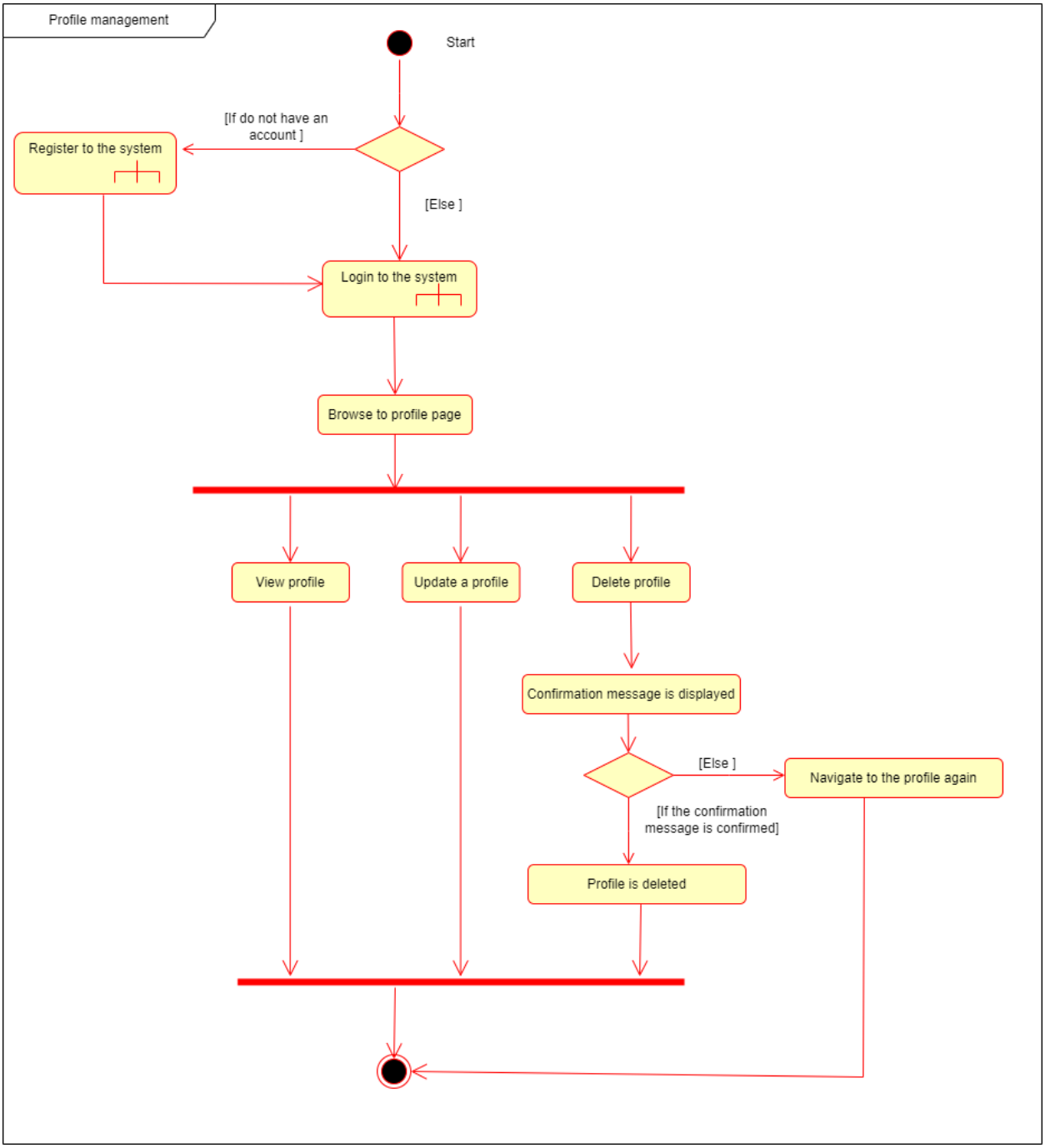


Figure 2.3 – Activity Diagram – Profile Management

Product Service:

Product catalog management: Allows administrators to view, remove products in the platform's catalog, including their name, description, price, and images. Add, remove update view sellers.

Product catalog management: Allows administrators to view, remove products in the platform's catalog, including their name, description, price, and images.

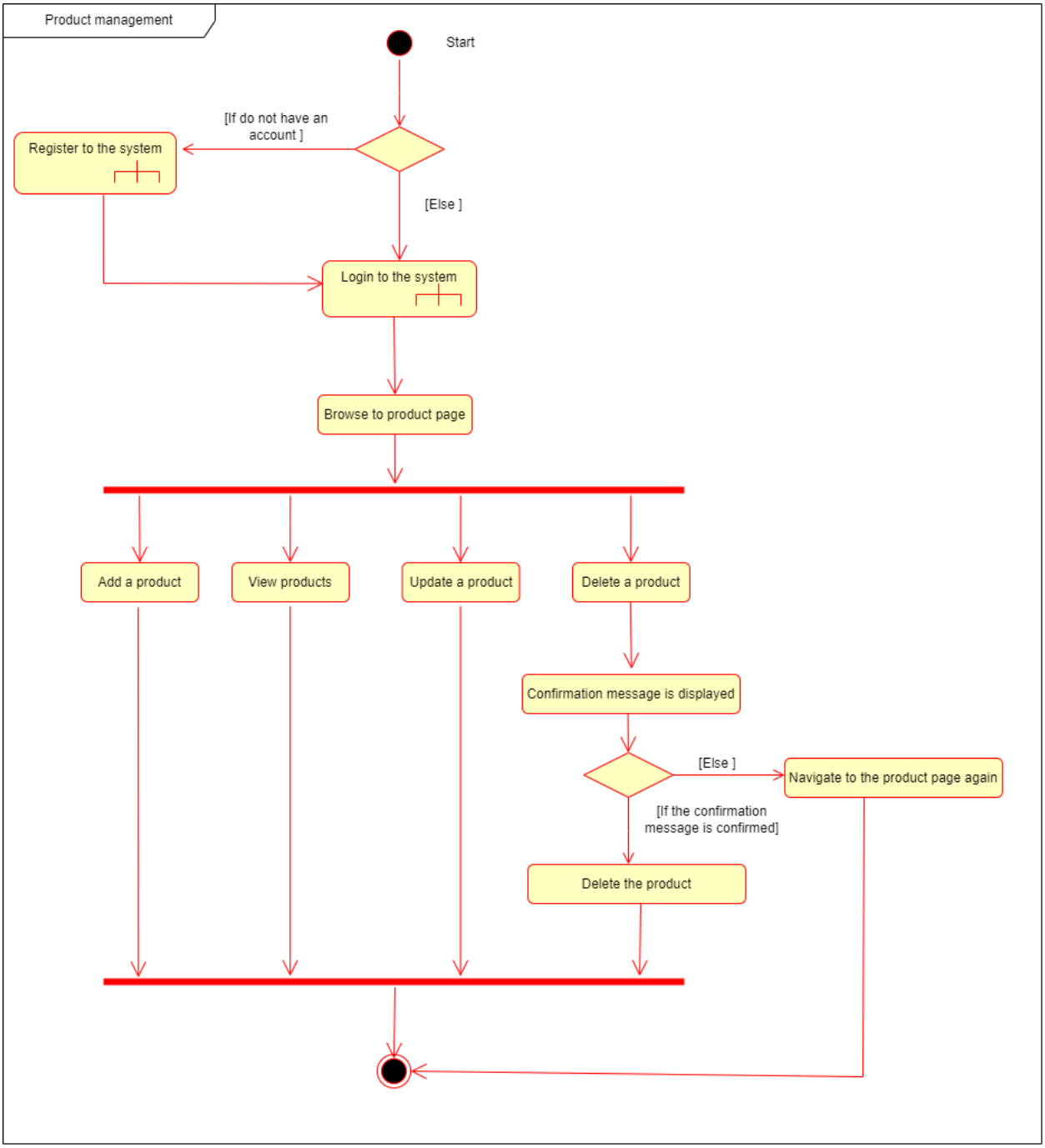


Figure 2.4 – Activity Diagram - Register

Order Service:

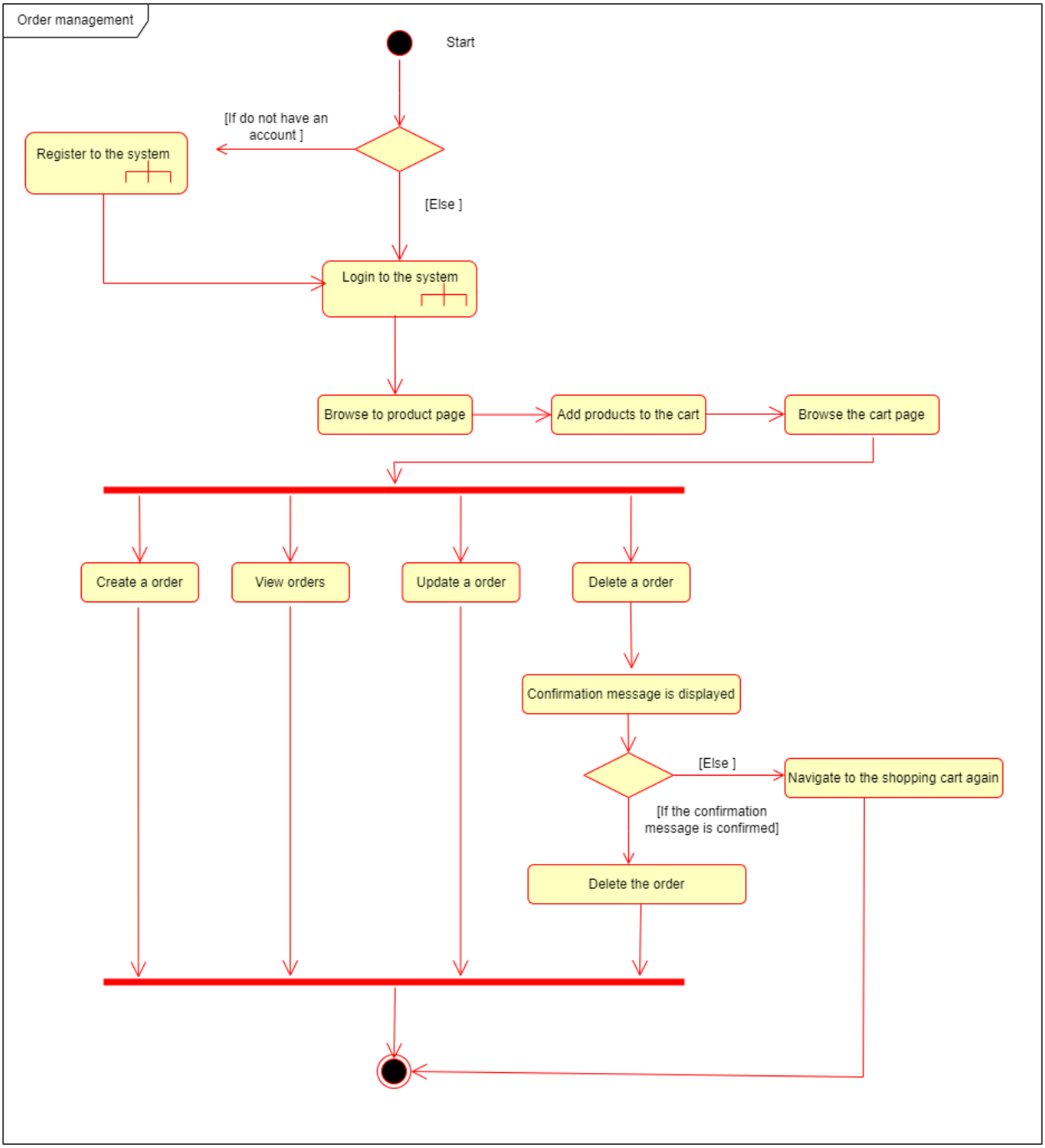
Shopping cart management: Allows users to add, remove, or update products in their shopping cart before making an

order.

Enables users to place orders on the platform and tracks the order status from processing to delivery.

Order history: Provides users with a complete history of their past orders, including order details and receipts.

Order analytics: Enables administrators to track order details.



Figurer 2.6 – Activity Diagram – Order Management

Checkout Service:

Payment processing: Integrates with third-party payment providers to securely process user payments and prevent fraud.

Shipping management: Calculates shipping fees based on the user's location and the quantity of the products.

Order summary and confirmation: Provides users with a summary of their order details and confirms their purchase before finalizing payment.

Review Service:

RearAyur manages product reviews by allowing users to post reviews and keep track of them in one place.

RearAyur administrators can filter product reviews and delete improper or fake reviews.

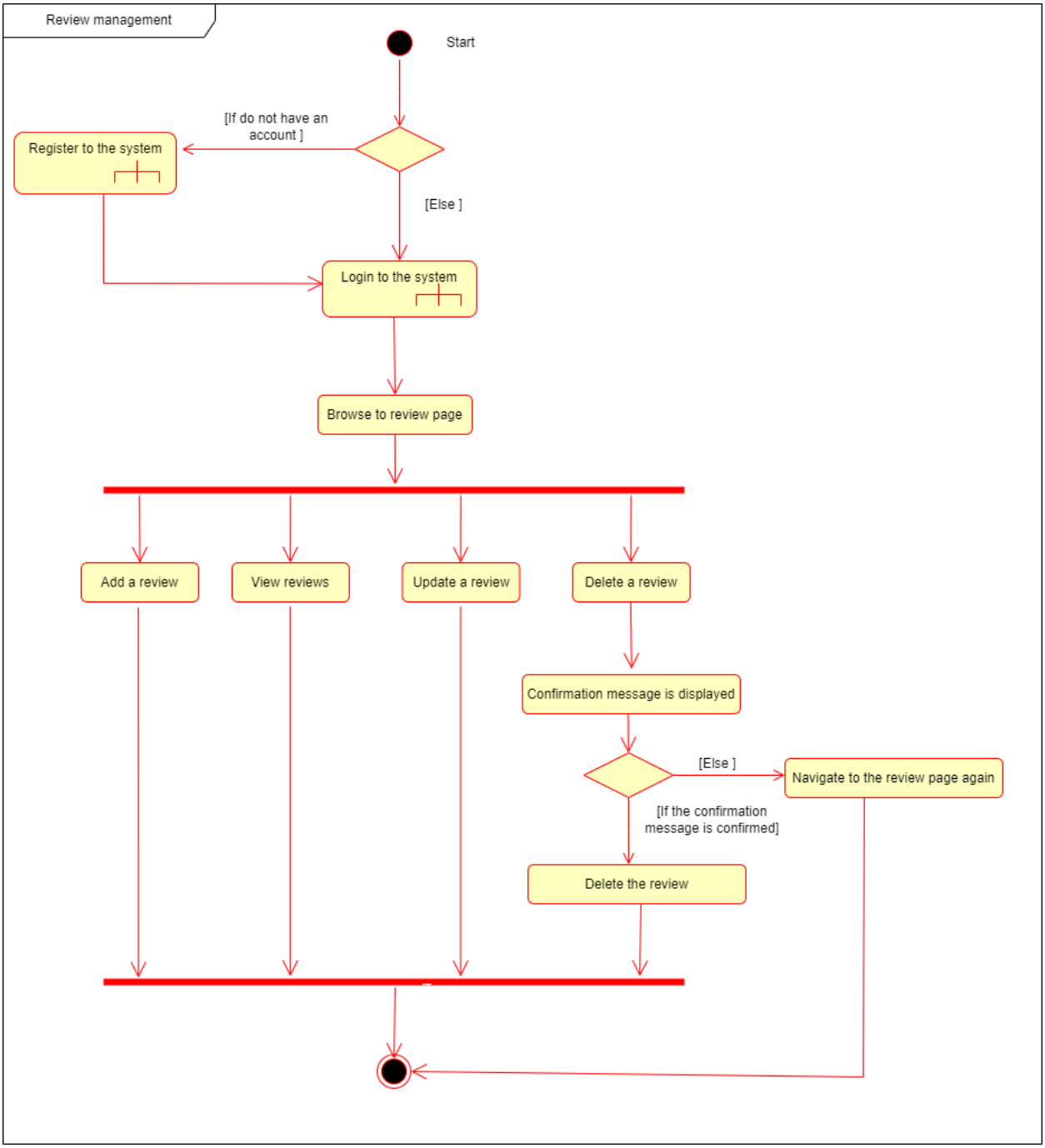


Figure2.7 – Review Management

3) System Feature

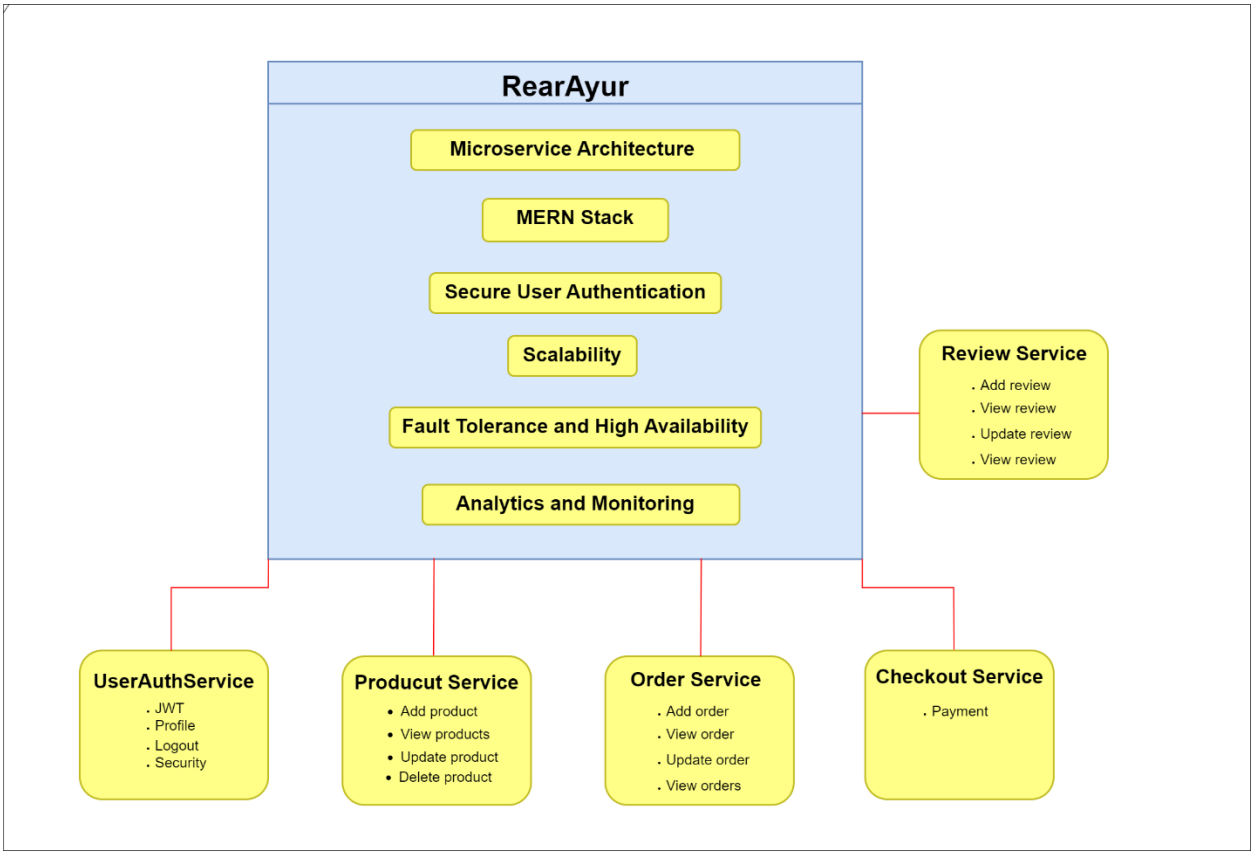


*Figure **Error! Use the Home tab to apply 0 to the text that you want to appear here.**3.1 – System features.*

The key system features of RareAyur are illustrated in this diagram, with each microservice shown as a box with a unique set of supporting features. The services can operate separately and still be able to communicate with one another through APIs thanks to the communication between them, as indicated by the arrows between the boxes.

These are some of RareAyur's system features:

• Microservice Architecture enables independent operation of each service while allowing them to interact with one another via APIs.

• Kubernetes and Docker Deployment: By packing and deploying each service as a container, the application can be scaled and managed more easily.

• MERN Stack: Facilitates rapid and effective application creation, testing, and deployment.

• Secure User Authentication: Using industry-standard protocols like JWT, secure user authentication is provided.

• Scalability: each service can be grown individually based on demand, as it was designed to be scalable.

• Fault Tolerance and High Availability: each service is meant to run in numerous instances at once and is fault-tolerant and highly available.

• Analytics and Monitoring: Contains analytics and monitoring technologies that let developers monitor the performance of their applications, detect problems, and make the most use of their resources.

4) Tools and Technology

The **MERN stack** was used as the foundational technology stack when RareAyur was being developed. With the help of this stack, which consists of MongoDB, Express.js, React.js, and Node.js, developers can develop web apps that are effective, responsive, and scalable. The RareAyur platform has a strong and adaptable back end and front-end architecture because of the implementation of this stack.

Another tool used in the development of RareAyur is **Docker**. It is a platform for containerization that makes it possible to build and distribute applications in discrete, portable settings known as containers. Because each microservice can be developed in a separate environment using Docker, security is improved, and resource efficiency is improved.

Furthermore, **Kubernetes** is being used in the creation of RareAyur. The implementation, scaling, and control of containerized applications are all automated by this open-source platform for container orchestration. The RareAyur platform can be grown up or down as necessary to manage various levels of traffic by utilizing Kubernetes.

Another technology being used in the creation of RareAyur is the **REST API**. It is a web-based API design that enables online connection between various software programs. Due to the standardized method of communication that this technique gives RareAyur, integrating it with applications and other services is made simpler.

Additionally, the creation of RareAyur takes advantage of **API gateways**. They manage routing, load balancing, and authentication while serving as a single point of entry for all incoming requests to the system. In RareAyur, the use of API gateways guarantees that each of the requests is handled consistently, resulting in a better user experience.

Finally, Stripe is being used by the team to process payments. A well-known platform for payment processing called Stripe makes it simple for companies to take payments online. The team is able to provide consumers with a safe and dependable payment processing method by incorporating Stripe into RareAyur.

In conclusion, the use of these kinds of technologies in the building of RareAyur ensures better security, improves resource utilization, makes deployment more scalable, allows for standardization of communication, and makes sure a consistent user experience.

1. Architecture

The microservice architecture, which involves separating the system into smaller, and independent services that can be created, deployed, and scaled separately, is used to build the platform. Greater flexibility, scalability, and reliability are made possible by this approach.

The userAuth-service, order service, product service, checkout service and are the platform's four primary microservices. A brief description of each service is presented below:

1) UserAuth-service

The management of user authentication and authorization falls under the control of this service. It manages user profile management in addition to user login and registration.

2) Product service

Product data management, including management of product descriptions and photos, is the responsibility of this service. Its job is to manage the addition, viewing, updating, and deletion of product data.

3) Order service

Order processing, order addition, order viewing, order updating, and order deletion are all handled by this service.

4) Checkout service

This service oversees monitoring the checkout procedure, which includes order summary, shipment details, and payment processing. Once the money has been successfully processed, it connects with the order service to generate a new order.

5) Review service

Review addition, Review viewing, Review updating, and Review deletion are all handled by this service.

Every microservice is intended to be loosely connected, which allows for communication without being completely integrated. This enables the system to be more flexible and modular.

Microservices often communicate with one another through message queues or RESTful APIs. The result is that the system can adapt to failures or modifications to one service and enables each microservice to communicate with one another in a decoupled manner.

Using containerization and orchestration tools like Docker and Kubernetes, the design of the system can be improved even more. Each microservice can be containerized, making it simpler to set up and oversee them in various environments. inevitably may organize the deployment and scaling of these containers using Kubernetes, which makes it simpler to manage the overall system.

A collaborative purchasing platform for Ayurvedic and Herbal medications and supplements can be built using a microservice architecture since it offers a highly scalable, adaptable, and robust solution to doing so.

2. Service Interfaces

1) UserAuth Service

User authentication and authorization are handled by the UserAuth service. It offers an API through which other services can create new users, authenticate existing users, and grant access to resources. The following is a description of a few of the main service interfaces that the UserAuth service offers.

i. Endpoint for user authentication: The UserAuth service offers an endpoint that different services can use to verify users. By providing a username and password, this endpoint generates an access token that may be used to get access to other system services. Here is an illustration of a possible call to this endpoint.

```
const router = require("express").Router();
const {createUser,authenticateUser} = require("../controller/userController");

router.post("/login",authenticateUser);
router.post("/signup",createUser);

module.exports = router;
```

```
const authenticateUser = async (req, res) => {
    try {
        const { error } = validateAuth(req.body);
        if (error)
            return res.status(400).send({ message: error.details[0].message });

        const user = await User.findOne({ email: req.body.email });
        if (!user)
            return res.status(401).send({ message: "Invalid Email or Password" });

        const validPassword = await bcrypt.compare(
            req.body.password,
            user.password
        );
        if (!validPassword)
            return res.status(401).send({ message: "Invalid Email or Password" });

        const token = user;//.generateAuthToken();
        res.status(200).send({ data: token, message: "Login is successfully" });
    } catch (error) {
        res.status(500).send({ message: "Internal Server Error" });
    }
};
```

ii. Endpoint for user creation: The UserAuth service offers an endpoint that enables the creation of new users by other services. This endpoint receives user data and outputs a freshly constructed user object. Here is the code that could be made to this endpoint.

```
const createUser = async (req, res) => {
    try {
        const { error } = validate(req.body);
        if (error)
            return res.status(400).send({ message: error.details[0].message });

        const user = await User.findOne({ email: req.body.email });
        if (user)
            return res
                .status(409)
                .send({ message: "User with given email already Exist!" });

        const salt = await bcrypt.genSalt(Number(process.env.SALT));
        const hashPassword = await bcrypt.hash(req.body.password, salt);

        await new User({ ...req.body, password: hashPassword }).save();
        res.status(201).send({ message: "User created successfully" });
    } catch (error) {
        res.status(500).send({ message: "Internal Server Error" });
    }
};
```

iii. The UserAuth service also offers middleware for authorizing access to resources, which other services can use. This middleware accepts an access token as input and checks to see if the user identified by that token is authorized to access the resource being requested. Middleware issues an error answer if the user's use is not permitted.

```
let cookies = {};

  const cookiesArray = req.headers.cookie.split(';');

  cookiesArray.forEach((cookie) => {
      const [key, value] = cookie.trim().split('=');
      cookies[key] = value;
  });

  // console.log(cookies.jwt);

if (!cookies?.jwt)
  return res.status(401).json({ message: "Unauthorized 101" });

const refreshToken = cookies.jwt;

jwt.verify(
  refreshToken,
  "gApYVNX8Z9iCm7Jt",
  async (err, decoded) => {
    if (err) return res.status(403).json({ message: "Forbidden" });

    const foundUser = await User.findOne({ email: decoded.username }).exec();

    if (!foundUser)
      return res.status(401).json({ message: "Unauthorized user" });

    const accessToken = jwt.sign(
      {
        UserInfo: {
          username: foundUser.email,
          roles: foundUser.type,
        },
      },
      "eL6Jadh6jBThpztk",
      { expiresIn: "15m" }
    );
      console.log(accessToken);
    res.json({ accessToken });
  }
);
```

```
};

const logout = (req, res) => {
  const cookies = req.cookies;
  if (!cookies?.jwt) return res.sendStatus(204); //No content
  res.clearCookie("jwt", { httpOnly: true, sameSite: "None", secure: true });
  res.json({ message: "Cookie cleared" });
};

const validateAuth = (data) => {
  const schema = Joi.object({
    email: Joi.string().email().required().label("Email"),
    password: Joi.string().required().label("Password"),
  });
  return schema.validate(data);
};

module.exports = {
  createUser,
  authenticateUser,
  refresh,
  logout,
};
```

2) Product Service

RearAyur's product service offers an interface for managing all items. The following endpoints are part of the service interface.

```
//this is to get all data
router.get('/',getProducts)

//get a single data
router.get('/:id',getOneProduct)

//post a new data
router.post('/',addProduct)

//delete data
router.delete('/:id',deleteProduct)

//udate data
router.patch('/:id',updateProduct)
```

i. POST /products: This endpoint retrieves a list of all the items that are currently in stock.

```
//get all products
const getProducts = async (req, res) => {
  const products = await Product.find({}).sort({ createdAT: -1 });
  res.status(200).json(products);
};
```

ii. GET /products/:id: By using its unique ID, this endpoint obtains a specific product.

```
//geta single products
const getOneProduct = async (req, res) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id)) {
    return res.status(404).json({ error: "No such Product" });
  }

  const oneProduct = await Product.findById(id);

  if (!oneProduct) {
    return res.status(400).json({ error: "No such Product" });
  }

  res.status(200).json(oneProduct);
};
```

iii. POST /products: This endpoint adds a brand-new item to the inventory. Details on what is going to be created should be included in the request body.

```
//add new product
const addProduct = async (req, res) => {
  const { image, tital, price, quantity, description } = req.body;

  //add product to db
  try {
    const product = await Product.create({
      image,
      tital,
      price,
      quantity,
      description,
    });
    res.status(200).json(product);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};
```

iv. PATCH/products/:id: A currently available item in the inventory is updated by this endpoint. The most recent product information should be included in the request body.

```
//update a product
const updateProduct = async (req, res) => {
  const { id } = req.params;
  if (!mongoose.Types.ObjectId.isValid(id)) {
    return res.status(404).json({ error: "No such Product" });
  }
  const product = await Product.findByIdAndUpdate(
    { _id: id },
    {
      ...req.body,
    }
  );
  if (!product) {
    return res.status(400).json({ error: "No such Product" });
  }
  res.status(200).json(product);
};
```

v. DELETE /products/:id: By using its unique ID, this endpoint removes a product from the inventory.

```
//delete a product
const deleteProduct = async (req, res) => {
  const { id } = req.params;
  if (!mongoose.Types.ObjectId.isValid(id)) {
    return res.status(404).json({ error: "No such Product" });
  }

  const product = await Product.findOneAndDelete({ _id: id });

  if (!product) {
    return res.status(400).json({ error: "No such Product" });
  }

  res.status(200).json(product);
};
```

3) Order Service

RearAyur's order service offers a management interface for user orders placed on the site. The following endpoints are part of the service interface.

```
//get all the orders
router.get('/',getOrders)

//get a order
router.get('/:id',getOrder)

//add a order
router.post('/',creatOrder)

//delete a order
router.delete('/:id',deleteOrder)

//update a order
router.patch('/:id',updateOrder)
```

i. POST /orders: This endpoint retrieves a list of all the orders that are currently in the database.

```
// get all orders
const getOrders=async(req,res)=>{
    const orders=await Order.find({}).sort({createAt:-1})

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

ii. GET / order /:id: By using its unique ID, this endpoint obtains a specific order.

```
//get a single order
const getOrder=async(req,res)=>{
    const {id}=req.params

    if(!mongoose.Types.ObjectId.isValid(id)){
        return res.status(404).json({error:'no such id'})
    }
    const order=await Order.findById(id)

    if(!order){
        return res.status(404).json({error:'no such oerder'})
    }

    res.status(200).json(order)
}
```

iii. POST / order: This endpoint adds a brand-new order to the database. Details on what is going to be created should be included in the request body.

```
//add a order
const creatOrder=async(req,res)=>{

    //add data to db
    const {userId,products,subtotal,total,shipping,order_status,payment_status}=req.body
    try{
        const order=await Order.create({userId,products,subtotal,total,shipping,order_status,payment_status})
        res.status(200).json(order)
    }catch(error){
        console.log(error);
        res.status(400).json({error:error})
    }

}
```

iv. PATCH/orders/:id: A currently available order in the database is updated by this endpoint. The most recent order information should be included in the request body.

```
//update order
const updateOrder= async(req,res)=>{
    const {id}=req.params

    if(!mongoose.Types.ObjectId.isValid(id)){
        return res.status(404).json({error:'no such id'})
    }
    const order =await Order.findOneAndUpdate({_id:id},{
        ...req.body
    })

    if(!order){
        return res.status(404).json({error:'no such oerder'})
    }

    res.status(200).json(order)
}
```

v. DELETE /order/:id: By using its unique ID, this endpoint removes an order from the inventory.

```
//delete order
const deleteOrder= async(req,res)=>{
    const {id}=req.params

    if(!mongoose.Types.ObjectId.isValid(id)){
        return res.status(404).json({error:'no such id'})
    }

    const order=await Order.findOneAndDelete({_id:id})

    if(!order){
        return res.status(404).json({error:'no such oerder'})
    }

    res.status(200).json(order)

}
```

4) Checkout Service

RearAyur utilizes Stripe as its payment processor since it offers a safe and dependable platform for handling online transactions. This makes it simple for clients to pay for their orders using the payment method of their choice while also guaranteeing the security and privacy of their private financial data. Customers may easily and conveniently complete their purchases because payment service is incorporated into the checkout procedure.

5) Review Service

RearAyur's review service offers an interface for managing all feedback of the customers. The following endpoints are part of the service interface.

```
//get all the review
router.get('/',getReview)

//add a review
router.post('/',creatReview)

module.exports=router
```

i. POST /review: This endpoint retrieves a list of all the reviews that are currently database.

```
//get all review
const getReview=async(req,res)=>{
    const reviews=await Review.find({}).sort({createAt:-1})
    res.status(200).json(reviews)
}
```

ii. POST /review: This endpoint adds a brand-new review to the database. Details on what is going to be created should be included in the request body.

```
// add review
const creatReview=async(req,res)=>{
    //add data to db
    const{userID,description,productID}=req.body
    try{
        const review=await Review.create({userID,description,productID})
        res.status(200).json(review)
    }catch(error){
        res.status(400).json({error:error.messager})
    }
}
```

3. Communications Interfaces

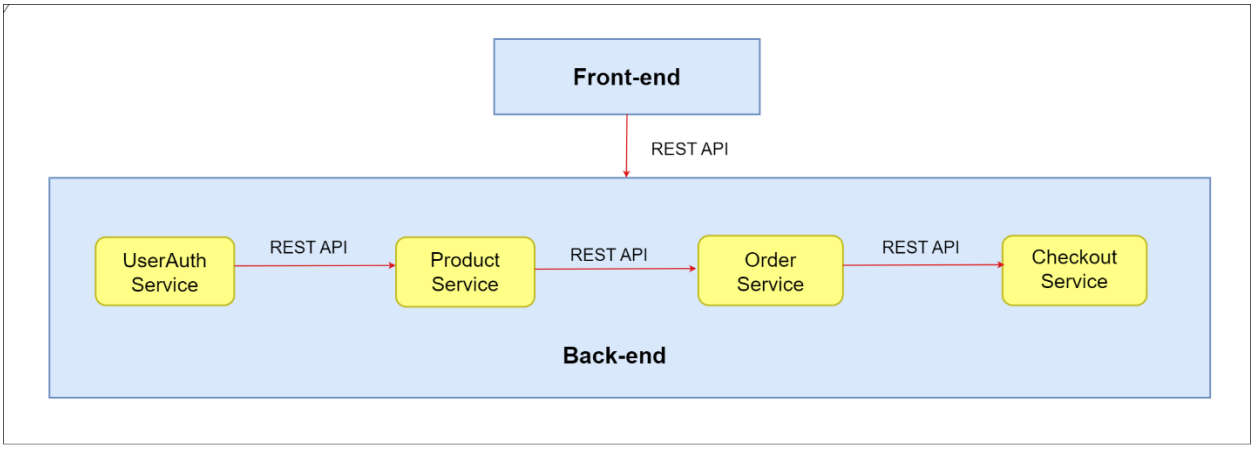


Figure 4.1 – Communication Interfaces

REST APIs are the foundation of RareAyur's communication interfaces amongst its multiple microservices. A common architectural design for creating web services is called REST (Representational State Transfer), and it makes use of HTTP requests to carry out various tasks like data retrieval and system modification.

Each microservice in RareAyur offers a set of REST APIs and lets other services communicate with it. For example, the order service expose APIs so that the checkout service gets details about the order that is Soffered.

With REST APIs, RareAyur can develop a loose relationship between its microservices, enabling them to work separately while yet being able to interact and collaborate efficiently. REST APIs are a trustworthy and convenient choice for developing communication interfaces because they are also extensively used and well-supported.

In conclusion, REST APIs serve as the foundation for RareAyur's communication interfaces, enabling its microservices to connect with one another in a dependable and loosely coupled manner.

5) Other Nonfunctional Requirements

1. Performance

RearAyur's performance can be credited to the adoption of cutting-edge tools, technologies, and microservice architecture.

First off, RearAyur may have a modular design because of the microservice architecture, which enables services to be created and deployed individually. This enhances scalability, dependability, and maintainability. By dividing the load among several services, it also makes it possible for the software to handle huge traffic loads.

Second, RearAyur has access to orchestration and containerization features thanks to the utilization of Docker and Kubernetes. Reduced downtime and higher performance are the results of effective application deployment, scaling, and management.

Thirdly, RearAyur has access to a reliable, scalable, and fast tech stack for creating web apps thanks to the MERN stack (MongoDB, Express.js, React.js, and Node.js). High scalability and efficiency for data storage and retrieval are provided by the document-based NoSQL database structure used by MongoDB. React.js offers a productive front-end interface enabling user interaction, and Express.js and Node.js make it possible to build a small and quick server-side application.

Finally, RearAyur uses API gateways and load balancers to route requests to the proper microservices, ensuring that the application can withstand heavy traffic loads and improve response times. Performance and reliability are increased thanks to the API gateway, which serves as a single point of entry for every client request and adds an extra layer of protection and authentication.

RearAyur is an appropriate solution for a collaborative shopping website for Ayurvedic and Herbal medications and supplements because of its technological stack, tools, and microservice design, which give the application exceptional performance, scalability, stability, and maintainability.

2. Safety & Security

Utilizing the most recent technology, tools, and architectural frameworks, RearAyur has been created with security and safety in mind. The usage of JWT (JSON Web Tokens) for user authentication and authorization is one of RearAyur's key security features.

A JSON object can be securely transmitted between parties using the JWT standard. When users log in and do actions on the platform, RearAyur uses JWT to securely authenticate and authorize them. This ensures that only authorized users can access the system and helps to avoid unauthorized use of sensitive data.

JWT is just one of the security mechanisms used by RearAyur. It also uses encrypted data storage, user input validation, and HTTPS for a secure connection between both the client and the server to avoid

injection attacks. The isolation of various services and reduction of the attack surface area provided by the microservice architecture additionally led to increased security.

RearAyur is an overall safe and secure system that is built to safeguard user data and guard against illegal access. The microservice architecture helps to reduce the risk of security vulnerabilities, and the implementation of JWT and other safety features helps guarantee that the platform is secured against common security threats.

6) Individual Contribution

Description	Student ID	Name	Contribution
Group Leader	IT21028878	Kulasekara D.A.M. N	• All work related to UseAuth service.
Member 2	IT21034404	Nipun P.G. I	• All work related to order service and checkout service. • Final integration
Member 3	IT21039140	Dewasurendra S. V	• All work related to product service.
Member 4	IT21071034	Wanniarachchi T. T	• All work related to Review service.

Appendix

Backend

UserAuth Service

- Models

```
const mongoose = require("mongoose");
const jwt = require("jsonwebtoken");
const Joi = require("joi");
const passwordComplexity = require("joi-password-complexity");

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true },
  password: { type: String, required: true },
//   cfrmpassword: { type: String, required: true },
  type: { type: String, required: true },
});

// userSchema.methods.generateAuthToken = function () {
//   const token = jwt.sign({ _id: this._id }, process.env.JWTPRIVATEKEY);
//   return token;
// };

const User = mongoose.model("user", userSchema);

const validate = (data) => {
  const schema = Joi.object({
    name: Joi.string().required().label("Name"),
    email: Joi.string().email().required().label("Email"),
```

```
    password: passwordComplexity().required().label("Password"),
    // cfrmpswd: passwordComplexity().required().label("Confirm password"),
    type: Joi.string().required().label("Type"),
  });
  return schema.validate(data);
};

module.exports = { User, validate };
```

- Controllers

```
const { User, validate } = require("../models/user");
// const express = require("express");
const jwt = require("jsonwebtoken");
const Joi = require("joi");
require("dotenv").config();
// const cookieParser = require('cookie-parser')
// const app = express();
// app.use(cookieParser());

const createUser = async (req, res) => {
  try {
    const { error } = validate(req.body);
    if (error)
      return res.status(400).send({ message: error.details[0].message });

    const user = await User.findOne({ email: req.body.email });
    if (user)
      return res
        .status(409)
        .send({ message: "User with given email already Exist!" });

    // const salt = await bcrypt.genSalt(Number(process.env.SALT));
    const hashPassword = req.body.password;

    await new User({ ...req.body, password: hashPassword }).save();
    res.status(201).send({ message: "User created successfully" });
  } catch (error) {
    res.status(500).send({ message: "Internal Server Error" });
```

```
  }
};

const authenticateUser = async (req, res) => {
  try {
    const { error } = validateAuth(req.body);
    if (error)
      return res.status(400).send({ message: error.details[0].message });

    const user = await User.findOne({ email: req.body.email });
    if (!user)
      return res.status(401).send({ message: "Invalid Email or Password" });

    if ( req.body.password != user.password)
      return res.status(401).send({ message: "Invalid Email or Password" });

    const accessToken = jwt.sign(
      {
        UserInfo: {
          username: user.email,
          roles: user.type,
        },
      },
      "eL6Jadh6jBThpztk",
      { expiresIn: "15m" }
    );

    const refreshToken = jwt.sign({ username: user.email }, "gApYVNX8Z9iCm7Jt", {
      expiresIn: "7d",
    });

     res.cookie('jwt', refreshToken, {
       httpOnly: true, //accessible only by web server
       // secure: true, //https
       // sameSite: 'None', //cross-site cookie
      // maxAge: 7 * 24 * 60 * 60 * 1000 //cookie expiry: set to match rT
    });
    // Send accessToken containing username and roles
    // res.json({ accessToken });

    res.status(200).send({
      data: user,
      accesstoken: accessToken,
      message: "Login is successfully",
```

```
    });
  } catch (error) {
    res.status(500).send({ message: "Internal Server Error" });
    console.log(error);
  }
};

const refresh = (req, res) => {
  // const cookies = req.headers.cookie.trim().split('=');
  // // console.log(req.rawHeaders.Cookie);
  // console.log(req.headers.cookie);
  // // console.log(req);

  let cookies = {};

    const cookiesArray = req.headers.cookie.split(';');

    cookiesArray.forEach((cookie) => {
        const [key, value] = cookie.trim().split('=');
        cookies[key] = value;
    });

    // console.log(cookies.jwt);

  if (!cookies?.jwt)
    return res.status(401).json({ message: "Unauthorized 101" });

  const refreshToken = cookies.jwt;

  jwt.verify(
    refreshToken,
    "gApYVNX8Z9iCm7Jt",
    async (err, decoded) => {
      if (err) return res.status(403).json({ message: "Forbidden" });

      const foundUser = await User.findOne({ email: decoded.username }).exec();

      if (!foundUser)
        return res.status(401).json({ message: "Unauthorized user" });

      const accessToken = jwt.sign(
        {
          UserInfo: {
            username: foundUser.email,
```

```
            roles: foundUser.type,
          },
        },
        "eL6Jadh6jBThpztk",
        { expiresIn: "15m" }
      );
        console.log(accessToken);
      res.json({ accessToken });
    }
  );
};

const logout = (req, res) => {
  const cookies = req.cookies;
  if (!cookies?.jwt) return res.sendStatus(204); //No content
  res.clearCookie("jwt", { httpOnly: true, sameSite: "None", secure: true });
  res.json({ message: "Cookie cleared" });
};

const validateAuth = (data) => {
  const schema = Joi.object({
    email: Joi.string().email().required().label("Email"),
    password: Joi.string().required().label("Password"),
  });
  return schema.validate(data);
};

module.exports = {
  createUser,
  authenticateUser,
  refresh,
  logout,
};
```

- Routes

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

const {createUser,authenticateUser,refresh,logout} = require("../controller/userController");


router.post("/login",authenticateUser);
```

```
router.post("/signup",createUser);
router.get("/refresh",refresh);

module.exports = router;
```

Product Service

- Controllers

```
const Product = require("../models/productModels");
const mongoose = require("mongoose");

//get all products
const getProducts = async (req, res) => {
  const products = await Product.find({}).sort({ createdAT:-1 });
  res.status(200).json(products);
};
//geta single products
const getOneProduct = async (req, res) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id)) {
    return res.status(404).json({ error: "No such Product" });
  }

  const oneProduct = await Product.findById(id);

  if (!oneProduct) {
    return res.status(400).json({ error: "No such Product" });
  }
```

```
  res.status(200).json(oneProduct);
};

//add new product
const addProduct = async (req, res) => {
  const { image, tital, price, quantity, description } = req.body;

  //add product to db
  try {
    const product = await Product.create({
      image,
      tital,
      price,
      quantity,
      description,
    });
    res.status(200).json(product);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

//delete a product
const deleteProduct = async (req, res) => {
  const { id } = req.params;
  if (!mongoose.Types.ObjectId.isValid(id)) {
    return res.status(404).json({ error: "No such Product" });
  }
```

```
  const product = await Product.findOneAndDelete({ _id: id });

  if (!product) {
    return res.status(400).json({ error: "No such Product" });
  }

  res.status(200).json(product);
};

//update a product
const updateProduct = async (req, res) => {
  const { id } = req.params;
  if (!mongoose.Types.ObjectId.isValid(id)) {
    return res.status(404).json({ error: "No such Product" });
  }
  const product = await Product.findByIdAndUpdate(
    { _id: id },
    {
      ...req.body,
    }
  );
  if (!product) {
    return res.status(400).json({ error: "No such Product" });
  }
  res.status(200).json(product);
};

module.exports = {
  getProducts,
```

```
  getOneProduct,
  deleteProduct,
  updateProduct,
  addProduct,
};
```

- Models

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

const schema=mongoose.Schema

const productSchema=new schema({
   image:{
      type:String,
      require:true
   },
   tital:{
      type:String,
      require:true
   },
   price:{
      type:Number,
      require:true
   },
   quantity:{
      type:Number,
      require:true
   },
```

```
    description:{
        type:String,
        require:true
    }
},{timestamps:true})
module.exports=mongoose.model('product',productSchema)
```

- Routes

```
const express=require('express')
const multer = require('multer');
// const{v4:uuidv4}=require('uuid');
const path=require("path")

const {
    getProducts,
    getOneProduct,
    deleteProduct,
    updateProduct,
    addProduct
} = require ('../controllers/productController')

const router =express.Router()

//this is to get all data
router.get('/',getProducts)

//get a single data
router.get('/:id',getOneProduct)
```

```
//post a new data

router.post('/',addProduct)

//delete data
router.delete('/:id',deleteProduct)



//udate data
router.patch('/:id',updateProduct)



module.exports = router
```

Order Service

- Controllers

```
const Order=require('../models/OrdersModel')
const mongoose=require('mongoose')

// get all orders
const getOrders=async(req,res)=>{
   const orders=await Order.find({}).sort({createAt:-1})


   res.status(200).json(orders)
}
//get a single order
```

```
const getOrder=async(req,res)=>{
    const {id}=req.params

    if(!mongoose.Types.ObjectId.isValid(id)){
        return res.status(404).json({error:'no such id'})
    }
    const order=await Order.findById(id)

    if(!order){
        return res.status(404).json({error:'no such oerder'})
    }

    res.status(200).json(order)
}

//add a order
const creatOrder=async(req,res)=>{

    //add data to db
    const {userId,products,subtotal,total,shipping,order_status,payment_status}=req.body
    try{
        const order=await
Order.create({userId,products,subtotal,total,shipping,order_status,payment_status})
        res.status(200).json(order)
    }catch(error){
        console.log(error);
        res.status(400).json({error:error})
    }
```

```
}
//delete order
const deleteOrder= async(req,res)=>{
    const {id}=req.params

    if(!mongoose.Types.ObjectId.isValid(id)){
        return res.status(404).json({error:'no such id'})
    }

    const order=await Order.findOneAndDelete({_id:id})

    if(!order){
        return res.status(404).json({error:'no such oerder'})
    }

    res.status(200).json(order)

}

//update order
const updateOrder= async(req,res)=>{
    const {id}=req.params

    if(!mongoose.Types.ObjectId.isValid(id)){
        return res.status(404).json({error:'no such id'})
    }
    const order =await Order.findOneAndUpdate({_id:id},{
        ...req.body
```

```
    })

    if(!order){
        return res.status(404).json({error:'no such oerder'})
    }

    res.status(200).json(order)
}



module.exports={
    creatOrder,
    getOrder,
    getOrders,
    deleteOrder,
    updateOrder
}
```

- Models

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

const orderSchema = new mongoose.Schema(
  {
    userId: { type: String, required: true },

    products: [
      { productId: { type: String }, quantity: { type: Number, default: 1 } ,unit_amount:{ type: Number, required: true }},
    ],
```

```
    subtotal: { type: Number, required: true },

    total: { type: Number, required: true },

    // shipping: { type: Object, required: true },

    shipping: {
      city: {type: String, required: true},
      country: {type: String, required: true},
      line1: {type: String, required: true},
      line2: {type: String, required: true},
      postal_code: {type: String, required: true},
      state: {type: String, required: true}
    },

    order_status: { type: String, default: "pending" },

    payment_status: { type: String, required: true },
  },
  { timestamps: true }
);

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

module.exports = Order;


// "shipping": {
```

```
//   "address": {
//     "city": "kkemwk",
//     "country": "US",
//     "line1": "dwkdkw",
//     "line2": "dkwndnwk",
//     "postal_code": "02000",
//     "state": "MA"
//   },
//   "name": "dwndw"
// },


 // shipping: { type: Object, required: true },

 /*

 shipping: {
  city: {type: String, required: true},
  country: {type: String, required: true},
  line1: {type: String, required: true},
  line2: {type: String, required: true},
  postal_code: {type: String, required: true},
  state: {type: String, required: true}
},
```

```
 */
```

- Routes

```
const express=require('express')
const router=express.Router()
const {
    creatOrder,
    getOrder,
    getOrders,
    deleteOrder,
    updateOrder
}=require('../controllers/orderController')


//get all the orders
router.get('/',getOrders)

//get a order
router.get('/:id',getOrder)

//add a order
router.post('/',creatOrder)

//delete a order
router.delete('/:id',deleteOrder)

//update a order
```

```
router.patch('/:id',updateOrder)


module.exports=router
```

Checkout Service

- Controllers

```
const Stripe = require("stripe");
require("dotenv").config();
const axios = require("axios");
let endpointSecret;
// endpointSecret =
//   "whsec_8d3f13404fa343ad1780924c598d84b847dff91a119169b0f879bc5ce1860c82";


const stripe = Stripe(

"sk_test_51MzASQCqzSR1UB2eMsDHitJ4OeQizkxaYUySMVae15qauDPxmakT24IfGl0iMa5Jr3CvpDyT84CR
XRBnyzvQv2pZ00PtWqEuiE"
);




//---------------------------------------------------------------------------------------------------
//handle the order creation after completed payment


const handleOrder = (data, customer, lineItems) => {
  const allProducts = lineItems.data;
  const productData = allProducts.map((item) => {
    return {
      productId: item.description,
```

```
      quantity: item.quantity,
      unit_amount: item.price.unit_amount/100.0,
    };
  });

  let userId, subtotal, total, payment_status, products, shipping;

  userId = customer.metadata.userId;
  products = productData;
  subtotal = data.amount_subtotal/100.0;
  total = data.amount_total/100.0;
  shipping = data.shipping_details.address;
  payment_status = data.payment_status;

  axios
    .post("http://localhost:5003/api/orders/", {
      userId,
      products,
      subtotal,
      total,
      shipping,
      payment_status,
    })
    .then((res) => {
      console.log(res.data);
    })
    .catch((error) => console.log(error));
};
```

```
//-----------------------------------------------------------------------------
//create a checkout using stripe payment gateway

const makeACheckout = async (req, res) => {
  const customer = await stripe.customers.create({
    metadata: {
      userId: "1234",
    },
  });

  const line_items = req.body.cart.map((item) => {
    return {
      price_data: {
        currency: "usd",
        product_data: {
          name: item.id,
          images: [item.image],
          description: item.title,
          metadata: {
            id: item.id,
          },
        },
        unit_amount: item.price * 100,
      },
      quantity: item.quantity,
```

```
  };
});
const session = await stripe.checkout.sessions.create({
  shipping_address_collection: { allowed_countries: ["US", "CA"] },
  shipping_options: [
    {
      shipping_rate_data: {
        type: "fixed_amount",
        fixed_amount: { amount: 0, currency: "usd" },
        display_name: "Free shipping",
        delivery_estimate: {
          minimum: { unit: "business_day", value: 5 },
          maximum: { unit: "business_day", value: 7 },
        },
      },
    },
    {
      shipping_rate_data: {
        type: "fixed_amount",
        fixed_amount: { amount: 1500, currency: "usd" },
        display_name: "Next day air",
        delivery_estimate: {
          minimum: { unit: "business_day", value: 1 },
          maximum: { unit: "business_day", value: 1 },
        },
      },
    },
  ],
  line_items,
```

```
    mode: "payment",
    customer: customer.id,
    success_url: `http://localhost:3000/success`,
    cancel_url: `http://localhost:3000/cart`,
  });

  res.send({ url: session.url });
};




//-------------------------------------------------------------------------------------------
//Stripe webhook- used to fetch data from payment in stripe

const stripeWebhook = (req, res) => {
  const sig = req.headers["stripe-signature"];

  let data;
  let eventType;

  if (endpointSecret) {
    let event;

    try {
      event = stripe.webhooks.constructEvent(req.body, sig, endpointSecret);
      console.log("webhook verified");
```

```
  } catch (err) {
    console.log(`Webhook Error: ${err.message}`);
    res.status(400).send(`Webhook Error: ${err.message}`);
    return;
  }
  data = event.data.object;
  eventType = event.type;
} else {
  data = req.body.data.object;
  eventType = req.body.type;
}

// Handle the event

if (eventType === "checkout.session.completed") {
  stripe.customers
    .retrieve(data.customer)
    .then(async (customer) => {
      try {
        stripe.checkout.sessions.listLineItems(
          data.id,
          {},
          function (err, lineItems) {
            console.log(data);
            handleOrder(data, customer, lineItems);
          }
        );
      } catch (err) {
        console.log(err);
```

```
      }
    })
    .catch((err) => console.log(err.message));
 }

 // Return a 200 response to acknowledge receipt of the event
 res.send().end();
};

module.exports = {
 makeACheckout,
 stripeWebhook,
};
```

- Routes

```
// This is your test secret API key.
const express = require('express');
const {makeACheckout,stripeWebhook} = require('../controller/checkoutController');

const router = express.Router();

router.post('/create-checkout-session',makeACheckout);
router.post('/webhook', express.raw({type: 'application/json'}),stripeWebhook);

module.exports = router;
```

Review Service

- Controllers

```
const Review=require('../models/reviewModel')


//get all review
const getReview=async(req,res)=>{
   const reviews=await Review.find({}).sort({createAt:-1})
   res.status(200).json(reviews)
}


// add review
const creatReview=async(req,res)=>{
   //add data to db
   const{userID,description,productID}=req.body
   try{
      const review=await Review.create({userID,description,productID})
      res.status(200).json(review)
   }catch(error){
      res.status(400).json({error:error.messager})
   }
}
module.exports={
   getReview,
   creatReview
}
```

- Models

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

const reviewSchema = new mongoose.Schema(
  {
    userID:{
      type: String,
      required:true
    },
    description:{
      type: String,
      required:true
    },
    productID:{
      type: String,
      required:true
    }
  },
  { timestamps: true }
);
module.exports = mongoose.model("Review", reviewSchema);
```

- Routes

```
const express=require('express')
const router=express.Router()
const {
  getReview,
```

```
    creatReview
}=require('../controller/reviewController')

//get all the review
router.get('/',getReview)

//add a review
router.post('/',creatReview)

module.exports=router
```