UniCat -Online Learning Management System

Project Report

Sri Lanka Institute of Information Technology

SE3020 – Distributed Systems

Y3.S1.WE.SE.02.02

Group ID: S2-SE-WE-61

IT21214820 - Dharmathilake K.A. D. K.D

IT21196188 – Bhanuka H.L.

IT21146374 –Nuwanthika P.G.P. J.

IT21214516 – Fernando N. K. B.

Declaration

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

Authors:

Author SID	Author name
IT21214820	Dharmathilake K. A. D. K. D.
IT21196188	Bhanuka H. L.
IT21146374	Nuwanthika P. G. P. J.
IT21214516	Fernando N. K. B

Date: 13/05/2024

Abstract

This report outlines the development of UniCat, a web-based microservices application stack for the SE3020 - Distributed Systems module. Employing modern technologies including Docker, Kubernetes, and a MERN (MongoDB, Express, React, Node.js) stack, UniCat aims to transform online learning by offering a user-friendly interface for learners and instructors alike. The backend of the application is divided into microservices. Course Management Service (CMS) Learner Service, Payment Integration Service and Notification Service. Each service serving a function. This architecture improves scalability and maintenance while ensuring service delivery. Security measures have been put in place to protect data and user privacy through authentication, authorization mechanisms and password encryption. Data integrity is maintained by storing information in a database accessible only to authorized users. UniCat's online accessibility allows to use from anywhere at any time creating an environment, for continuous learning. The development process focused on efficiency and reliability to create a web-based application that aims to revolutionize education.

Acknowledgment

The work detailed in this document was carried out as part of our project for the Distributed Systems module in 2024. As a team, we express our sincere gratitude to all who provided guidance and support to help us successfully finish our project. Special thanks go to all the lecturers and instructors attached to the Distributed Systems (DS) module. Their advice and encouragement were really helpful in completing this project successfully.

Finally, we sincerely appreciate all team members who devoted their utmost effort and commitment to the successful completion of the project. This educational platform serves as a testament to our collective hard work and dedication throughout the semester.

Table of Contents

Table Of Figures

1. Introduction

1.1. Background

Our project focuses on creating an educational platform designed to meet the growing need for online learning. With the rise of digital technology and the expanding reach of the internet, there is an increasing demand for a versatile platform that provides a wide variety of courses to learners globally. Drawing inspiration from successful platforms such as Coursera and Udemy, our goal is to develop an intuitive and user-friendly interface that enables smooth browsing, enrollment, and access to courses spanning a wide range of subjects and disciplines. The platform will function as a centralized hub where students can discover a wide range of courses covering various subjects, from computer science and engineering to humanities, business, and more. Learners can sign up for as many classes as they wish simultaneously without concerns about schedule conflicts, thereby maximizing their learning potential. Instructors will have access to a dedicated Course Management Service, allowing them to easily add, update, and remove course content based on their needs. Administrators will also be responsible for approving course content, integrating payment gateways, and managing financial transactions for course enrollments. This ensures a streamlined and efficient process for both course creators and platform administrators.

Our main objective is to establish an educational environment that promotes ongoing learning and individual development. With a wide range of courses, user-friendly interfaces, and strong security, we hope to enable students from all backgrounds to reach their academic goals and realize their full potential.

1.1. Problems & Motivation

The field of education has experienced a notable transformation in recent years, as online learning platforms have revolutionized how people gain knowledge and skills. Despite the many platforms available, there are still several challenges that hinder the effective sharing and learning of educational content. One major problem is the limited access to high-quality educational materials, especially for people living in remote or underserved areas. Many students might find traditional educational institutions to be financially or geographically unaffordable, limiting their opportunities for personal and professional growth.

The motivation for creating an educational platform that provides a variety of courses is to tackle these urgent challenges and empower learners with accessible, engaging, and effective educational experiences. By utilizing cutting-edge technology and educational best practices, our platform strives to democratize education access, promote lifelong learning, and empower individuals with the knowledge and skills necessary to succeed in a constantly evolving world.

1.2. Structure of the report

The following section of the report will outline the tools used to complete this project, along with the implementation, design development, testing, and evaluation procedures.

1.3. GitHub Link

https://github.com/Kavithma-Dharmathilake/DS-Assignment

1.4. Objectives

• Provide a smooth and user-friendly interface that allows learners to easily navigate, explore, and enroll in courses across various subjects and disciplines.

• Empower Instructors with a user-friendly Course Management Service, enabling them to easily add, update, and remove course content, track learner progress, and engage with enrolled students..

• Facilitate multiple course enrollments for learners, ensuring a flexible and personalized learning experience without scheduling conflicts.

• Integrate with trusted third-party payment gateways to ensure secure and seamless course enrollment transactions.

• Offer easy-to-use payment gateways that securely handle transactions and safeguard client data, along with secure payment methods.

• To prevent fraud and maintain platform integrity, administrators should manually verify and approve course enrollments.

• Give students a to use platform to check their enrollment status reach course materials and interact with teachers and classmates via discussion forums and messaging tools.

- To support educated decision-making and ongoing platform enhancement, allow students to evaluate and review courses, teachers, and course materials.

2. Solution Overview

2.1. User service

The user service manages user authentication and registration within the Coursecraft platform. User registration and login are key functions of the user service. Customers must use their email address as a username and provide the necessary password, address, role, and contact number to create an account. These fields are mandatory except for the profile field. Therefore, validations are needed for email, password, and contact number. Users cannot register with an existing email address, must adhere to the correct email pattern, use a password of at least 6 characters, ensure the contact number is 10 digits long, and avoid using letters and special characters.

Using Docker, the user service can be containerized into separate functionalities such as user authentication, authorization, and profile management, each packaged as a Docker image and deployed as a container. This allows for flexible scaling, where containers for specific functionalities can be scaled based on user demand. Kubernetes can be used to manage and scale the containers, ensuring high availability and security for the user service.

2.2. Course Service

The Course Service manages the catalog of courses available on the educational platform. It includes features like course listing, detailed course info, and personalized recommendations. Instructors can create, update, and manage course content with comprehensive details on topics, syllabi, and materials. Learners can search, browse, and enroll in courses based on interests and objectives.

Docker containerization modularizes Course Service functions, such as course catalog management, search, and inventory. Each function is in a Docker image and deployed as a container for scalable demand. Kubernetes manages containers for fault tolerance and high availability.

2.3. Lerner Service

Learners can easily find courses from their favorite instructors, building trust and loyalty in the platform. Instructors can establish their own online store, personalize the design to showcase their teaching style and knowledge, and oversee elements like course listings, orders, and learner engagement.

With Docker, the store service can be containerized into separate components such as web server, caching, and content delivery, each packaged as a Docker image and deployed as a container. This allows for horizontal scaling, where additional containers can be added or removed dynamically based on web traffic. Kubernetes can be used to manage and scale the containers, providing load balancing and automatic scaling capabilities for the store service.

2.4. Payment service

The course payment service is designed to offer customers a smooth and convenient payment experience. As courses grow in popularity, having a reliable and secure payment service is crucial. It provides multiple payment options including credit cards and debit cards, with Stripe as a payment choice. Customers can choose the payment method that suits their needs best. Every transaction is handled with care and security. Enhancing the payment experience, features like order tracking and secure payment information storage are also available.

With Docker, the payment service can be containerized into separate components such as payment gateways, transaction processors, and databases, each packaged as a Docker image and deployed as a container. This allows for horizontal scaling, where additional containers can be added or removed dynamically based on the payment processing load. Kubernetes can be used to manage and scale the containers across a cluster of Docker nodes, ensuring high availability and fault tolerance for the payment service.

3. Individual Contributions

3.1. Bhanuka H. L. - IT21196188

User-Service Backend

In the backend, I developed the user-service module to manage users, categorized as students, instructors, and admins. This module facilitates user authentication, allowing all three user types to sign up and log in. Additionally, it offers user management functionalities, empowering users to modify their personal information.

The purpose of this service lies in its robust authentication and authorization system. System routes are secured, requiring a valid token for access. Upon successful token validation, users gain access to the system's services; otherwise, an authorization error is displayed. This crucial functionality was implemented through the integration of the JSON Web Token package within the service. Further I used bcrypt encryption for password hashing, supporting the service's security measures. By using bcrypt, sensitive user credentials are safeguarded against unauthorized access, thereby enhancing overall system data protection.

Payment Service

I have implemented the seamless integration of Stripe payment gateway into our front-end React application, a important enhancement that increase payment functionality. Now, users can effortlessly and securely complete transactions for their orders using credit cards, ensuring a streamlined and convenient payment process. This integration not only optimizes user experience but also reinforces the application's reliability and trustworthiness in handling sensitive financial transactions.

Frontend Contributions

In terms of my front-end contributions, I took charge of implementing user authentication and authorization workflows using React components. This included crafting user-friendly login and signup forms with validation and error handling features. To ensure security, I stored the JSON Web Tokens received from the backend securely in the browser's local storage for consistent user sessions. Moreover, I designed and developed the user profile page, enabling users to view and update their personal information seamlessly. To enhance security measures, I utilized Bcrypt for password encryption and validator for data validation, ensuring data integrity throughout the process.

In addition to user management, I handled the payment functionalities for course purchases, integrating Stripe for secure transactions. Users can easily add courses to their shopping cart and complete single or multiple course purchases effortlessly. Furthermore, I implemented a payment history feature allowing users to track and search their transactions conveniently by payment ID. For administrative purposes, I ensured that admins have access to view all payments made by users across the system. This comprehensive overview allows admins to manage payments effectively and maintain financial records efficiently.

3.2. Dharmathilake K. A. D. K. D. - IT21214820

Course-Service Backend – Instructor User Role

During the development of the Course Management Service, my main role was to create and implement the backend features that allow course instructors to efficiently manage course content and enrollment details. I set up endpoints within the Course Management Service to manage requests from course instructors for adding, updating, or deleting course content. This involved designing data models and database schemas to securely store course details and videos. Through the implementation of proper validation and error handling mechanisms, I maintained the integrity and consistency of course data.

Furthermore, I developed features for instructors to track learner progress within their courses. This included creating endpoints to retrieve learner progress data, like completed watched videos, and overall course completion status.

Frontend Contributions

In the realm of frontend development, I crafted and implemented user-friendly interfaces for instructors to efficiently handle course content and enrollment details. This involved designing responsive web interfaces for easy addition, modification, or removal of course materials, like videos.

I employed React components and state management libraries to construct interactive interfaces that empower instructors to upload new content, edit materials, and track learner progress seamlessly. By integrating feedback mechanisms and error handling features, I guaranteed a seamless and intuitive user experience for instructors.

3.3. Fernando N. K. B - IT21214516

Learner-Service Backend

In the backend, I developed the learner service. This service is used by users to enroll in and follow up on the courses that they purchased. Users can enroll into a course directly from the home page after making the payment successfully or user can select multiple courses and add to cart and enroll multiple courses simultaneously. When enrolling in a course or unenrolling from a course, users will be notified by email and SMS. Email service and SMS service are integrated by using third-party services, which are EmailJS and NotifyMe. I also added the e functionality for the user to track their progress on each enrolled course.

Frontend Contributions

In terms of my contributions to front-end development, I was responsible for implementing a cart for this website so that users could add courses to their cart and enroll in multiple courses.

3.4. Nuwanthika P. G. P. J. - IT21146374

Course Service Backend – Admin User Role

For the backend development of the Admin Role, I was responsible for implementing key functionalities needed for administrative tasks in the educational platform. This involved creating and enhancing backend services for admin users to oversee learners, enrollments, payments, and course content approval. The Admin Role backend service enables administrators to see all learners registered on the platform and take actions like enrolling or unenrolling learners from courses as required. This entailed setting up API endpoints to retrieve learner data and implementing procedures to enroll or unenroll learners from courses, guaranteeing data integrity and security. Additionally, I implemented functionality for administrators to handle maintaining transaction records securely within the system. In addition, I created a feature for administrators to review and approve course content that instructors upload. This included establishing a process for course content approval, enabling administrators to assess and authorize course materials before they are accessible to learners. This guarantees the quality and significance of course content on the platform, improving the overall learning experience for users.

Frontend Contributions

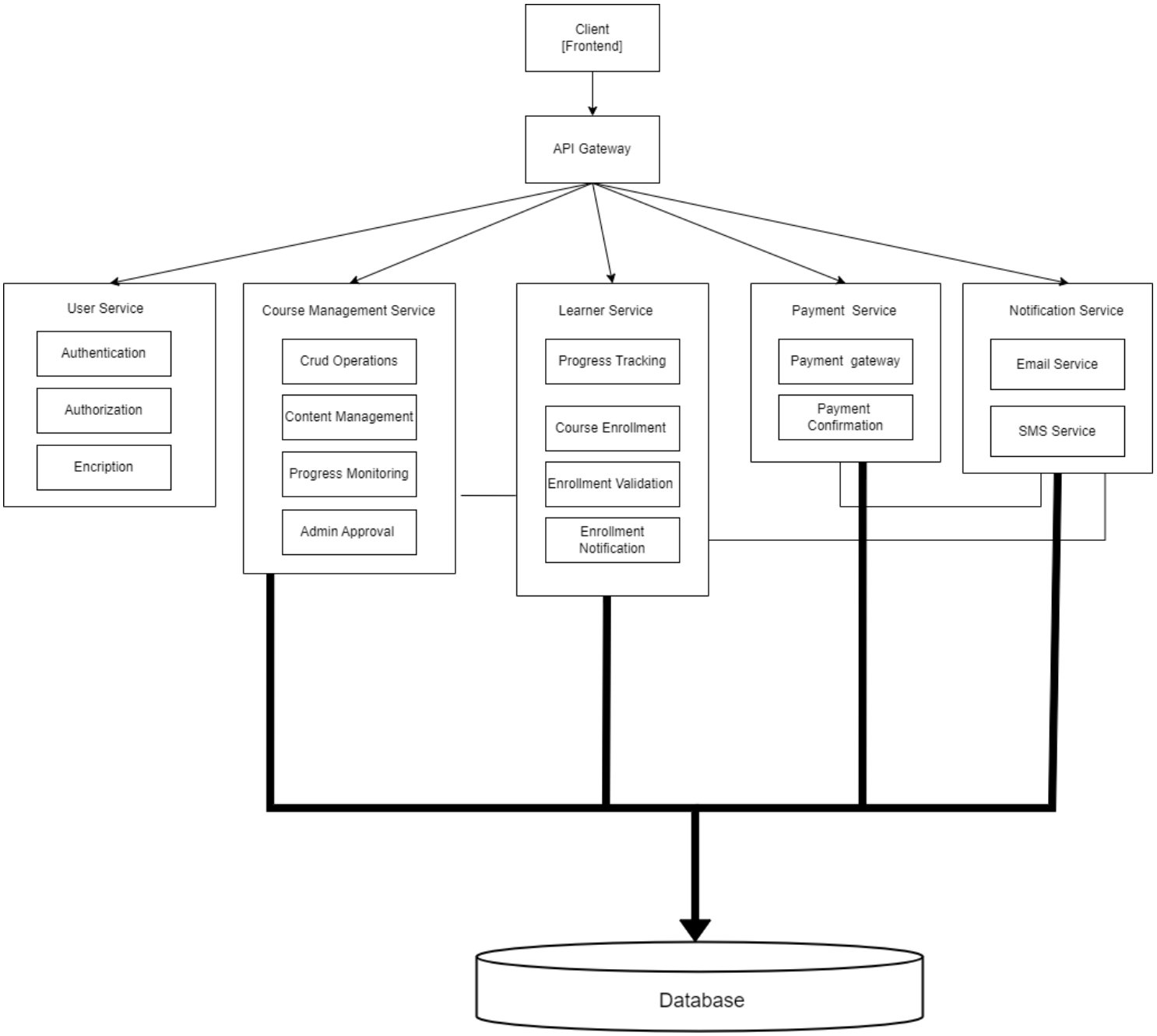
I created user-friendly interfaces for administrators to view learner data, enroll or unenroll learners from courses, and manage payments efficiently. Additionally, I developed interfaces for administrators to approve course content uploaded by instructors, providing a streamlined workflow for content approval.

Docker

I created Docker files for the Admin Role service and other backend services, optimizing the build process for Docker images. These images were pushed to Docker Hub and integrated into the Kubernetes deployment file. Collaborating with the team, I defined deployment configurations for each service as separate containers. Additionally, I set up the Kubernetes service file to enable load balancing and efficient traffic management for the deployed services, ensuring scalability and high availability of the educational platform. My work on Kubernetes and Docker deployment streamlined the process of deploying and managing backend services in a containerized environment.

4. Design

4.1. High Level Architectural Diagram



4.1.Overview of architecture

4.2. Interface Explanations

The educational platform adopts a microservices architectural style, breaking down the backend into distinct services to cater to various functions. Each service exposes interfaces that are meticulously crafted to streamline interactions between users and the core features of the platform.

- **User-service** – The User Service oversees user authentication and authorization for different user categories: learners, course instructors, and administrators. It offers interfaces for user login and registration, utilizing JSON Web Tokens (JWT) for authentication. Upon successful login, JWT tokens are generated and utilized to authorize subsequent requests. Passwords are securely encrypted and decrypted using BcryptJS with appropriate salt values. This service also collaborates with an email service to dispatch confirmation emails upon user registration. Furthermore, it safeguards routes of other services by validating tokens on every request, ensuring secure access to platform resources.

- **Course-Service** –The Course Service manages course-related functionalities, empowering course instructors to add, modify, and remove course content like lecture notes, videos. It provides interfaces for instructors to handle course materials and enrollment particulars. Learners can access course content and monitor their progress through the interfaces offered by this service.

- **Payment-Service** – The Payment Service oversees all payment transactions within the platform, furnishing interfaces for learners to enroll in courses and make payments securely. It supports multiple payment methods. Features such as order tracking and secure storage of payment information enhance the overall payment experience for users. Additionally, real-time updates on delivery status are furnished to ensure transparency and dependability.

4.3. Justification of the usage of Docker and Kubernetes

The decision to incorporate Docker and Kubernetes into our project was driven by the need for scalability, reliability, and performance optimization. By leveraging these technologies, we aimed to streamline our development and deployment processes while ensuring consistency and efficiency across the entire system architecture.

Docker Integration:

Each backend service, such as the User Service, Course Service and Payment Service was encapsulated as separate Docker images. This approach allowed us to isolate each service along with its dependencies, ensuring compatibility and portability across different environments. By containerizing our services, we eliminated concerns regarding system dependencies and configuration discrepancies, thereby minimizing potential deployment issues.

Moreover, Docker containers facilitated consistent deployment practices across various environments, from local development to production servers. The uniformity achieved through Dockerization reduced the likelihood of compatibility issues and ensured seamless deployment workflows.

Kubernetes Orchestration:

Kubernetes was used to orchestrate and manage the deployment of Docker containers, providing robust capabilities for load balancing, scalability, and fault tolerance. Through Kubernetes YAML configuration files, we defined deployment specifications for each service, enabling automated scaling and redundancy.

With Kubernetes, our backend services were deployed across multiple instances, ensuring high availability and fault tolerance. Kubernetes' self-healing capabilities automatically detected and replaced failed containers, maintaining system availability and reliability.

Scalability and Performance Optimization:

By leveraging Kubernetes, we gained automatic scaling capabilities that dynamically adjusted resource allocation based on incoming traffic. As the demand for our services fluctuated, Kubernetes automatically scaled the backend services by provisioning additional containers, distributing the workload efficiently, and ensuring optimal resource utilization.

Additionally, the separation of the frontend and backend applications enabled by Docker and Kubernetes allowed for independent deployment and scalability. The frontend application, developed using React, was served separately, enabling easier management and scalability of frontend resources.

In conclusion, the integration of Docker and Kubernetes provided a robust infrastructure that enhanced the scalability, reliability, and performance of our educational platform. By adopting containerization and orchestration technologies, we achieved greater consistency, flexibility, and efficiency in our development and deployment processes, ultimately delivering a high-performance and fault-tolerant system.

4.4. Authentication and authorization mechanism used.

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

Whenever payments, enrollment routes are called, a middleware function is triggered and check whether user is available or not by AuthContext. This middleware function acts as a gatekeeper and first receives the request before validating the associated JWT. The JWT checked for its validity, ensuring that it hasn't been tampered with or expired. If the token is valid, the middleware function allows the request to proceed to the intended resource. However, if the token is not valid, the middleware function will throw an error with the message "Request is not authorized", preventing unauthorized access to the Unicat resources.

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

4.5. Development Tools and Technologies

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

4.6. Testing Methods

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

During Development

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

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

After Development

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

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

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

Appendix

(Please note that more than 17% of the similarity score of the report is from keywords of Nodejs.)

REST API

User Service

server.js

```
require('dotenv').config()

const express = require('express')
const mongoose = require('mongoose')
const userRoutes = require('./routes/user')
const cors = require('cors');

// express app
const app = express()

app.use(cors({ origin: 'http://localhost:5173' }));

// Parse JSON request bodies
app.use(express.json());

app.use((req, res, next) => {
    console.log(req.path, req.method)
    next()
})

app.use('/api/user', userRoutes)

// connect to db
mongoose.connect(process.env.MONGO_URI)
    .then(() => {
        // listen for requests
        app.listen(process.env.PORT, () => {
            console.log('connected to db & listening on port', process.env.PORT)
        })
    })
    .catch((error) => {
        console.log(error)
    })
```

userController.js

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

const createToken = (_id) => {
  return jwt.sign({ _id }, process.env.SECRET, { expiresIn: "3d" });
};

const loginUser = async (req, res) => {
  const { email, password } = req.body;

  try {
    const user = await User.login(email, password);

    // create a token
    const token = createToken(user._id);

    res.status(200).json({ user });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

// signup a user
const signupUser = async (req, res) => {
  const { email, password, role, contact, address } = req.body;

  try {
    const user = await User.signup(email, password, role, contact, address);

    // create a token
    const token = createToken(user._id);

    res.status(200).json({ email, token, role });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

// Retrieve user profile data by email
const getUserProfileByEmail = async (req, res) => {
  try {
    console.log("aedededededede-----" + req.body)
```

```
    const { email } = req.params;
    const user = await User.findOne({ email: req.body.email });
    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }
    res.json(user);
  } catch (error) {
    console.error("Error retrieving user profile:", error);
    res.status(500).json({ error: "Internal server error" });
  }
};

// Update user profile
const updateUserByEmail = async (req, res) => {
  try {
    const { email, contact, address } = req.body;
    const updatedUser = await User.findOneAndUpdate(
      { email: email },
      { $set: { contact: contact, address: address } },
      { new: true }
    );
    if (!updatedUser) {
      return res.status(404).json({ error: "User not found" });
    }
    console.log("User data updated successfully:", updatedUser);
    res.status(200).json({ message: "User data updated successfully", updatedUser });
  } catch (error) {
    console.error("Error updating user data:", error);
    res.status(500).json({ error: "Internal server error" });
  }
};

const getInstructors = async (req, res) => {
  try {
    // Fetch users with role as 'instructor'
    const instructors = await User.find({ role: 'instructor' });
    res.status(200).json(instructors);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

const getStudents = async (req, res) => {
  try {
    // Fetch users with role as 'student'
    const students = await User.find({ role: 'student' });
    res.status(200).json(students);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};


module.exports = {
  signupUser, loginUser, getUserProfileByEmail, updateUserByEmail,
  getInstructors,
```

```
  getStudents
};
```

User.js routes

```
const express = require('express')

// controller functions
const { loginUser, signupUser, getUserProfileByEmail,updateUserByEmail,getInstructors, getStudents } =
require('../controllers/userController')

const router = express.Router()

// login route
router.post('/login', loginUser)

// signup route
router.post('/signup', signupUser)

// getUserDataRoutes
router.post("/getUserData", getUserProfileByEmail);

// update route
router.put("/update", updateUserByEmail);

// getnst route
router.get('/getinst', getInstructors)

// get getstu route
router.get('/getstu', getStudents)

module.exports = router
```

userModel.js

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

const Schema = mongoose.Schema;

const userSchema = new Schema({
  userId: {
    type: String,
    unique: true,
    required: true
  },
  email: {
    type: String,
    required: true,
```

```
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  role: {
    type: String,
    required: true
  },
  contact: {
    type: String,
    required: true,
    validate: {
      validator: function (v) {
        // Validate contact number format with country code and plus mark
        return /^\+\d{1,3}\d{10}$/.test(v);
      },
      message: props => `${props.value} is not a valid contact number with country code`
    }
  },
  address:{
    type: String,
    required: false
  }
});

// Static signup method
userSchema.statics.signup = async function(email, password, role, contact, address) {
  // Validations
  if (!email || !password || !role || !contact) {
    throw Error('All fields must be filled');
  }
  if (!validator.isEmail(email)) {
    throw Error('Email not valid');
  }
  if (!validator.isStrongPassword(password)) {
    throw Error('Password not strong enough');
  }
  if(!validator.isMobilePhone(contact)) {
    throw Error('Invalid mobile number');
  }

  //Implement userID sequence
  const lastUser = await this.findOne({}, {}, { sort: { 'userId': -1 } });
  let userId;
  if (lastUser) {
    const lastUserId = parseInt(lastUser.userId.substr(1));
    userId = 'U' + (lastUserId + 1).toString().padStart(3, '0');
  } else {
    userId = 'U001';
  }

  //Validation for existing emails
  const exists = await this.findOne({ email });
```

```
  if (exists) {
    throw Error('Email already in use');
  }

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

  const user = await this.create({ userId, email, password: hash, role, contact, address });

  return user;
};

// Static login method
userSchema.statics.login = async function(email, password) {
  if (!email || !password) {
    throw Error('All fields must be filled');
  }

  const user = await this.findOne({ email });
  if (!user) {
    throw Error('Incorrect email');
  }

  const match = await bcrypt.compare(password, user.password);
  if (!match) {
    throw Error('Incorrect password');
  }

  return user;
};

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

requireAuth.js

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

const requireAuth = async (req, res, next) => {
  // verify user is authenticated
  const { authorization } = req.headers

  if (!authorization) {
    return res.status(401).json({error: 'Authorization token required'})
  }

  const token = authorization.split(' ')[1]

  try {
    const { _id } = jwt.verify(token, process.env.SECRET)

    req.user = await User.findOne({ _id }).select('_id')
    next()

  } catch (error) {
    console.log(error)
    res.status(401).json({error: 'Request is not authorized'})
  }
}

module.exports = requireAuth
```

.env

```
PORT=4003
MONGO_URI=mongodb+srv://lasenhewage111:lasen@cluster0.wikqzcj.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0
SECRET=ninjaddgojoshifuyoshimarioluigipeachbowser
```

Payment Service

Server.js

```
require('dotenv').config();

const express = require('express');
const cors = require('cors');
const paymentRoutes = require('./routes/paymentRoutes.js');
const mongoose = require('mongoose')

const app = express();

app.use(cors({ origin: 'http://localhost:5173' }));
app.use(express.json());

app.get("/", (req, res) => {
   res.send("It is working");
});

app.use('/', paymentRoutes);

mongoose.connect(process.env.MONGO_URI)
   .then(() => {
      // listen for requests
      app.listen(process.env.PORT, () => {
         console.log('connected to db & listening on port', process.env.PORT)
      })
   })
   .catch((error) => {
      console.log(error)
   })
```

paymentController.js

```
const stripe =
require("stripe")("sk_test_51PE7P8P6kZHWcO3DgNY5pT3TVuYMqYWyK0MJ2AYgW91p0bVDcReGYXVWuqgAjDWmoj6gpo0pjnuStW3
HM7i4lsMy00hxOzdGc0");
const { v4: uuidv4 } = require("uuid");
const Payment = require("../models/payment.js");

const processPayment = async (req, res) => {
  try {
    const { products, token, email } = req.body;
    const transactionKeys = uuidv4();

    // Calculate total amount
    const totalAmount = products.reduce((total, product) => total + product.price, 0);

    // Extract courseIds from products
    const courseIds = products.map(product => product.courseId).join(', ');

    // Create a charge for the total amount
    const charge = await stripe.charges.create({
      amount: totalAmount * 100,
      currency: "LKR",
      source: token.id,
      description: courseIds
    });
    // Save payment details in MongoDB
    const payment = new Payment({
      paymentId: charge.id,
      totalAmount: totalAmount,
      currency: charge.currency,
      description: courseIds ,
      email: email
    });
    console.log("---Payment---", payment);
    await payment.save();

    res.json({ message: "Stripe payment processed successfully", charge });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "An error occurred while processing the payment" });
  }
};

// Process single payment
const processSinglePayment = async (req, res) => {
  try {
    const { products, token, email } = req.body;
    const transactionKeys = uuidv4();

    const charge = await stripe.charges.create({
      amount: products.price * 100,
```

```
            currency: "LKR",
            source: token.id,
            description: products.name
        });
        // Save payment details in MongoDB
        const payment = new Payment({
            paymentId: charge.id,
            totalAmount: products.price,
            currency: charge.currency,
            description: products.name ,
            email: email
        });
        console.log("---Payment---", payment);
        await payment.save();

        res.json({ message: "Stripe payment processed successfully", charge });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: "An error occurred while processing the payment" });
    }
};

// Retrieve payment details
const getPayments = async (req, res) => {
    try {
        let payments;
        if (req.body.role === "admin") {
            // Fetch all payments if user is admin
            payments = await Payment.find();
        } else {
            // Fetch payments based on logged email if user is not admin
            payments = await Payment.find({ email: req.body.email });
        }
        res.json(payments);
    } catch (error) {
        console.error("Error fetching payments:", error);
        res.status(500).json({ error: "An error occurred while fetching payments" });
    }
};

module.exports = {
    processPayment,
    processSinglePayment,
    getPayments
};
```

Payment.js model

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

const paymentSchema = new Schema({
    paymentId: { type: String, required: true },
    totalAmount: { type: Number, required: true },
    currency: { type: String, required: true },
    description: { type: String, required: true },
    email: { type: String, required: true },
    createdAt: { type: Date, default: Date.now }
});

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

paymentRoutes.js

```
const express = require('express');
const router = express.Router();
const paymentController = require('../controllers/paymentController.js');

// Process payment route
router.post("/payment", paymentController.processPayment);

// Get payments route
router.post("/getPayments", paymentController.getPayments);

// Process single payment route
router.post("/singlePayment", paymentController.processSinglePayment);

module.exports = router;
```

Course Service

Controllers

courseContentController.js

```
const CourseContent = require('../models/courseContentModel');
const mongoose = require('mongoose');
const multer = require('multer');
const path = require('path');

//get the content
const getCourseContent = async (req, res) => {
    const courseContents = await CourseContent.find({}).sort({createdAt: -1});

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

const getSingleContent = async (req, res) => {
    const { id } = req.params;

    if(!mongoose.Types.ObjectId.isValid(id)){
        return res.status(404).json({error: 'CourseContent not found'})
    }

    const courseContent = await CourseContent.find({courseId: id});

    if(!courseContent){
        return res.status(404).json({error: 'CourseContent not found'})
    }

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

//create the content
const createCourseContent = async (req, res) => {
    const {title, courseId} = req.body
    // console.log(req.body)
    let file = null

    let emptyFields = []

    if(!req.file){
        emptyFields.push('file')
    } else {
        file = req.file.path
    }

    if (!title) {
        emptyFields.push('title')
    }
    if (emptyFields.length > 0) {
```

```
        return res.status(400).json({ error: 'Please fill the required fields', emptyFields});
    }

    //add to db
    try{
        const courseContent = await CourseContent.create({title, file, courseId})
        res.status(200).json({courseContent});
    }catch (error) {
        res.status(400).json({ error: error.message })
    }
}

//delete course contents
const deleteCourseContents = async (req, res) => {
    const { id } = req.params

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

    const courseContents = await CourseContent.findOneAndDelete({ _id: id })

    if(!courseContents){
        return res.status(404).json({ error: 'No content found'})
    }

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

module.exports = {
    getCourseContent,
    createCourseContent,
    deleteCourseContents,
    getSingleContent
}
```

courseController.js

```
const Course = require('../models/courseModel');
const mongoose = require('mongoose');
const multer = require('multer');
const path = require("path")

//get all courses
const getAllCourses = async (req, res) => {
    const courses = await Course.find({}).sort({createdAt: -1})

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

//get a single course
const getCourse = async (req, res) => {
    const { id } = req.params

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

    const course = await Course.findById(id)

    if(!course){
        return res.status(404).json({error: 'No such course'})
    }

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

const getTeacherCourse = async (req, res) => {
    const { id } = req.params;
    console.log(id);



    try {
        // Assuming Course schema has a 'teacherId' field
        const courses = await Course.find({ teacher: id });

        if (!courses || courses.length === 0) {
            return res.status(404).json({ error: 'No courses found for this teacher' });
        }

        res.status(200).json(courses);
    } catch (error) {
        console.error('Error fetching teacher courses:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
```

```
};


const getAcceptedTeacherCourses = async (req, res) => {
    const { id } = req.params;
    console.log(id);

    try {
        // Assuming Course schema has a 'teacherId' field and a 'status' field
        const courses = await Course.find({ teacher: id, status: 'Accepted' });

        if (!courses || courses.length === 0) {
            return res.status(404).json({ error: 'No accepted courses found for this teacher'
});
        }

        res.status(200).json(courses);
    } catch (error) {
        console.error('Error fetching accepted teacher courses:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
};


const getRejectedTeacherCourses = async (req, res) => {
    const { id } = req.params;
    console.log(id);

    try {
        // Assuming Course schema has a 'teacherId' field and a 'status' field
        const courses = await Course.find({ teacher: id, status: 'Pending' });

        if (!courses || courses.length === 0) {
            return res.status(404).json({ error: 'No accepted courses found for this teacher'
});
        }

        res.status(200).json(courses);
    } catch (error) {
        console.error('Error fetching accepted teacher courses:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
};

const getTeacherCourseCount = async (req, res) => {
    const { id } = req.params;
    console.log(id);

    try {
        // Assuming Course schema has a 'teacherId' field
        const courseCount = await Course.countDocuments({ teacher: id });

        res.status(200).json({ courseCount });
```

```
    } catch (error) {
        console.error('Error fetching teacher course count:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
};


//create a new course
const createCourse = async (req, res) => {
    const {name, courseCode, duration, description,price,teacher} = req.body
    console.log(req.file)
    let file = null

    let emptyFields = []

    if(!req.file){
        emptyFields.push('file')
    }else {
        file = req.file.path
    }

    if(!name){
        emptyFields.push('name')
    }
    if(!courseCode){
        emptyFields.push('courseCode')
    }
    if(!duration){
        emptyFields.push('duration')
    }
    if(!description){
        emptyFields.push('description')
    }
    if(emptyFields.length > 0){
        return res.status(400).json({error: 'Please fill in all the fields', emptyFields })
    }

    //add documentat to db
    try{
        const course = await Course.create({name, courseCode, duration, description,
file,price,teacher})
        res.status(200).json({course})
    }catch(error){
        res.status(400).json({error: error.message})
    }
}


//delete a course
const deleteCourse = async (req, res) => {
    const { id } = req.params

    if(!mongoose.Types.ObjectId.isValid(id)){
```

```
        return res.status(404).json({error: 'No such course'})
    }

    const course = await Course.findOneAndDelete({_id: id})

    if(!course){
        return res.status(404).json({error: 'No such course'})
    }

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

//update a course
const updateCourse = async (req, res) => {
    const { id } = req.params

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

    const course = await Course.findOneAndUpdate({_id: id}, {
        ...req.body
    })

    if(!course){
        return res.status(404).json({error: 'No such course'})
    }

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

const AcceptCourse = async (req, res) => {
    const { id } = req.params;

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

    try {
        let course = await Course.findOneAndUpdate(
            { _id: id },
            { ...req.body, status: 'Accepted' }, // Update the status to 'accept'
            { new: true } // Return the updated document
        );

        if (!course) {
            return res.status(404).json({ error: 'No such course' });
        }

        res.status(200).json(course);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
```

```
}

const DeclineCourse = async (req, res) => {
    const { id } = req.params;

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

    try {
        let course = await Course.findOneAndUpdate(
            { _id: id },
            { ...req.body, status: 'Rejected' }, // Update the status to 'accept'
            { new: true } // Return the updated document
        );

        if (!course) {
            return res.status(404).json({ error: 'No such course' });
        }

        res.status(200).json(course);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}


module.exports = {
    getAllCourses,
    getCourse,
    createCourse,
    deleteCourse,
    updateCourse,
    getTeacherCourse,
    AcceptCourse,
    DeclineCourse,
    getTeacherCourseCount,
    getAcceptedTeacherCourses,
    getRejectedTeacherCourses
}
```

Middleware

multerCourseContent.js

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

const storage = multer.diskStorage({
    destination: function (req, file, cb) {
        //destination of content storage
        cb(null, './uploads/lectures/')
    },
    filename: function (req, file, cb) {
        cb(null, file.originalname)
    }
})

const fileFilter = (req, file, cb) => {
    if (
        file.mimetype === 'application/pdf' ||
        file.mimetype === 'application/zip' ||
        file.mimetype === 'video/mp4' ||
        file.mimetype === 'video/webm' ||
        file.mimetype === 'video/3gpp'
    ) {
        cb(null, true);
    } else {
        cb(new Error('Invalid file type'), false);
    }
};


const upload = multer({
    storage: storage,
    limits: {
        fileSize: 1024 * 1024 * 500
    },
    fileFilter: fileFilter
})

module.exports = upload
```

multerTraining.js

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

const storage = multer.diskStorage({
    destination: function (req, file, cb) {
        //destination of file storage
        cb(null, './uploads/thumbnails/');
    },
    filename: function (req, file, cb) {
        cb (null, file.originalname)
    },
});

const fileFilter = (req, file, cb) => {
    //reject files if its not a jpg, png or pdf file
    if(
        file.mimetype === 'image/jpeg' ||
        file.mimetype === 'image/jpg' ||
        file.mimetype === 'image/png'
        // file.mimetype === 'video/mp4' ||
        // file.mimetype === 'application/pdf'
    ){
        cb(null, true);
    } else {
        cb(null, false);
    }
}

const upload = multer({
    storage: storage,
    limits: {
        fileSize: 1024 * 1024 * 10
    },
    fileFilter: fileFilter
})

module.exports = upload;
```

Models

courseContentModel.js

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

const Schema = mongoose.Schema

const courseContentSchema = new Schema({
    title: {
        type: String,
        required: true
    },
    file: {
        type: String,
        required: [true, 'Provide content'],
    },
    courseId:{
        type: Schema.Types.ObjectId,
        ref: 'courses'
    }
}, {timestamps: true});

module.exports = mongoose.model('CourseContent', courseContentSchema)
```

courseModel.js

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

const Schema = mongoose.Schema

const courseSchema = new Schema({
    name:{
        type:String,
        required:true
    },
    courseCode:{
        type:String,
        required:true
    },
    price:{
        type:Number,
        required:true
    },
    duration:{
        type:Number,
        required:true
    },
    description:{
        type:String,
        required:true
    },
```

```
    file: {
        type:String,
        required:[true, 'Provide an image'],
    },
    teacher:{
        type:String,
        required:true
    },
    status: {
        type: String,
        default: 'pending' // Set default value to 'pending'
    }
}, {timestamps: true});




module.exports = mongoose.model('Course', courseSchema)
```

Routes

courseContents.js

```
const multer = require('multer');
const upload = require('../middleware/multerCourseContent');
const express = require('express');

const {
    getCourseContent,
    createCourseContent,
    deleteCourseContents,
    getSingleContent
} = require('../controllers/courseContentController');

const router = express.Router();

router.get('/', getCourseContent);

router.get('/:id', getSingleContent);

router.route('/').get(getCourseContent).post(upload.single('file'), createCourseContent);

router.delete('/:id', deleteCourseContents);

module.exports = router;
```

Course.js

```
const multer = require('multer');
const upload = require('../middleware/multerTraining')
const express = require('express');

const {
    createCourse,
    getAllCourses,
    getCourse,
    deleteCourse,
    updateCourse,
    getTeacherCourse,
    AcceptCourse,
    DeclineCourse,
    getTeacherCourseCount,
    getAcceptedTeacherCourses,
    getRejectedTeacherCourses
} = require('../controllers/courseController');

const router = express.Router();

//GET all courses
router.get('/', getAllCourses);


//GET all courses
router.get('/teacher/:id', getTeacherCourse);

//GET a single course
router.get('/:id', getCourse);

//POST a new course
//router.post('/', createCourse).post(upload.single('file'), createCourse)
router.route('/').get(getCourse).post(upload.single('file'), createCourse)

//DELETE a course
router.delete('/:id', deleteCourse)

//UPDATE a course
router.patch('/:id', updateCourse)

router.patch('/accept/:id', AcceptCourse)

router.patch('/decline/:id', DeclineCourse)

router.get('/coursecount/:id',getTeacherCourseCount)
router.get('/acceptcount/:id',getAcceptedTeacherCourses)
router.get('/rejectcount/:id',getRejectedTeacherCourses)
```

```
module.exports = router;
```

Utills

appError.js

```
class AppError extends Error {
    constructor(message, statusCode) {
      super(message);

      this.statusCode = statusCode;
      this.status = `${statusCode}`.startsWith('4') ? 'fail' : 'error';
      this.isOperational = true;

      Error.captureStackTrace(this, this.constructor);
    }
  }

  module.exports = AppError;
```

Lerner Service

Server.js

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const { success, error } = require("consola");

const { MONGO_URI, PORT } = require("./config/index");

//export routes
const enrollRoutes = require("./routes/enrollment");

const app = express();
app.use(cors());

app.use(express.json());

app.use((req, res, next) => {
  console.log(req.path, req.method);
  next();
});

app.use("/api/enroll/", enrollRoutes);
```

```
//connecting to DB and starting server
const startApp = async () => {
  try {
    await mongoose
      .connect(MONGO_URI, {
        useNewUrlParser: true,
        useUnifiedTopology: true,
      })
      .then(() => {
        app.listen(PORT, () => {
          success({
            message: `Successfully connected to database and server running on ${PORT}`,
            badge: true,
          });
        });
      })
      .catch((err) => {
        console.log("Error: ", err);
      });
  } catch (error) {
    error({ message: `Error connecting to database: ${error.message}` });
  }
};

startApp();
```

index.js

```
require("dotenv").config();

module.exports = {
  MONGO_URI: process.env.MONGO_URI,
  PORT: process.env.PORT,
};
```

enrollmentModel.js

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

const enrollmentSchema = new Schema(
  {
    userId: {
      type: "String",
      required: true,
    },
    courseId: {
      type: "String",
      required: true,
    },
```

```
    courseName: {
      type: "String",
      required: true,
    },
    count: {
      type: Number,
      required: true,
    },
    outOf: {
      type: Number,
      required: true,
    },
  },
  { timestamps: true }
);

module.exports = mongoose.model("enrollCourses", enrollmentSchema);
```

cartModel.js

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

const Schema = mongoose.Schema;

const cartSchema = new Schema(
  {
    courseId: {
      type: "String",
      required: true,
    },
    courseName: {
      type: "String",
      required: true,
    },
    userId: {
      type: "String",
      required: true,
    },
    price: {
      type: Number,
      required: true,
    },
  },
  { timestamps: true }
);

module.exports = mongoose.model("Cart", cartSchema);
```

enrollmentController.js

```
require("dotenv").config();
const Enroll = require("../models/enrollmentModel");
const Course = require("../models/courseModel");
const Cart = require("../models/cartModel");
const _ = require("lodash");
const mongoose = require("mongoose");

//enrolling to a new course
const enrollCourse = async (req, res) => {
  try {

    const { userId, courseCode } = req.params;
    const { enrollKey } = req.body;


    const courseDetails = await Course.findOne({ _id:courseCode });

    console.log(`Course: ${courseDetails}, enrollKey: ${enrollKey}`);

    if (!courseDetails) {
      return res.status(404).json({ message: "Course not found" });
    }

    if (true) {

      // Check if the user is already enrolled
      const enrolledCheck = await Enroll.findOne({
        userId: userId,
        courseId: courseDetails._id
      });


      if (!_.isEmpty(enrolledCheck)) {
        return res
          .status(409)
          .json({ message: "You are already enrolled in this course." });
      }


      // Perform enrollment logic here
      const count = 0;
      console.log(courseDetails.name);

      const enrolling = await Enroll.create({
        userId: userId,
        courseId: courseDetails._id,
        courseName: courseDetails.name,
        count: count,
        outOf: courseDetails.duration,
```

```
      });

      const mail = [];

      const subject = "Course enrollment";
      const message = `${userId}, you are successfully enrolled into ${courseDetails.name}
course.`;

      mail.push(subject, message, userId);
      return res.status(200).json(mail);

    // } else {
    //   console.log("Invalid enrollKey");
    //   return res.status(400).json({ message: "Invalid enrollKey" });
    // }
    }
  } catch (error) {
    return res.status(400).json({ error: error.message });
  }
};

const getEnrollments = async (req, res) => {
  try {
    // Assuming Enroll model is imported properly
    const myEnrolls = await Enroll.find({ /* Add filtering criteria if necessary */ });

    if (myEnrolls.length === 0) {
      return res.status(404).json({ message: "No enrollments found." });
    }

    return res.status(200).json(myEnrolls);
  } catch (error) {
    console.error("Error retrieving enrollments:", error);
    res.status(500).json({ error: "Internal server error" });
  }
};

const getStudentsCountPerCourse = async (req, res) => {
  try {
      const studentCounts = await Enroll.aggregate([
          {
              $group: {
                  _id: '$courseId',
                  students: { $addToSet: '$userId' } // Collect unique userIds per course
              }
          },
          {
              $project: {
                  courseId: '$_id',
                  studentCount: { $size: '$students' } // Calculate the count of unique
```

```
userIds per course
            }
        }
    ]);

    // If there are no enrollments or no students in any course
    if (!studentCounts || studentCounts.length === 0) {
        return res.status(404).json({ error: 'No enrollments found' });
    }

    res.status(200).json(studentCounts);
  } catch (error) {
    console.error('Error fetching student counts per course:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
};


const myLernings = async (req, res) => {
  try {
    const { userId } = req.params;

    const myEnrolls = await Enroll.find({ userId: userId });

    const enrolls = [];

    for (const enroll of myEnrolls) {
      enrolls.push({
        courseId: enroll.courseId,
        courseName: enroll.courseName,
      });
    }

    return res.status(200).json(enrolls);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

const cancelEnrollment = async (req, res) => {
  try {
    const { userId, courseId } = req.params;

    const record = await Enroll.findOne({ userId: userId, courseId: courseId });
    console.log(record._id);

    const courseDetails = await Course.findOne({ _id:courseId });

    await Enroll.findOneAndDelete({ _id: record._id });
```

```
    const mail = [];

    const subject = "Course Unenrollment";
    const message = `${userId}, you are successfully unenrolled from ${courseDetails.name}
course.`;

    mail.push(subject, message, userId);

    return res.status(200).json(mail);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

const addToCart = async (req, res) => {
  const { userId, courseId } = req.params;
  console.log(userId, courseId);
  try {
    const courseDetails = await Course.findOne({ _id:courseId  });

    if (!courseDetails) {
      return res.status(404).json({ message: "Course not found" });
    } else {
      const { name } = courseDetails;

      // Create cart item with userId, courseId, and courseName
      const cartItem = await Cart.create({
        userId: userId,
        courseId: courseId,
        courseName: name,
        price: courseDetails.price,
      });

      return res.status(200).json(cartItem);
    }
  } catch (error) {
    return res
      .status(404)
      .json({ message: "Error in adding to cart: ", error });
  }
};

const myCart = async (req, res) => {
  try {
    const { userId } = req.params;
    const myItems = await Cart.find({ userId: userId });

    if (!myItems) {
      return res.status(404).json({ message: "Item not found" });
    } else {
```

```
      const mycartItems = [];

      for (const cart of myItems) {
        mycartItems.push({
          courseId: cart.courseId,
          courseName: cart.courseName,
          price: cart.price,
        });
      }

      console.log(mycartItems);

      return res.status(200).json(mycartItems);
    }
  } catch (error) {
    return res.status(404).json({ message: "Error in myCart: ", error });
  }
};

const removeCart = async (req, res) => {
  try {
    const { userId, courseId } = req.params;
    console.log(userId, courseId);
    const record = await Cart.findOne({
      userId: userId,
      courseId: courseId,
    });
    console.log(record);
    await Cart.findByIdAndDelete({ _id: record._id });
    return res.status(200).json(record);
  } catch (error) {
    return res.status(404).json({ message: "Error in removeCart: ", error });
  }
};

const trackProgress = async (req, res) => {
  try {
    const { userId, courseId } = req.params;
    const { count, outOf } = req.body;

    const record = await Enroll.findOne({ userId: userId, courseId: courseId });
    const newCount = count;

    await Enroll.findOneAndUpdate(
      { _id: record._id },
      {
        count: newCount,
      }
    );
```

```
    const progress = (newCount / outOf) * 100;

    console.log(`trackProgress: [${userId}-${courseId}]-${progress}`);
    return res.status(200).json(progress);
  } catch (error) {
    return res.status(404).json({ message: "Error in trackProgress: ", error });
  }
};

const getProgress = async (req, res) => {
  try {
    const { userId, courseId } = req.params;
    const record = await Enroll.findOne({ userId: userId, courseId: courseId });
    const p = Math.round((record.count / record.outOf) * 100);
    console.log(`getProgress: ${p}`);
    return res.status(200).json(p);
  } catch (error) {
    return res.status(404).json({ message: "Error in getProgress: ", error });
  }
};



module.exports = {
  enrollCourse,
  cancelEnrollment,
  myLernings,
  addToCart,
  myCart,
  removeCart,
  trackProgress,
  getProgress,
  getEnrollments,
  getStudentsCountPerCourse
};
```

enrollment.js

```
const express = require("express");
const router = express.Router();
const {
  enrollCourse,
  cancelEnrollment,
  myLernings,
  addToCart,
  myCart,
  removeCart,
  trackProgress,
  getProgress,
  getEnrollments,
```

```
  getStudentsCountPerCourse
} = require("../controllers/enrollmentController");

//enrolling into a new course
router.post("/:userId/:courseCode", enrollCourse);

//cancel enrollment
router.delete("/:userId/:courseId/cancel", cancelEnrollment);

//myLernings
router.get("/:userId/myLernings", myLernings);

router.post("/addCart/:userId/:courseId", addToCart);

router.get("/myCart/:userId", myCart);
router.get("/studentpercourse", getStudentsCountPerCourse);

router.get("/allenroll", getEnrollments);

router.delete("/remove/:userId/:courseId/", removeCart);

router.patch("/myProgress/:userId/:courseId", trackProgress);

router.get("/getProgress/:userId/:courseId", getProgress);

module.exports = router;
```