ONLINE LEARNING PLATFORM

Submitted by **TEAM NM ID: NM2024TMID16440**

NAME:	**NM ID:**
SADHANA PRIYA R	7F7E1602E8BB0488BB74CDA03B4D7361
SARAJ RAJA S	C070F1EA9E6F01A9A724F8DEED1F4575
SHALINI J	74E3A91CC3650C4810E4ADA1DFC225FB
SIVADHANUSH R	4CE9818E476CFCE80F247856EA055A8F

In partial completion towards the attainment of the degree of

BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING

THANGAVELU ENGINEERING COLLEGE

KARAPAKKAM, CHENNAI – 600097

ANNA UNIVERSITY

CHENNAI – 600025

1. **Introduction**

The purpose of a learning application is to serve as a dynamic educational tool designed to enhance the learning experience by leveraging the capabilities of modern technology. These applications are crafted to provide learners with an interactive, adaptive, and engaging platform that goes beyond traditional educational methods. They aim to make learning more accessible, flexible, and efficient, allowing users to gain knowledge and skills at their own pace, anytime and anywhere. This flexibility not only caters to diverse learning needs but also helps overcome barriers like time constraints, location, or lack of resources.

Learning applications are developed with a set of specific goals to maximize educational impact. One of the primary goals is to improve knowledge retention by using interactive methods such as quizzes, videos, games, and simulations, which help reinforce concepts in a more memorable way than passive learning. By integrating real-time feedback and assessments, these applications help learners monitor their progress, identify gaps in understanding, and continuously adjust their learning strategies.

Another key goal is to support personalized learning experiences. By utilizing adaptive algorithms, learning applications can tailor content to match the learner's proficiency level, learning style, and interests. This individualized approach not only increases motivation but also ensures that learners remain engaged and challenged, optimizing their potential for growth. Furthermore, these apps encourage self-directed learning, empowering users to take charge of their education and explore topics beyond the confines of a traditional classroom.

In addition to academic learning, these applications are designed to develop essential life skills, such as problem-solving, critical thinking, and digital literacy. They also promote collaboration through features like discussion forums, peer feedback, and project-sharing, fostering a sense of community and social interaction in a digital learning environment. Ultimately, the overarching goal of a learning application is to create a more inclusive and effective educational experience, supporting learners in their pursuit of knowledge, skills, and lifelong learning.

The purpose of E-learning Management System is to automate the existing manual sys by the help of computerized equipment's and full-fledged computer software, fulfil their requirements, so that their valuable data/information can be stored for a longer pea with easy accessing and manipulation of the same. The required software and hardware easily available and easy to work with. E-learning Management System, as described above, can lead to error free, secure, reliable and fast management system. It can assist user to concentrate on their other activities rather to concentrate on the record keeping Thus it will help organization in better utilization of resources. The organization can maintain computerized records without redundant entries. That means that one need not distracted by information that is not relevant, while being able to reach the information.

Functionalities provided by E-learning Management System:

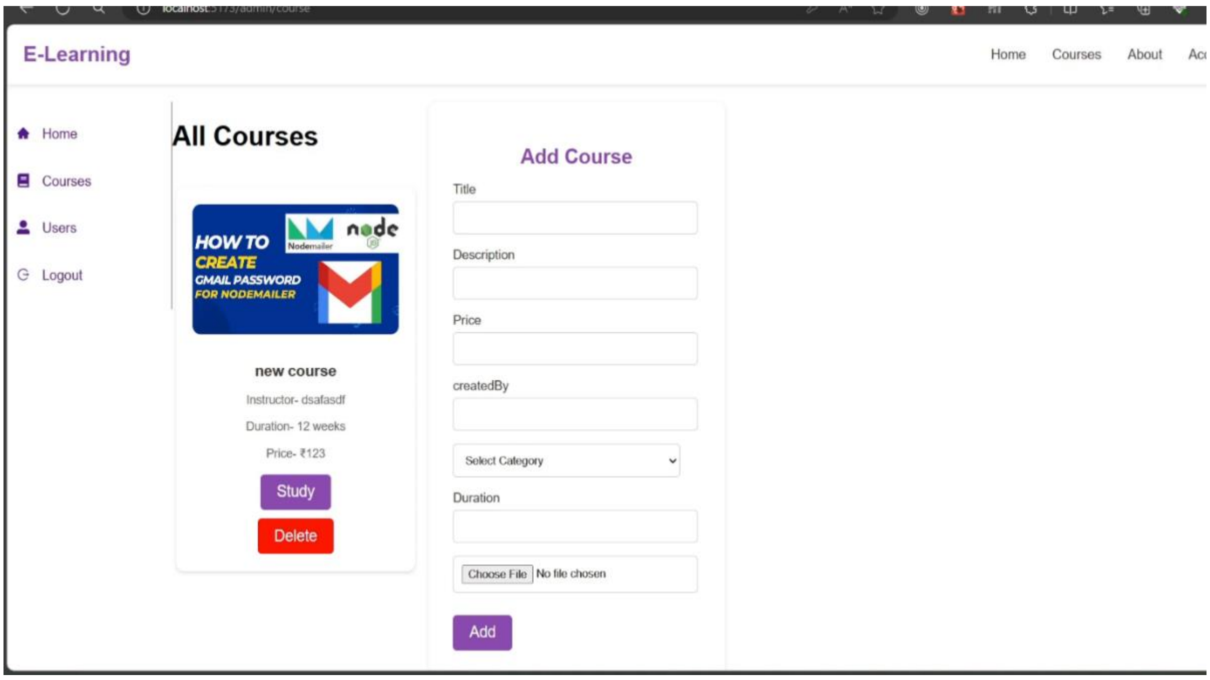
Provides the searching facilities based on various factors. Such as Assignment TEACHER, QUIZ, QUESTION.

Manage the information of Student.

Manage the information of Assignment

Editing, adding and updating of Records is improved which results in proper resource management of data.

Integration of all records .



All Courses

new course
Instructor- dsafasdf
Duration- 12 weeks
Price- ₹123

2. Key Features

An effective e-learning platform incorporates a range of features designed to enhance the learning experience, foster engagement, and support learners in achieving their educational goals. Here are some of the key features commonly found in successful e-learning platforms:

1. User-Friendly Interface

- An intuitive, easy-to-navigate interface ensures that learners of all ages and skill levels can access content seamlessly, reducing barriers to entry and allowing users to focus on learning rather than struggling with the platform.

2. Personalized Learning Paths

- Adaptive learning algorithms assess users' strengths, weaknesses, and preferences to customize learning journeys. This personalization keeps learners engaged by offering content suited to their current skill level and interests.

3. Interactive Content and Multimedia Support

- The use of videos, animations, infographics, quizzes, and simulations makes learning more engaging and effective by catering to different learning styles. Interactive elements like drag-and-drop exercises or clickable graphics enhance user interaction.

4. Assessments and Quizzes

- Integrated assessments, such as quizzes, tests, and assignments, help learners track their progress, reinforce concepts, and provide instant feedback, allowing for continuous improvement.

5. Gamification

- Elements like badges, leaderboards, rewards, and challenges help motivate learners and boost engagement by turning the learning process into an enjoyable, game-like experience.

6. Progress Tracking and Analytics

- Real-time dashboards and analytics provide insights into learner performance, helping both users and educators identify strengths, gaps, and areas for improvement. It also enables institutions to evaluate the effectiveness of their content.

7. Social Learning and Collaboration Tools

- Features like discussion forums, chat rooms, peer-to-peer messaging, and group projects encourage collaboration, allowing learners to interact, share knowledge, and learn from each other.

8. Mobile Compatibility

- A responsive design ensures the platform works seamlessly across various devices, including smartphones and tablets, enabling learners to access content on the go and learn at their own convenience.

9. Offline Access

- The ability to download lessons or resources for offline use ensures continuous learning even in environments with limited or no internet connectivity.

10. Content Management System (CMS)

- A robust CMS allows instructors to upload, organize, and update content easily. This feature ensures that course materials remain current and relevant.

11. Live Classes and Webinars

- Built-in tools for live streaming, virtual classrooms, and webinars support real-time interaction between instructors and students, mimicking the experience of traditional classrooms.

12. Certification and Badging

- Upon course completion, learners receive certificates or digital badges, which can be shared on professional platforms like LinkedIn, helping to demonstrate acquired skills and achievements.

13. Secure and Scalable Platform

- Strong data privacy, secure login mechanisms, and scalability to accommodate a growing number of users ensure that the platform remains robust, reliable, and safe for all users.

14. Multilingual Support

- To cater to a global audience, e-learning platforms often include support for multiple languages, ensuring inclusivity and accessibility for learners worldwide.

15. AI-Powered Support and Chatbots

- Integrated AI chatbots provide 24/7 assistance to users, answering questions, providing recommendations, and enhancing the overall user experience.

By integrating these features, e-learning platforms can create an enriching, versatile, and engaging learning environment that caters to diverse educational needs and supports continuous skill development.

3. Architecture

The provided React code is part of an admin dashboard for managing courses in an e-learning platform. It allows the admin to view all existing courses and add new courses to the platform. Let's break down the architecture and explain the functionality of each part.

1. High-Level Overview

- The component is named `AdminCourses` and is responsible for displaying existing courses and providing a form for admins to add new courses.

- The component is structured into two main sections:

  - Left Panel Displays all available courses.

  - Right Panel: Contains a form for adding new courses.

2. Dependencies and Imports

```javascript
import React, { useState } from "react";

import Layout from "../Utils/Layout";

import { useNavigate } from "react-router-dom";

import { CourseData } from "../../context/CourseContext";

import CourseCard from "../../components/coursecard/CourseCard";

import "./admincourses.css";
import toast from "react-hot-toast";

import axios from "axios";

import { server } from "../../main";
```

- React and useState are used for managing state and rendering components.

- useNavigate from `react-router-dom` is used to redirect users.

- CourseData is a context hook that fetches all courses.

- CourseCard is a component that displays individual course information.

- toast from `react-hot-toast` is used for notifications.

- axios is used to make HTTP requests to the backend.

- server is imported to get the base URL for API calls.

3. Authorization Check

```javascript
const navigate = useNavigate();

if (user && user.role !== "admin") return navigate("/");
```

- The component checks if the logged-in user is an admin. If not, it redirects them to the home page.

4. State Variables

```javascript
const [title, setTitle] = useState("");

const [description, setDescription] = useState("");

const [category, setCategory] = useState("");

const [price, setPrice] = useState("");

const [createdBy, setCreatedBy] = useState("");
const [duration, setDuration] = useState("");

const [image, setImage] = useState("");

const [imagePrev, setImagePrev] = useState("");

const [btnLoading, setBtnLoading] = useState(false);
```

- State variables are used to capture input values from the form, handle file uploads, and manage loading states.

5. File Upload Handling

```javascript
const changeImageHandler = (e) => {

  const file = e.target.files[0];

  const reader = new FileReader();


  reader.readAsDataURL(file);


  reader.onloadend = () => {

    setImagePrev(reader.result);

    setImage(file);

  };

};
```

- When an admin selects an image file, it reads the file as a Data URL using `FileReader` and stores it for preview (`imagePrev`) and upload (`image`).

6. Fetching Courses from Context

```javascript
const { courses, fetchCourses } = CourseData();
```

- This uses the `CourseData` context to access the list of courses and a method to refresh the courses after adding a new one.

7. Form Submission Handler

```javascript
const submitHandler = async (e) => {
  e.preventDefault();
  setBtnLoading(true);

  const myForm = new FormData();
  myForm.append("title", title);
  myForm.append("description", description);
  myForm.append("category", category);
  myForm.append("price", price);
  myForm.append("createdBy", createdBy);
  myForm.append("duration", duration);
  myForm.append("file", image);

  try {
    const { data } = await axios.post(`${server}/api/course/new`, myForm, {
      headers: {
        token: localStorage.getItem("token"),
      },
    });

    toast.success(data.message);
    setBtnLoading(false);
    await fetchCourses();

    // Reset form fields
    setImage("");
    setTitle("");
    setDescription("");
    setDuration("");
    setImagePrev("");
    setCreatedBy("");
    setPrice("");
    setCategory("");
  } catch (error) {
    toast.error(error.response.data.message);
    setBtnLoading(false);
  }
};
```

- This function handles the form submission to create a new course:
  - It uses `FormData` to handle the form data, especially the file upload.
  - Sends a POST request to the backend endpoint to create a new course.

  - Shows a success or error notification using `toast`.

  - Resets the form fields upon success.

  - Fetches the updated list of courses to reflect changes immediately.

8. JSX Structure

The return statement renders the UI of the component.

- Provides a form to enter the course details and upload an image.
- Includes a file preview section.
- The form is submitted using the `submitHandler` function.

9. Styling

- The file `admincourses.css` is used for styling the component.
- Classes like `.admin-courses`, `.left`, `.right`, and `.add-course` are used for layout and design.

- `server` is a base URL defined in `../../main`.
- The token is retrieved from `localStorage` to authenticate the admin user.

This SCSS code styles a section for an "admin courses" page, which seems to consist of two main areas: a list of courses (`left`) and a form for adding or editing courses (`right`). Below is a breakdown of the code and its functionality.

The structure can be visualized as:

```
```

.admin-courses
├── .left
│   └── .dashboard-content
└── .right
    └── .course-form

1. `.admin-courses` Container

```scss
.admin-courses {
  display: flex;
  justify-content: center;
  flex-wrap: wrap;
  gap: 1rem;
}
```

`display: flex;`: This makes the `.admin-courses` container a flexbox, allowing its direct children to align horizontally by default.

- **`justify-content: center;`**: Centers the content horizontally within the `.admin-courses` container.

- **`flex-wrap: wrap;`**: Allows the content to wrap to the next line if it overflows the available space.

- **`gap: 1rem;`**: Adds a uniform gap of `1rem` between the flex items.

2. `.left` Section

```scss
```

```
.left {
  .dashboard-content {
    display: flex;
    justify-content: space-around;
    flex-wrap: wrap;
    gap: 20px;
    margin-top: 40px;
    margin-left: 5px;
  }
}
```

- .left`: Contains the course dashboard content.
  - `.dashboard-content`:
    - `display: flex;`: Sets this section as a flex container.
    - `justify-content: space-around;`: Distributes items with space around them.
    - ‘flex-wrap: wrap;`: Allows items to wrap onto new lines if necessary.
    - `gap: 20px;`: Adds a 20px space between items within the `.dashboard-content`.
    -`margin-top: 40px; margin-left: 5px;`: Adds space above and to the left of the content.

3. `.right` Section (Course Form)

```scss
.right {
  .course-form {
    background-color: #fff;
    padding: 30px;
```

    border-radius: 10px;

    box-shadow: 0px 2px 4px rgba(0, 0, 0, 0.1);

    text-align: center;

    width: 300px;

- `.right`: Contains the form for managing courses.

  - `.course-form`:

    - `background-color: #fff;`: Sets the background color to white.

    `padding: 30px;`: Adds padding inside the form.

    `border-radius: 10px;`: Rounds the corners of the form.

    `box-shadow: 0px 2px 4px rgba(0, 0, 0, 0.1);: Adds a subtle shadow for a lifted effect.

    text-align: center;`: Centers the text inside the form.

    width: 300px;`: Fixes width of the form.

4. Form Styling

```scss
h2 {

  font-size: 24px;

  color: #8a4baf;

  margin-bottom: 15px;

}
```

  `h2`: Styles the form heading.

  `font-size: 24px;`: Sets the heading size.

  `color: #8a4baf;`: Sets the heading color to a shade of purple.

`margin-bottom: 15px;`: Adds space below the heading.

```scss
form {
  text-align: left;

  label {
    display: block;
    margin-bottom: 5px;
    font-size: 15px;
    color: #333;
  }

  input,
  select {
    width: 92%;
    padding: 10px;
    margin-bottom: 15px;
    border: 1px solid #ccc;
    border-radius: 5px;
  }
}
```

`form`: Styles the course form itself.

`text-align: left;`: Aligns the form content to the left.

`label`: Styles the form labels.

display: block;`: Makes each label appear on its own line.

`margin-bottom: 5px;`: Adds space below the label.

`font-size: 15px; color: #333;`: Sets the font size and color.

`input, select`: Styles the input fields and dropdowns.

`width: 92%;`: Makes the inputs almost full-width (92% of the parent width).

`padding: 10px;`: Adds padding inside the input fields.

`margin-bottom: 15px; :Adds space below each input.

`border: 1px solid #ccc;`: Sets a light gray border.

border-radius: 5px;`: Rounds the corners of the inputs.

1. Import Statements

javascript

```
import React, { useEffect, useState } from "react";

import "./users.css";

import { useNavigate } from "react-router-dom";

import axios from "axios";

import { server } from "../../main";

import Layout from "../Utils/Layout";

import toast from "react-hot-toast";
```

- React Hooks:
  - useEffect: Used to run side effects, like fetching data when the component loads.

    - useState: Manages the state of the component.
- CSS File: The styles for this component are defined in "./users.css".
- Router:
    - useNavigate: Helps redirect users to other routes.
- Axios: Used for making HTTP requests.
- Server Endpoint: The server variable holds the base URL for your backend API.
- Layout Component: Layout is a wrapper component that likely provides a consistent structure for admin pages.
- Toast Notifications: react-hot-toast is used for displaying success/error notifications.

2. Component: AdminUsers

javascript

```
const AdminUsers = ({ user }) => {
  const navigate = useNavigate();
```

- This component accepts a user prop, which contains information about the currently logged-in user.
- useNavigate is used for navigation.

3. Access Control Check

javascript

```
if (user && user.mainrole !== "superadmin") return navigate("/");
```

- Role-Based Access: This ensures that only users with the role superadmin can access this page.
- If the logged-in user is not a superadmin, they are redirected to the homepage ("/").

4. State Management

javascript

```
const [users, setUsers] = useState([]);
```

- users: An array that stores the list of users fetched from the server.
- setUsers: A function to update the users state.

5. Fetching Users from the Server

javascript

```
async function fetchUsers() {
  try {
    const { data } = await axios.get(`${server}/api/users`, {
      headers: {
        token: localStorage.getItem("token"),
      },
    });
    setUsers(data.users);
  } catch (error) {
    console.log(error);
  }
}
```

- fetchUsers(): An asynchronous function that fetches all users from the backend API.
- Authorization: The request includes a token stored in localStorage to authenticate the admin user.
- Error Handling: Logs errors to the console if the request fails.

Triggering the Fetch

javascript

Copy code

```
useEffect(() => {
  fetchUsers();
}, []);
```

- useEffect is used to call fetchUsers() when the component mounts (i.e., loads for the first time).

6. Updating User Roles

javascript

```
const updateRole = async (id) => {
  if (confirm("are you sure you want to update this user role")) {
    try {
      const { data } = await axios.put(
        `${server}/api/user/${id}`,
        {},
        {
          headers: {
            token: localStorage.getItem("token"),
          },
        }
      );

      toast.success(data.message);
      fetchUsers(); // Refresh the users list after updating
    } catch (error) {
      toast.error(error.response.data.message);
    }
  }
};
```

- updateRole(): This function is called when an admin wants to change the role of a user.
- Confirmation Prompt: Uses confirm() to ensure that the admin really wants to update the role.
- API Call: Sends a PUT request to update the role of the specified user.
- Success Notification: Displays a toast notification if the update is successful.
- Error Handling: Displays an error notification if the update fails.

7. Rendering the Component

javascript

```
console.log(users);
return (
  <Layout>
    <div className="users">
      <h1>All Users</h1>
      <table border={"black"}>
        <thead>
          <tr>
            <td>#</td>
            <td>name</td>
            <td>email</td>
            <td>role</td>
            <td>update role</td>
          </tr>
        </thead>

        {users &&
          users.map((e, i) => (
            <tbody key={e._id}>
              <tr>
                <td>{i + 1}</td>
                <td>{e.name}</td>
                <td>{e.email}</td>
                <td>{e.role}</td>
                <td>
                  <button
                    onClick={() => updateRole(e._id)}
                    className="common-btn"
                  >
                    Update Role
                  </button>
                </td>
              </tr>
            </tbody>
          ))}
      </table>
    </div>
  </Layout>
);
```

- Layout: The entire content is wrapped inside the Layout component.
- Table:
    - Displays a list of users with columns for their name, email, role, and an option to update their role.
    - The updateRole() function is attached to the "Update Role" button.

8. Exporting the Component

javascript

Copy code

export default AdminUsers;

Back end

code snippet is implementing a set of Express.js controllers for managing an online course platform. It includes functionality to create courses, add and delete lectures, manage users, and update roles, all wrapped within a custom error-handling middleware called TryCatch. Let's break down each section of the code in detail to understand what it's doing:

1. Imports

javascript

Copy code

import TryCatch from "../middlewares/TryCatch.js";

import { Courses } from "../models/Courses.js";

import { Lecture } from "../models/Lecture.js";

import { rm } from "fs";

import { promisify } from "util";

import fs from "fs";

import { User } from "../models/User.js";

- TryCatch: A middleware function used to handle errors in asynchronous functions.

- Courses, Lecture, User: Mongoose models representing courses, lectures, and users.
- rm: A function to delete files from the file system.
- promisify: Converts callback-based functions to promises, allowing the use of async/await.
- fs: Node.js file system module, used for file operations like deleting files.

2. Course Creation (createCourse Function)

javascript

Copy code

```
export const createCourse = TryCatch(async (req, res) => {
  const { title, description, category, createdBy, duration, price } = req.body;
  const image = req.file;

  await Courses.create({
    title,
    description,
    category,
    createdBy,
    image: image?.path,
    duration,
    price,
  });

  res.status(201).json({
    message: "Course Created Successfully",
  });
});
```

- Purpose: Creates a new course.
- Request Body: Expects title, description, category, createdBy, duration, and price.
- Request File: Expects an uploaded image (if any).
- Response: Returns a success message with status 201.

3. Add Lectures to a Course (addLectures Function)

javascript

Copy code

```
export const addLectures = TryCatch(async (req, res) => {
  const course = await Courses.findById(req.params.id);

  if (!course)
    return res.status(404).json({ message: "No Course with this id" });

  const { title, description } = req.body;
  const file = req.file;

  const lecture = await Lecture.create({
    title,
    description,
    video: file?.path,
    course: course._id,
  });

  res.status(201).json({
    message: "Lecture Added",
    lecture,
  });
});
```

- Purpose: Adds a lecture to an existing course.
- Request Params: Course ID (req.params.id).
- Request Body: Expects title and description.
- Request File: Expects an uploaded video file.
- Response: Returns the created lecture and a success message.

4. Delete a Lecture (deleteLecture Function)

javascript

Copy code

```
export const deleteLecture = TryCatch(async (req, res) => {
  const lecture = await Lecture.findById(req.params.id);

  rm(lecture.video, () => {
    console.log("Video deleted");
  });

  await lecture.deleteOne();

  res.json({ message: "Lecture Deleted" });
});
```

- Purpose: Deletes a lecture by its ID.
- Request Params: Lecture ID (req.params.id).
- File Deletion: Uses the rm function to delete the video file from the file system.

- Response: Returns a success message.

5. Delete a Course (deleteCourse Function)

javascript

Copy code

```
const unlinkAsync = promisify(fs.unlink);


export const deleteCourse = TryCatch(async (req, res) => {
  const course = await Courses.findById(req.params.id);
  const lectures = await Lecture.find({ course: course._id });


  await Promise.all(
    lectures.map(async (lecture) => {
      await unlinkAsync(lecture.video);
      console.log("video deleted");
    })
  );


  rm(course.image, () => {
    console.log("image deleted");
  });


  await Lecture.find({ course: req.params.id }).deleteMany();
  await course.deleteOne();


  await User.updateMany({}, { $pull: { subscription: req.params.id } });
  res.json({ message: "Course Deleted" });
});
```

- Purpose: Deletes a course and all its associated lectures.
- Request Params: Course ID (req.params.id).
- File Deletion:
  - Deletes all video files associated with the lectures.
  - Deletes the course image.
- Database Updates:
  - Removes the course from all users' subscriptions.
- Response: Returns a success message.

6. Get Platform Statistics (getAllStats Function)

javascript

Copy code

```javascript
export const getAllStats = TryCatch(async (req, res) => {
  const totalCoures = (await Courses.find()).length;
  const totalLectures = (await Lecture.find()).length;
  const totalUsers = (await User.find()).length;

  const stats = { totalCoures, totalLectures, totalUsers };

  res.json({ stats });
});
```

- Purpose: Provides statistics for the platform, including total courses, lectures, and users.
- Response: Returns the statistics.

7. Get All Users (getAllUser Function)

javascript

Copy code

```
export const getAllUser = TryCatch(async (req, res) => {
  const users = await User.find({ _id: { $ne: req.user._id } }).select("-password");

  res.json({ users });
});
```

- Purpose: Fetches all users except the currently logged-in user.
- Response: Returns a list of users without their passwords.

8. Update User Role (updateRole Function)

javascript

Copy code

```
export const updateRole = TryCatch(async (req, res) => {
  if (req.user.mainrole !== "superadmin")
    return res.status(403).json({ message: "This endpoint is assigned to superadmin" });

  const user = await User.findById(req.params.id);

  if (user.role === "user") {
    user.role = "admin";
    await user.save();
    return res.status(200).json({ message: "Role updated to admin" });
  }
  if (user.role === "admin") {

    user.role = "user";

    await user.save();

    return res.status(200).json({ message: "Role updated" });

  }

});
```

- Purpose: Updates a user's role. Only a superadmin can access this endpoint.
- Request Params: User ID (req.params.id).
- Role Change:
  - user to admin
  - admin to user
- Response: Returns a success message.

Database

code defines a utility function to connect to a MongoDB database using the mongoose library in a Node.js application. Let's break down each part in detail:

Code Explanation

javascript

Copy code

```
import mongoose from "mongoose";


export const connectDb = async () => {

  try {

    await mongoose.connect(process.env.DB);

    console.log("Database Connected");

  } catch (error) {
    console.log(error);
  }
};
```

Detailed Breakdown

1. Importing mongoose:

javascript

```
import mongoose from "mongoose";
```

- Purpose: The mongoose library is used to connect to a MongoDB database and perform database operations (like querying, creating, updating, and deleting documents).
- Note: Ensure that the mongoose package is installed in your project (npm install mongoose).

2. Defining connectDb Function:

```
export const connectDb = async () => { ... }
```

- This is an asynchronous function that attempts to connect to a MongoDB database using Mongoose.

3. Database Connection:

javascript

```
await mongoose.connect(process.env.DB);
```

- mongoose.connect(): This method is used to establish a connection to the MongoDB database.
- process.env.DB: The database connection string is stored in an environment variable (DB). This is usually stored in a .env file to keep sensitive information (like database credentials) secure.

Example .env file:

```
DB=mongodb://localhost:27017/mydatabase
```

   - The connection string can vary depending on whether you're using a local database or a cloud database like MongoDB Atlas.

4. Success Message:

```
javascript

console.log("Database Connected");
```

   - If the connection is successful, a message "Database Connected" is logged to the console.

5. Error Handling:

```
javascript

} catch (error) {

  console.log(error);

}
```

   - If there's an error during the connection process, it gets caught in the catch block.
   - The error is logged to the console, helping you diagnose issues like:
     - Invalid connection string
     - Incorrect database credentials
     - Network issues preventing connection to the database server

How to Use This Function

1. Import and Call connectDb in Your Main Server File (index.js or app.js):

```
javascript

import express from "express";

import { connectDb } from "./config/database.js"; // Adjust path as necessary



const app = express();
```

```
// Connect to the database
connectDb();


app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

2. Environment Variables Setup:
    - Make sure you have a .env file at the root of your project containing:

```
bash
DB=mongodb://localhost:27017/mydatabase
```

3. Install Dependencies:

```
npm install mongoose dotenv
```

4. Running the Application:

```
node index.js
```

- If the database connection is successful, you should see:

```
arduino
Database Connected
Server is running on port 3000
```

Here's a detailed guide for completing each section of your project documentation based on the provided instructions:

4. Setup Instructions

Prerequisites

Before running the application, ensure the following dependencies are installed:

- Node.js: v14.x or later

- npm: v6.x or later (comes with Node.js)
- MongoDB: Community or Atlas version
- Git: For cloning the repository

Installation Steps

1. Clone the Repository:

```
git clone <repository-url>

cd <repository-name>
```

2. Install Dependencies: For the server:

```
cd server

npm install
```

For the client:

```
cd client

npm install
```

3. Set Up Environment Variables:
    - Create a .env file in the server directory.
    - Add the following variables:

```
makefile

PORT=5000

DB=<MongoDB connection string>

JWT_SECRET=<your_jwt_secret>
```

- For the client, add any required .env configurations if needed.

5. Folder Structure

Client (React Frontend)

graphql

```
client/
|
├── public/          # Static assets
├── src/
|   ├── components/   # Reusable components
|   ├── pages/        # Page-level components
|   ├── context/      # Context API for state management
|   ├── utils/        # Helper functions and utilities
|   └── App.js        # Main App component
└── package.json      # Frontend dependencies and scripts
```

Server (Node.js Backend)

```
server/
|
├── config/          # Configuration files (e.g., database, environment)
├── controllers/      # Business logic for routes
├── middlewares/      # Middleware functions (e.g., error handling, authentication)
├── models/          # Mongoose schemas and models
├── routes/          # API route definitions
├── utils/           # Utility functions (e.g., file handling)
└── server.js        # Entry point for the backend
└── package.json      # Backend dependencies and scripts
```

6. Running the Application

Start the Frontend

1. Navigate to the client directory:

```
cd client
```

2. Start the development server:

```
npm start
```

Start the Backend

1. Navigate to the server directory:

```
cd server
```

2. Start the server:

```
npm start
```

7. API Documentation

Endpoint	Method	Parameters	Description
/api/courses	POST	title, description, category, etc.	Create a new course
/api/courses/:id	DELETE	id	Delete a course
/api/lectures/:id	POST	id, title, description	Add a lecture to a course
/api/users/:id/role	PATCH	id	Update a user's role

Include example requests and responses using tools like Postman or Swagger.

8. Authentication

- Authentication Method: JSON Web Tokens (JWT)
    - Upon successful login, the server generates a token and sends it to the client.
    - The token is stored in the browser's localStorage or cookies.
- Authorization:

    - Middleware checks if the user has a valid token before granting access to protected routes.
    - Role-based access (e.g., admin, superadmin) for endpoints like role updates.

9. User Interface

- Key Features:
    - Responsive design
    - Login and registration forms
    - Course management (Create, Update, Delete)
    - Lecture addition and playback
- Include screenshots of key screens, such as the dashboard, course creation form, and lecture list.

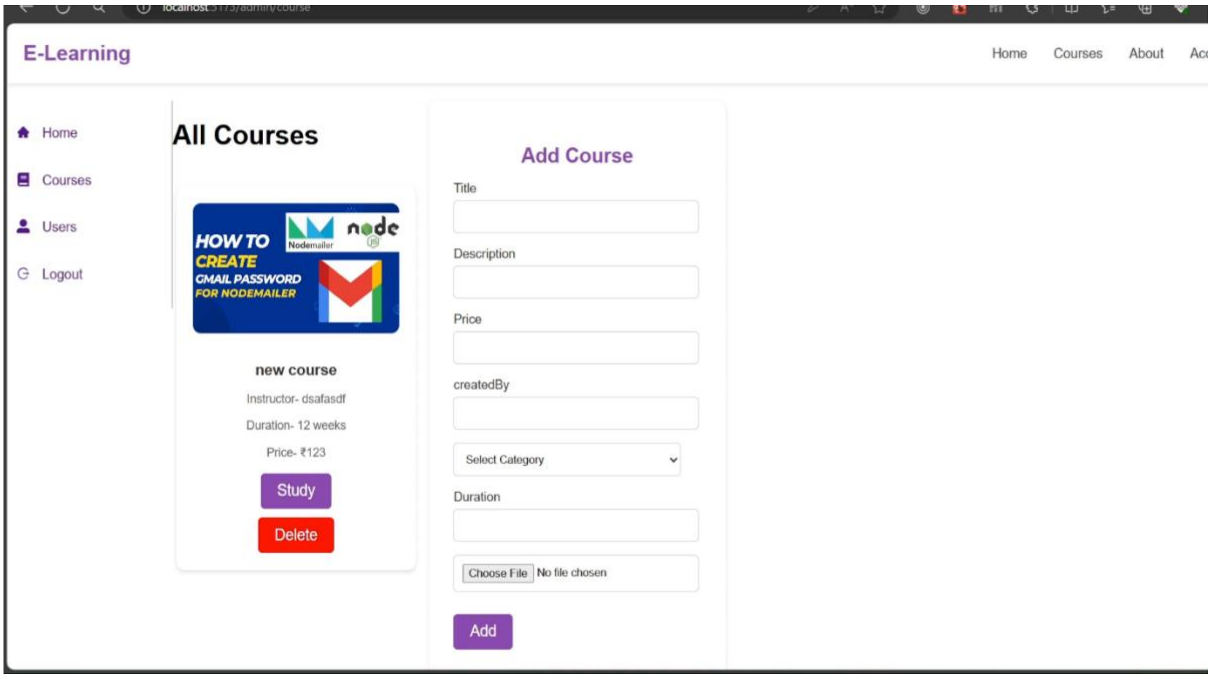
10. Testing

- Testing Strategy:
    - Unit tests for individual components and functions.
    - Integration tests for API endpoints.
- Tools:
    - Jest: For testing JavaScript code.
    - Postman/Newman: For testing API endpoints.

Example test script:

```
npm test
```

11. Screenshots or Demo



E-Learning

All Courses

new course

Instructor- dsafasdf

Duration- 12 weeks

Price- ₹123

Add Course

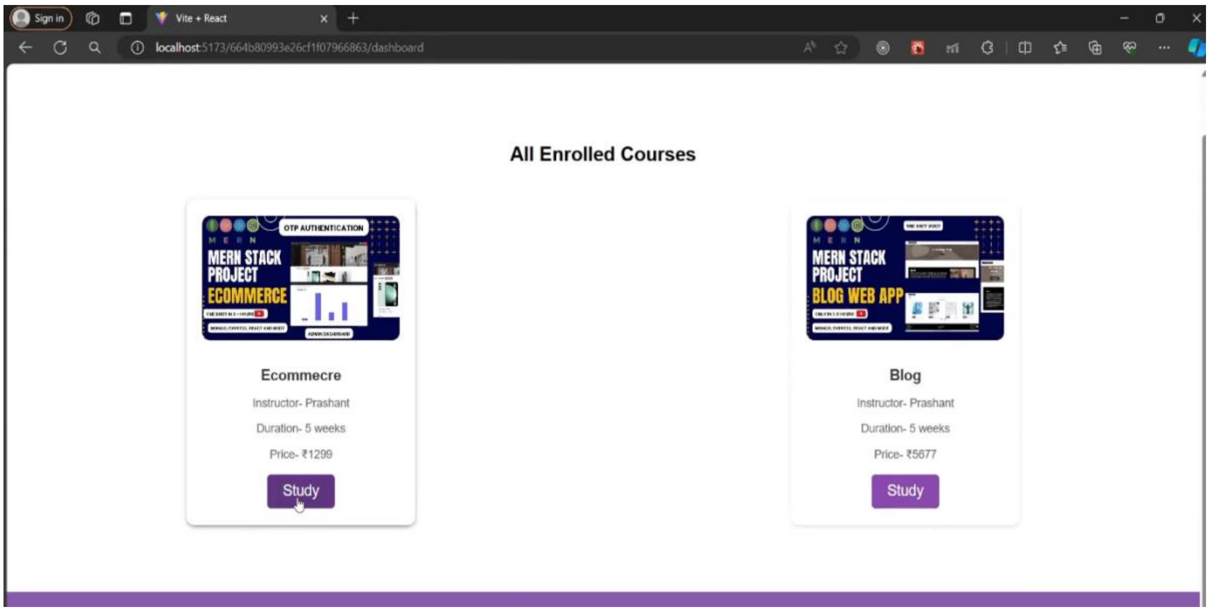
Title

Description

Price

createdBy

Duration



All Enrolled Courses

Ecommecre

Instructor- Prashant

Duration- 5 weeks

Price- ₹1299

Blog

Instructor- Prashant

Duration- 5 weeks

Price- ₹5677

12. Known Issues

- File upload fails when the file size exceeds X MB (add specific details).
- Authentication tokens expire without notification, requiring a manual login.

13. Future Enhancements

- Features:
    - Implement a search functionality for courses and lectures.
    - Add payment integration for course purchases.
    - Enable multi-language support.
- Improvements:
    - Optimize database queries for performance.
    - Enhance the UI with animations and transitions.