Project Name:Online Learning Platform

Team ID : LTVIP2025TMID25255

Team Size : 5

Team Leader : C Vidyasree

Team member : G Abhinaya

Team member: K Reddy Sai

Team member: R Varalakshmi

Team member: S Chandhra Sekhar

Online Learning Platform

With the rise of digital education, there is an increased demand for an accessible and effective online learning environment. Many existing platforms are either prohibitively expensive, inflexible, or do not provide seamless course management for both students and instructors.

- Instructors need a system that allows them to easily build and administer courses, post lessons, and create quizzes to test student learning.

- Students want a structured platform via which they may enroll in classes, track their progress, take tests, and receive feedback.

- Security is critical to ensuring adequate access control, as only authorized users can create, manage, and enroll in courses.

This project seeks to deliver a scalable and user-friendly solution for online learning while retaining security and efficiency.
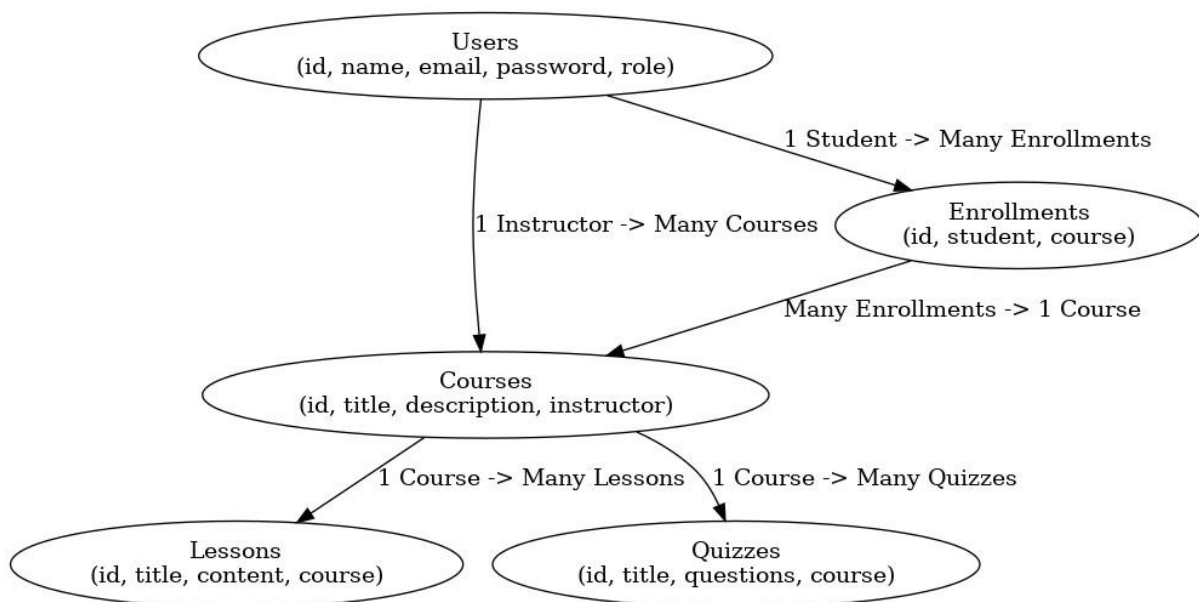
User based Use case scenario-

User: Alex (Aspiring Web Developer, Student). Alex is keen to enhance his web development skills and discovers an Online Learning Platform that provides structured classes on a variety of programming topics.

1. Account Creation and Authentication-

- Alex registers with the site and logs in as a student.

- The system authenticates him with JWT tokens and allows him access to student features.

2. Browse and enrol in a course-

- Alex finds a course called "Full Stack Web Development" and decides to enroll.

- When he enrols, the system determines whether he is already enrolled and securely keeps his enrolment record in the database.

3. Accessing Lessons and Quizzes-

- After enrolling, Alex can see course materials and study at his own pace.

- He also takes quizzes designed by the instructor to assess his knowledge.

4. Role-Based Access Control in Action-

- Alex attempts to establish a new course, but the system denies him access because course creation is only available to teachers.

- Meanwhile, Sarah, his instructor, can add new courses, tests, and change course content

5.   Secure and Structured Data Management-

- The platform uses MongoDB to securely store all user data, courses, enrolments, and progress.

- Alex's login token allows him to safely access his enrolled courses and lessons without disclosing critical information.

ER -Diagram

Explanation of Relationships-

Users and Enrolment-

- Students can enrol in several courses, and each course can have multiple students.

- An teacher can develop several courses, but each course belongs to a certain instructor.

Courses and Lessons-

- Each course can include many lessons, but each lesson belongs to a certain course.

Courses and Quizzes-

- Each course can include many quizzes, but each quiz belongs to a certain course.

Tech Stack- This project was created with the latest web technologies to provide, security, and efficiency.

Backend server is built using- ⬜

Node.js- A JavaScript runtime.

- **Express.js-** A web framework for processing API calls and routing.
- **MongoDB-** NoSQL database for storing users, courses, scalability lessons, and enrollments.
- **Mongoose-** An Object Data Modeling (ODM) library for interacting with MongoDB.

Authentication and Security-

- **JWT (JSON Web Token)** is used for user authentication and authorization.
- **bcrypt.js** is a hashing library for safe password storage.
- **dotenv** is a tool for securely managing environment variables.

Online Learning Platform

- o ER-Diagram o

  PRE REQUISITES

- o Roles and

  Responsibility o   Project

  Flow

PRE REQUISITES

To develop a Backend Platform using Node.js, Express.js and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application

Node.js and nm.: Install Node.js, which includes nm. (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server side.

• Download: https://nodejs.org/en/download/

• Installation :https://nodejs.org/en/download/package-manager/

MongoDB: Set up a MongoDB database to store hotel and booking information. Install MongoDB locally or use a cloud-based MongoDB service. • Download:

https://www.mongodb.com/try/download/community • Installation instructions :https://docs.mongodb.com/manual/installation/

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing, middleware, and API development. • Installation: open your command prompt or terminal and run the following   command: npm install express

Visual Studio Code : Download and install Visual Studio code

Postman or ThunderClient:

Download and install POSTMAN to check the output

   o   Use this link to download POSTMAN

You can also use Thunder Client extension in VS code

Roles and Responsibility

Roles & Responsibilities- This online learning platform is focused on:

- **Students-**

1. Browse the offered courses.

2. Enrol in courses.

3. Access the course content (videos, text, quizzes).

4. Track their development.

5. Take quizzes and view the results.

6. Instructors

- **Instructors-**

1. Create and manage courses.

2. Create lessons with videos and quizzes to check student knowledge.
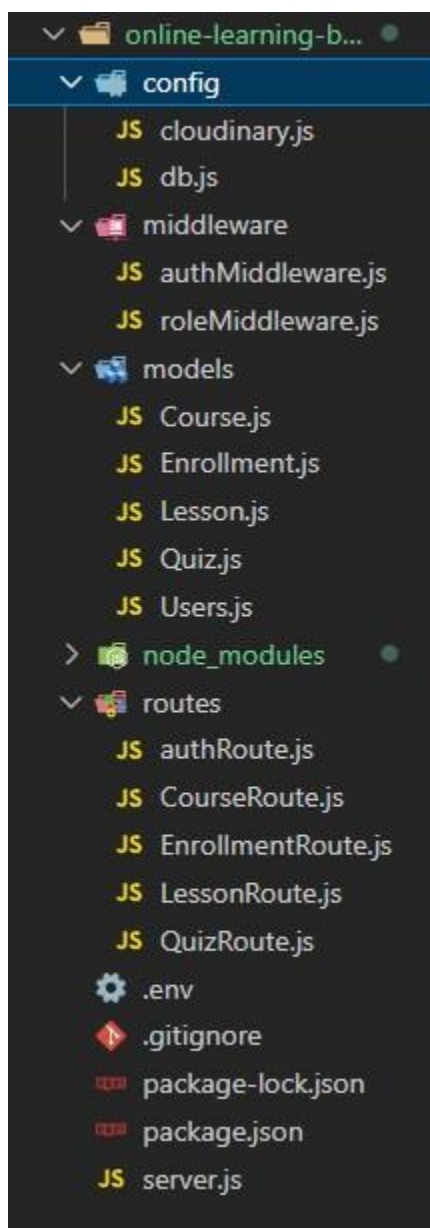
3. Manage enrolled pupils.

- **Admin-**

1. Oversee the platform

2. Manage users (students and instructors).

3. Maintain content quality and platform security.

4. This system offers a structured yet adaptable learning environment, guaranteeing that both students and teachers may effectively use the platform.

This system provides a controlled yet flexible learning environment, allowing students and instructors to interact with the platform more effectively.
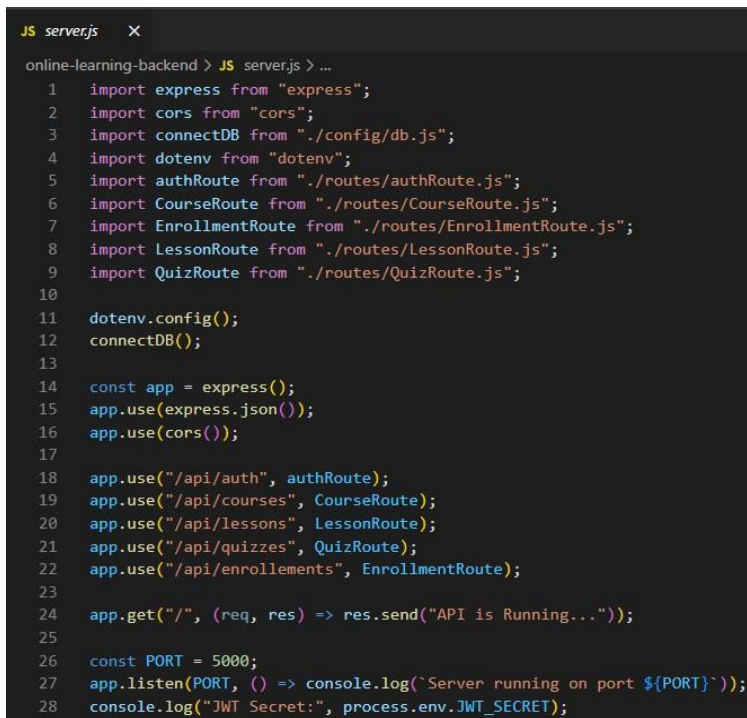
Project Setup and Configuration

File Structure- The project uses an organized directory layout to provide clean code organization and scalability.

```
∨ 📁 online-learning-b...  ●
  ∨ 📁 config
      JS cloudinary.js
      JS db.js
  ∨ 📁 middleware
      JS authMiddleware.js
      JS roleMiddleware.js
  ∨ 📁 models
      JS Course.js
      JS Enrollment.js
      JS Lesson.js
      JS Quiz.js
      JS Users.js
  > 📁 node_modules      ●
  ∨ 📁 routes
      JS authRoute.js
      JS CourseRoute.js
      JS EnrollmentRoute.js
      JS LessonRoute.js
      JS QuizRoute.js
    ⚙ .env
    ◆ .gitignore
    🔲 package-lock.json
    🔲 package.json
    JS server.js
```

The **config** folder contains two configuration files, one is **db.js** and other is **cloudinary.js**. Both of these files are essential for configuration and setting up connection with our database and cloudinary account. Then we have a folder named middleware, this folder contains files, which handle **authentication** and **authorization** logic we have implemented in the project. The **models** folder contains files which define the database **schemas** for users, courses, lessons, quizzes, and enrollments. The **routes** folder has files which handle **API** endpoints for authentication, courses, lessons, quizzes, and enrollments. In the **root** folder of the project we have a **.env** file which contains all the environment variables **package-lock.json** and **package.json** which are created by default and contain all the details about the dependencies of the project, and lastly our **server.js** file which is the starting point of our backend server.

We start with the server.js file, the **server.js** file is the backend application's entry point. It initializes the **Express server**, connects to the **MongoDB** database, and configures **middleware** and **routing**.

```js
import express from "express";
import cors from "cors";
import connectDB from "./config/db.js";
import dotenv from "dotenv";
import authRoute from "./routes/authRoute.js";
import CourseRoute from "./routes/CourseRoute.js";
import EnrollmentRoute from "./routes/EnrollmentRoute.js";
import LessonRoute from "./routes/LessonRoute.js";
import QuizRoute from "./routes/QuizRoute.js";

dotenv.config();
connectDB();

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

app.use("/api/auth", authRoute);
app.use("/api/courses", CourseRoute);
app.use("/api/lessons", LessonRoute);
app.use("/api/quizzes", QuizRoute);
app.use("/api/enrollements", EnrollmentRoute);

app.get("/", (req, res) => res.send("API is Running..."));

const PORT = 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
console.log("JWT Secret:", process.env.JWT_SECRET);
```

Imports & Setup-

- **express-** creates the backend server.

- **cors-** Allows cross-origin requests.

- **dotenv-** Loads environment variables from **.env**.

- **connectDB()-** connects to MongoDB.

Middleware-

- **express.json()-** parses JSON request bodies.

- **cors()-** Enables front-end apps to communicate with the backend.

API Routes-

- **/api/auth-** Handles user authentication.

- **/api/courses-** Manages course CRUD operations.

- **/api/lessons-** Manages lesson CRUD operations.

- **/api/quizzes-** Manages quizzes.

- **/api/enrollments-** Handles student enrollments.

Server Initialization-

- The application listens on the specified **PORT (5000 by default)**.

- Logs the current server status.

This file serves as the central hub, linking all routes and configuring the fundamental middleware functionality.

Backend Development

Now lets make our API endpoints. In our routes folder we have multiple files containing all the different logics for all the routes. Lets look at authRoute.js first.

So what is happening here, we are first importing the important dependencies:

```
JS authRoute.js U  X

online-learning-backend > routes > JS authRoute.js > ⬡ router.post("/register") callback > [∅] newUser
  1    import express from "express";
  2    import bcrypt from "bcryptjs";
  3    import jwt from "jsonwebtoken";
  4    import User from "../models/Users.js";
  5    import { verifyToken } from "../middleware/authMiddleware.js";
  6    import roleMiddleware from "../middleware/roleMiddleware.js";
  7
```

- **express-** Used to create an instance of a router (express.Router()) to define API routes.

- **bcryptjs-** Used to hash and compare passwords for security.

- **jsonwebtoken-** Used to create and verify authentication tokens (JWT).

- **User-** The User model imported from MongoDB schema (Users.js).

- **verifyToken-** Middleware that ensures only authenticated users access certain routes.

- **roleMiddleware-** Middleware that restricts access based on user roles (e.g., student, instructor, admin).

```javascript
const router = express.Router();
router.post("/register", async (req, res) => {
  try {
    const { name, email, password, role = "student" } = req.body;
    if (!["student", "instructor"].includes(role)) {
      return res.status(400).json({ message: "Invalid role. Allowed roles: student, instructor" });
    }

    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).json({ message: "Email already exists" });
    }

    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({
      name,
      email,
      password: hashedPassword,
      role,
    });

    await newUser.save();
    res.status(201).json({ message: "User registered successfully", role: newUser.role });
  } catch (error) {
    res.status(500).json({ error: "Server error, please try again" });
  }
});
```

Now the first, we have created the Register Route, we defined a **POST** route at **"/register"** for registration of a new user, it is an **async** method because it interacts with the MongoDB using **await**. Now it extracts **name**, **email**, **password** and the **role** from the request body (by default the role is set to **"student"** unless it is explicitly set to **"instructor"**). Now we check if the role entered is valid or not, only two roles are allowed **"student"** or

"instructor". After that we check that if the user is already an existing user or not by running a **findOne** query using the email entered by the user, if the email is already present in the database we return a message **"Email already exists"**, if not we hash the password with the help of **bcrypt.js** and create a new user based on the **User** schema model and await for the DB to save the user details, and we return a confirmation message that the new user has been registered successfully. We catch any errors that may happen during the registering a new user in the catch block and log them.

After the registration route we create another POST route for logging at "/login". We extract the email and password from the request body, run a findOne query based on the email entered by the user to check if the user is an existing user, and we match the password of the existing user with the hash of the password user entered, if the email and the password hashes fail to match, we return a message telling the user that the credentials entered by the user are invalid and failed to log in.

If the credentials do match, we initialise a JWT(JSON Web Token) containing the user id and user role, which is valid for 1day (after that the token expires), and the user successfully logs in. We catch any errors that may happen during the login process in the catch block and log

```
36   router.post("/login", async (req, res) => {
37     try {
38       const { email, password } = req.body;
39
40       const user = await User.findOne({ email });
41       if (!user || !(await bcrypt.compare(password, user.password))) {
42         return res.status(400).json({ message: "Invalid credentials" });
43       }
44
45       const token = jwt.sign(
46         { id: user._id, role: user.role },
47         process.env.JWT_SECRET,
48         { expiresIn: "1d" }
49       );
50
51       res.json({ message: "Login Successful", token, role: user.role });
52     } catch (error) {
53       res.status(500).json({ error: "Server error, please try again" });
54     }
55   });
```
them.

Now, if the user wishes to check the details of their profile, we create a GET route at "/me", which returns the details about the user profile. Now for this route we use a middleware "verifyToken" file which we discuss later, for security purposes.

We find the user based on the id, and hide the password here from the response. If the user doesnt exists, we return status 404, use not found, if the use is found we send back the user details. We catch the errors that may happen during this process in the Catch block.

```
router.get("/me", verifyToken, async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select("-password");
    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }
    res.json(user);
  } catch (error) {
    res.status(500).json({ error: "Server error, please try again" });
  }
});
```

At last we have a PUT route at "/update-role/:id" for a specific id. It is a protected route and requires not only the token to be verified but also the role to be set as "admin", which means that only admins can access this route and update the role of a user.

We extract the new role from the request body, ensure that the role that is to be updated is a valid one, run a query to find the user and update the document with findByIdAndUpdate and update the user role in the database. Return error with status 404 if the user doesn't exists. And at last we catch any errors that might happen during this process in the Catch block.

```
69  router.put(
70    "/update-role/:id",
71    verifyToken,
72    roleMiddleware(["admin"]),
73    async (req, res) => {
74      try {
75        const { role } = req.body;
76
77        if (!["student", "instructor", "admin"].includes(role)) {
78          return res.status(400).json({ message: "Invalid role" });
79        }
80
81        const updatedUser = await User.findByIdAndUpdate(
82          req.params.id,
83          { role },
84          { new: true }
85        );
86
87        if (!updatedUser) {
88          return res.status(404).json({ message: "User not found" });
89        }
90
91        res.json({ message: "User role updated successfully", updatedUser });
92      } catch (error) {
93        res.status(500).json({ error: "Server error, please try again" });
94      }
95    }
96  );
97
98  export default router;
```

Now lets look at the CourseRoute.js API.

Firstly we import the essential dependencies and initialize3 the router.

```
JS CourseRoute.js U, M  ✕

online-learning-backend > routes > JS CourseRoute.js > 🎁 router.post("/") callback
    1    import express from "express";
    2    import Course from "../models/Course.js";
    3    import { verifyToken } from "../middleware/authMiddleware.js";
    4    import roleMiddleware from "../middleware/roleMiddleware.js";
    5
    6    const router = express.Router();
    7
```

- **express:** Importing the Express framework to create routes.

- **Course**: Importing the Course model to interact with the courses collection in MongoDB.

- **verifyToken:** Middleware to ensure only authenticated users can access certain routes.

- **roleMiddleware:** Middleware to restrict access based on user roles (e.g., only instructors can create courses).

Our first route is to create a new course, so we create a POST route at "/create", which is only accessible for the instructors, which means that only those users who have their roles set as "instructor" can create a new course.

We create a new "Course" object with the request body and add req.user.id as the instructor (extracted from the JWT token). Then we save the new course to the MongoDB, and return with status code of 201. If there are any errors, we catch them in the Catch block.

```
 8    router.post("/", verifyToken, roleMiddleware(["instructor"]), async (req, res) => {
 9        try {
10            const course = new Course({ ...req.body, instructor: req.user.id });
11            await course.save();
12            res.status(201).json(course);
13        } catch (error) {
14            res.status(500).json({ error: error.message });
15        }
16    });
```

Next route in the line is to GET the list of all available courses at "/".

Finds all courses in the database. We replace instructor ID with the instructor's name and email with **.populate("instructor", "name email")**, and then return the result in JSON format. If any errors occur during this phase

```
18    router.get("/", async (req, res) => {
19        try {
20            const courses = await Course.find().populate("instructor", "name email");
21            res.json(courses);
22        } catch (error) {
23            res.status(500).json({ error: error.message });
24        }
25    });
```

Now next route is to GET the single Course based on the course id,

```
27    router.get("/:id", async (req, res) => {
28        try {
29            const course = await Course.findById(req.params.id).populate("instructor", "name email");
30            if (!course) return res.status(404).json({ message: "Course not Found" });
31            res.json(course);
32        } catch (error) {
33            res.status(500).json({ error: error.message });
34        }
35    });
```

It works similarly as the GET for the list of all courses, the query here is findById instead of find. If there are no courses present for the requested course id then we simply return status 404 and a message that says Course not Found, and if we find do get a course we return the result as as JSON response. Any errors that may occur during this process are caught in the catch block with status code 500.

Now the next route is to update a course, which is also another protected route, only a user with role set as "instructor" can make changes in an existing Course and update it.

We find a specific course based on its id, and update it with new details entered by the user. We check if the course with that specific id exists or not, if it doesnt we simply return status code 404 and a message Course not found, if it does we update it and return the result. Any errors that may happen during this process are captured in the catch block and are logged with status code 500.

```
37    router.put("/:id", verifyToken, roleMiddleware(["instructor"]), async (req, res) => {
38        try {
39            const course = await Course.findByIdAndUpdate(req.params.id, req.body, { new: true });
40            if (!course) return res.status(404).json({ message: "Course not found" });
41            res.json(course);
42        } catch (error) {
43            res.status(500).json({ error: error.message });
44        }
45    });
46
```

Lastly we have the DELETE route, to delete a specific course based on the course id, it is also a protected route and can only be accessed by the user who has the role of "instructor".

```
47    router.delete("/:id", verifyToken, roleMiddleware(["instructor"]), async (req, res) => {
48        try {
49            const course = await Course.findByIdAndDelete(req.params.id);
50            if (!course) return res.status(404).json({ message: "Course not found" });
51            res.json({ message: "Course deleted successfully" });
52        } catch (error) {
53            res.status(500).json({ error: error.message });
54        }
55    });
56
57    export default router;
```

We simply run the query findByIdAndDelete which takes the id present in the request, if there is a course present with the given id, it is deleted from the database and a confirmation message is sent as a result, if there is no course present in the database with the given id, we simply return with a status code of 404 and a message. Any errors happened during this process are captured in the Catch block. And at the last we export the router as default.

Now lets look at the EnrollmentRoute.js,

First things first, we
import

```
JS EnrollmentRoute.js U ✕

online-learning-backend > routes > JS EnrollmentRoute.js > 🔷 router.post("/") callback
    1    import express from "express";
    2    import Enrollment from "../models/Enrollment.js";
    3    import { verifyToken } from "../middleware/authMiddleware.js";
    4    import roleMiddleware from "../middleware/roleMiddleware.js";
    5
    6    const router = express.Router();
    7
```

- **express-** Importing Express to create API routes.

- **Enrollment-** Importing the Enrollment model from models/Enrollment.js.

- **verifyToken-** Middleware to ensure only authenticated users can access protected routes.

- **roleMiddleware-** Middleware to restrict access based on user roles (only student can enroll in courses).

And create an instance of the express.Router() to define API routes.

Our first route is to enroll in a course, for this we create a POST route at "/", and only those users who have their roles set to "student" can enroll in a course.

We extract the course that the student wants to enroll in from the request body then we check if the student is already enrolled in the course by querying Enrollement based on parameters student is the logged in user, and the course they are trying to enroll. If they are enrolled already in that course, we simply return a message that You are already enrolled in this course with a status code of 404, if not we simply make a new object of Enrollment based on the student id and the course they enrolled in, and simply return with status code 201 and the result and save it in the database. Any errors caught during this process are captured in the

```
8    router.post("/", verifyToken, roleMiddleware(["student"]), async (req, res) => {
9      try {
10       const { course } = req.body;
11
12       const existingEnrollment = await Enrollment.findOne({
13         student: req.user.id,
14         course,
15       });
16       if (existingEnrollment) {
17         return res
18           .status(400)
19           .json({ message: "You are already enrolled in this course" });
20       }
21
22       const enrollment = new Enrollment({ student: req.user.id, course });
23       await enrollment.save();
24       res.status(201).json(enrollment);
25     } catch (error) {
26       res.status(500).json({ error: error.message });
27     }
28   });
```

Catch block.

The next route is to return the list of all the enrollments a student is enrolled in. It is a GET route and only users with roles of "student" can access it.

```
30   router.get("/", verifyToken, roleMiddleware(["student"]), async (req, res) => {
31     try {
32       const enrollments = await Enrollment.find({
33         student: req.user.id,
34       }).populate("course", "title description");
35
36       if (!enrollments.length) {
37         return res.status(404).json({ message: "No enrollments found" });
38       }
39
40       res.json(enrollments);
41     } catch (error) {
42       res.status(500).json({ error: error.message });
43     }
44   });
45
46   export default router;
47
```

We simply run a find query to return the list of all the enrolled courses based on the student it as present in the request body. If there are no courses present in the enrollments, we simply return a code 404 with message No enrollments found. If there are enrollments we simply send them back as a JSON response. The errors are caught in the catch block. And lastly we export the router as default.

Our next API is for Lessons, LessonRoutes.js

```
JS LessonRoute.js U, M  ✕

online-learning-backend > routes > JS LessonRoute.js > ...
    1    import express from "express";
    2    import Lesson from "../models/Lesson.js";
    3    import mongoose from "mongoose";
    4    import { verifyToken } from "../middleware/authMiddleware.js";
    5    import roleMiddleware from "../middleware/roleMiddleware.js";
    6
    7    const router = express.Router();
```

Starting with the imports

- **express-** Importing Express to create API routes.

- **Lesson-** Importing the Lesson model from models/Lesson.js.

- **mongoose-** Importing Mongoose to validate courseId format.

- **verifyToken-** Middleware to ensure only authenticated users can access protected routes.

- **roleMiddleware-** Middleware to restrict access based on user roles (only instructor can create, update, or delete lessons).

And creating an instance of the express.Router().

Now the first route is to GET the list of all lessons available.

We simply retrieve all the lessons present in the database, if there are no lessons present in the database, just return a message No lessons found with code 404. As usual errors caught in

```
 9    router.get("/", async (req, res) => {
10      try {
11        console.log("Fetching lessons...");
12
13        const lessons = await Lesson.find();
14        console.log("Lessons found:", lessons);
15
16        if (!lessons.length) {
17          return res.status(404).json({ message: "No lessons found" });
18        }
19
20        res.json(lessons);
21      } catch (error) {
22        console.error("Error fetching lessons:", error);
23        res.status(500).json({ error: "Server error, please try again" });
24      }
25    });
```

the Catch block.

Now to create a new lesson, we create a POST route, this is an protected route and only instructors can create a new lesson.

We create a new Lesson object based on the request body and save in the database and simply return with the response and status code of 201. Errors are caught in Catch block.

```
28    router.post("/", verifyToken, roleMiddleware(["instructor"]), async (req, res) => {
29      try {
30        const lesson = new Lesson(req.body);
31        await lesson.save();
32        res.status(201).json(lesson);
33      } catch (error) {
34        res.status(500).json({ error: error.message });
35      }
36    });
```

Now to retrieve all the lessons of a specific course we create a GET route based on the specific course id.

We extract the course id from the request body, and check if the provided course id is a valid one or not, if its invalid we return with Invalid courseID message, if the id is correct we then check if the course contains any lessons or not, if there are no lessons in the course we send a message No lessons found for this course as a response, if there are lessons we simply send them back as JSON response. Any errors that may happen are caught and logged in the Catch

```
38    router.get("/:courseId", verifyToken, async (req, res) => {
39      try {
40        const { courseId } = req.params;
41
42        if (!mongoose.Types.ObjectId.isValid(courseId)) {
43          return res.status(400).json({ error: "Invalid courseId" });
44        }
45
46        const lessons = await Lesson.find({ course: courseId });
47
48        if (!lessons.length) {
49          return res.status(404).json({ message: "No lessons found for this course" });
50        }
51
52        res.json(lessons);
53      } catch (error) {
54        res.status(500).json({ error: error.message });
55      }
56    });
```

block.

Now to update an existing lesson, we create a PUT route for the specific lesson id.

Only instructors can update any existing lesson. We simply query findByIdAndUpdate for the specific lesson id and update the document. If the lesson does not exist we send back response with Lesson not found status code 404. If the lesson does exist we update the lesson and send back the resulting lesson changes. Errors are caught in Catch block.

```
58    router.put("/:id", verifyToken, roleMiddleware(["instructor"]), async (req, res) => {
59      try {
60        const lesson = await Lesson.findByIdAndUpdate(req.params.id, req.body, {
61          new: true,
62        });
63
64        if (!lesson) {
65          return res.status(404).json({ message: "Lesson not found" });
66        }
67
68        res.json(lesson);
69      } catch (error) {
70        res.status(500).json({ error: error.message });
71      }
72    });
```

Now for deleting an existing lesson by id, we create a DELETE route which takes the lesson id, and only instructors can access this route.

It is similar to that of PUT, the only difference is the query here is findByIdAndDelete.

```
74    router.delete("/:id", verifyToken, roleMiddleware(["instructor"]), async (req, res) => {
75      try {
76        const lesson = await Lesson.findByIdAndDelete(req.params.id);
77
78        if (!lesson) {
79          return res.status(404).json({ message: "Lesson not found" });
80        }
81
82        res.json({ message: "Lesson deleted" });
83      } catch (error) {
84        res.status(500).json({ error: error.message });
85      }
86    });
87
88    export default router;
89
```

Now the last route in our backend remaining is for the Quiz.

- **express**- Importing Express to create API routes.

```
JS QuizRoute.js U  X

online-learning-backend > routes > JS QuizRoute.js > 🗇 router.get("/") callback
   1    import express from "express";
   2    import Quiz from "../models/Quiz.js";
   3    import { verifyToken } from "../middleware/authMiddleware.js";
   4    import roleMiddleware from "../middleware/roleMiddleware.js";
   5
   6    const router = express.Router();
   7
```

- **Quiz**- Importing the Quiz model from models/Quiz.js.

- **verifyToken**- Middleware to ensure only authenticated users can access protected routes.

- **roleMiddleware**- Middleware to restrict access based on user roles (instructor for creating, updating, and deleting quizzes).

And creating the instance of the express.Router().

Now the very first route is to create a quiz, we need a POST route, which is only accessible by the instructors, only they can create a Quiz.

We simply, create a new Quiz object based on the request body and save it in the database. Once that is done we send back response with code 201 and the JSON of the new quiz. Any errors are caught in the Catch block.

```
8    router.post(
9      "/",
10     verifyToken,
11     roleMiddleware(["instructor"]),
12     async (req, res) => {
13       try {
14         const quiz = new Quiz(req.body);
15         await quiz.save();
16         res.status(201).json(quiz);
17       } catch (error) {
18         res.status(500).json({ error: error.message });
19       }
20     }
21   );
```

Now to retrieve all the available quizzes, we create a GET route at "/".

```
23   router.get("/", verifyToken, async (req, res) => {
24     try {
25       const quizzes = await Quiz.find().populate("course", "title");
26       res.json(quizzes);
27     } catch (error) {
28       res.status(500).json({ error: error.message });
29     }
30   });
```

We simply run a find query and replace the actual Course name and Title with the response we get back.

Now to retrieve the list of Quizzes of a specific Course, we need a GET route for specific

```
32    router.get("/:courseId", verifyToken, async (req, res) => {
33      try {
34        const quizzes = await Quiz.find({ course: req.params.courseId });
35
36        if (!quizzes.length) {
37          return res
38            .status(404)
39            .json({ message: "No quizzes found for this course" });
40        }
41
42        res.json(quizzes);
43      } catch (error) {
44        res.status(500).json({ error: error.message });
45      }
46    });
```
course id.

We extract the courseId from the request body and run the find query based on the course id. If there are no quizzes present for that course, we response back with code 404 and a message. If there are quizzes present for the course, we send back the list of them in the JSON response. Any or all errors are caught in the Catch block with code 500.

Now to update a specific quiz, we need a PUT route for the that specific quiz based on the

```
48    router.put(
49      "/:id",
50      verifyToken,
51      roleMiddleware(["instructor"]),
52      async (req, res) => {
53        try {
54          const quiz = await Quiz.findByIdAndUpdate(req.params.id, req.body, {
55            new: true,
56          });
57
58          if (!quiz) {
59            return res.status(404).json({ message: "Quiz not found" });
60          }
61
62          res.json(quiz);
63        } catch (error) {
64          res.status(500).json({ error: error.message });
65        }
66      }
67    );
```
quiz id.

This is also a protected route and can only be accessed by the instructors. As seen previously, we extract the quiz id from the request body and run a findByIdAndUpdate query based on the request body. If the quiz with that id is not present we send a response with message Quiz not found, if the quiz is present we update the quiz and send back the JSON response. The errors are caught in the catch block.

Now the last route is to DELETE the specific quiz based on its

```
69    router.delete(
70      "/:id",
71      verifyToken,
72      roleMiddleware(["instructor"]),
73      async (req, res) => {
74        try {
75          const quiz = await Quiz.findByIdAndDelete(req.params.id);
76
77          if (!quiz) {
78            return res.status(404).json({ message: "Quiz not found" });
79          }
80
81          res.json({ message: "Quiz deleted successfully" });
82        } catch (error) {
83          res.status(500).json({ error: error.message });
84        }
85      }
86    );
87
88    export default router;
```
id.

This is also a protected route and can only be accessed by the instructors. Similarly as the PUT route we extract the id from the request body and run a findByIdAndDelete query based on the id. If the quiz with the given id is not present, we response with quiz not found message, if the the quiz is present we delete the quiz from the database and respond with a confirmation message. Any errors occurred during this process are caught in the Catch block. At the last line we export the default router.

Now lets take a look at the middlewares. Our first middleware is authMiddleware.js.

```
JS authMiddleware.js U ✕

online-learning-backend > middleware > JS authMiddleware.js > ...
 1    import jwt from "jsonwebtoken";
 2    import Users from "../models/Users.js";
 3
 4    export const verifyToken = async (req, res, next) => {
 5      const token = req.headers.authorization?.split(" ")[1];
 6      if (!token) return res.status(401).json({ message: "Unauthorized" });
 7
 8      try {
 9        const decoded = jwt.verify(token, process.env.JWT_SECRET);
10        req.user = await Users.findById(decoded.id).select("-password");
11        next();
12      } catch (error) {
13        res.status(401).json({ message: "Invalid token" });
14      }
15    };
16
17    export const verifyInstructor = async (req, res, next) => {
18      await verifyToken(req, res, async () => {
19        if (req.user.role !== "instructor") return res.status(403).json({ message: "Access denied" });
20        next();
21      });
22    };
```

This file includes middleware functions that use JSON Web Tokens (JWT) to manage rolebased access control and authentication. First importing the essential modules.

- **jsonwebtoken-** Library for handling JWT authentication.

- **Users-** Importing the Users model to fetch user details from the database.

The first function (verifyToken) verifies and authenticate the user based on its JWT token. Now the function:

- Retrieves the token from the Authorization header (Bearer <token> format).

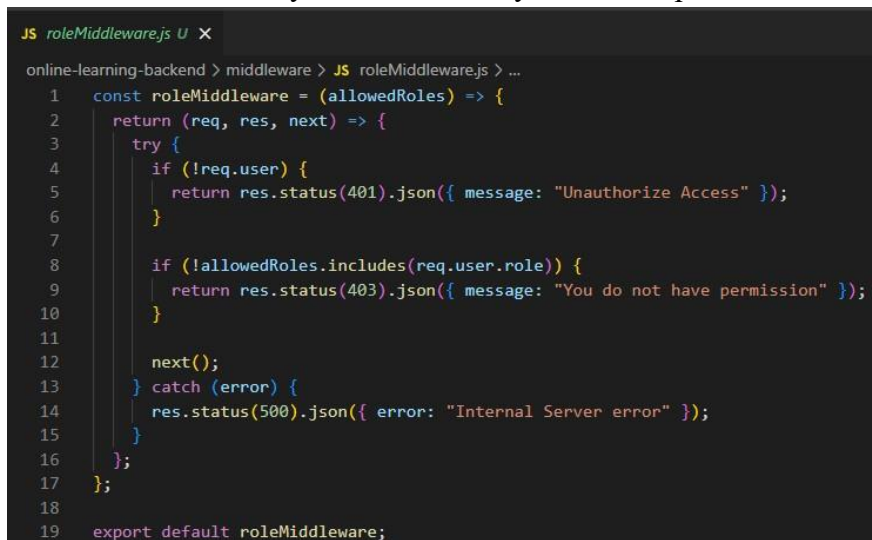- split(" ")[1] extracts only the token part.

- If no token is found, the request is unauthorized.

If there is no token, we reject the request with code 401 and message of Unauthorized. If the token is present we verify the token:

- **jwt.verify(token, process.env.JWT_SECRET)-** Decodes and verifies the token using the secret key.

- The decoded object contains user information (id, role, etc.).

Now after decoding the token, we fetch the user from the database using that decoded.id and hide the password field for security purposes. Then if the verification is successful, we pass the the request to the next middleware using next().

The other middleware file is roleMiddleware.js. A middleware function called roleMiddleware limits access to specific API routes according to user roles. It guarantees that certain actions can only be carried out by users who possess the necessary roles.

```
JS roleMiddleware.js U ✕

online-learning-backend > middleware > JS roleMiddleware.js > ...
   1    const roleMiddleware = (allowedRoles) => {
   2      return (req, res, next) => {
   3        try {
   4          if (!req.user) {
   5            return res.status(401).json({ message: "Unauthorize Access" });
   6          }
   7
   8          if (!allowedRoles.includes(req.user.role)) {
   9            return res.status(403).json({ message: "You do not have permission" });
  10          }
  11
  12          next();
  13        } catch (error) {
  14          res.status(500).json({ error: "Internal Server error" });
  15        }
  16      };
  17    };
  18
  19    export default roleMiddleware;
```
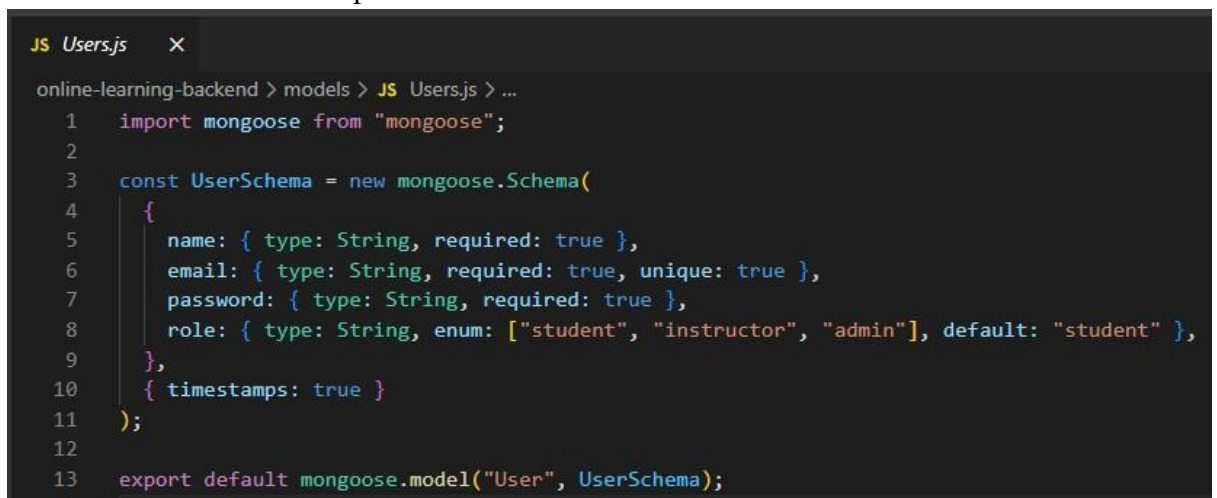
This middleware is used in routes where role-based authorization is required, such as allowing only instructors to create or modify courses, lessons, or quizzes. This function accepts an array of allowed roles. If the request is not from an authenticated user, it rejects the request and sends a message saying Unauthorized Access. Then it checks if the role of the user is one of the allowed roles, if it is not, it responds with You do not have permission, if the role is one of the allowed roles, the request proceeds with next() call. Any errors during this process are caught in the Catch block.

Database

Now lets dive into models, we create model schemas for different entities, the models specify the structure of the data saved in MongoDB. Each model represents a collection in the database and maintains data integrity. We create these model schemas using Mongoose.

User Model- The User model contains user information such as name, email address, password, and role (student/instructor).

- **name:** Saves the user's complete name.

```js
JS Users.js    ✕

online-learning-backend > models > JS Users.js > ...
    1   import mongoose from "mongoose";
    2
    3   const UserSchema = new mongoose.Schema(
    4     {
    5       name: { type: String, required: true },
    6       email: { type: String, required: true, unique: true },
    7       password: { type: String, required: true },
    8       role: { type: String, enum: ["student", "instructor", "admin"], default: "student" },
    9     },
   10     { timestamps: true }
   11   );
   12
   13   export default mongoose.model("User", UserSchema);
```

- **email:** Each user has a unique email.

- **password:** An encrypted password for authentication.

- **role:** Determines if the user is a student or an instructor.

- **timestamps:** The **createdAt** and **updatedAt** fields are added automatically.

Course Model- The Course model contains information about the courses created by

```js
JS Course.js  X

online-learning-backend > models > JS Course.js > ...
  1    import mongoose from "mongoose";
  2
  3    const CourseSchema = new mongoose.Schema(
  4      {
  5        title: { type: String, required: true },
  6        description: { type: String, required: true },
  7        instructor: {
  8          type: mongoose.Schema.Types.ObjectId,
  9          ref: "User",
 10          required: true,
 11        },
 12        price: { type: Number, default: 0 },
 13        category: { type: String, required: true },
 14        thumbnail: { type: String },
 15        published: { type: Boolean, default: false },
 16      },
 17      {
 18        timestamps: true,
 19      }
 20    );
 21
 22    export default mongoose.model("Course", CourseSchema);
```

instructors.

- **title (String, required):** The course title.

- **description (String, required):** A brief summary of what the course covers.

- **instructor (ObjectId, required, references User):** Links to the instructor who created the course.

- **price (Number, default: 0):** The cost of the course (0 means free).

- **category (String, required):** The category the course belongs to (e.g., Web Development, Data Science).

- **thumbnail (String, optional):** URL of the course thumbnail image.

- **published (Boolean, default: false):** Indicates if the course is published and visible to students.

- **timestamps (Automatic field):** Tracks when the course was created/updated.

Lesson Model- The Lesson model represents distinct lessons within a course. Each lesson has a title, a video URL, and extra content for further learning opportunities.

```js
import mongoose from "mongoose";

const LessonSchema = new mongoose.Schema(
  {
    course: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "Course",
      required: true,
    },
    title: { type: String, required: true },
    videoURL: { type: String, required: true },
    content: { type: String },
  },
  {
    timestamps: true,
  }
);

export default mongoose.model("Lesson", LessonSchema);
```

- **course (ObjectId, required, references Course):** Associates the lesson with a specific course.

- **title (String, required):** The name of the lesson.

- **videoURL (String, required):** A URL link to the lesson video (e.g., hosted on Cloudinary, YouTube, or any video service).

- **content (String, optional):** Additional textual content, such as notes or descriptions.

- **timestamps (Automatic field):** Keeps track of creation and update timestamps.

Quiz Model- The Quiz model depicts a quiz for a certain course. Each quiz comprises of multiple-choice questions, and each question has:

- A required question text.

- At least 4 options.

- A correct answer that must match one of the given options.

```js
JS Quiz.js  M  ✕
online-learning-backend > models > JS Quiz.js > ...
 2
      E:\projects\BackendProject\online-learning-backend • Contains emphasized items
 3    const QuestionSchema = new mongoose.Schema({
 4      question: { type: String, required: [true, "Question is required"] },
 5      options: {
 6        type: [String],
 7        required: [true, "Options are required"],
 8        validate: [(arr) => arr.length >= 4, "At least four options are required"],
 9      },
10      correctAnswer: {
11        type: String,
12        required: [true, "Correct answer is required"],
13        validate: {
14          validator: function (value) {
15            return this.options.includes(value);
16          },
17          message: "Correct answer must be one of the provided options",
18        },
19      },
20    });
21
22    const QuizSchema = new mongoose.Schema(
23      {
24        course: {
25          type: mongoose.Schema.Types.ObjectId,
26          ref: "Course",
27          required: [true, "Course ID is required"],
28        },
29        questions: {
30          type: [QuestionSchema],
31          required: [true, "At least one question is required"],
32          validate: [(arr) => arr.length > 0, "Quiz must have at least one question"],
33        },
34      },
35      { timestamps: true }
36    );
37
38    export default mongoose.model("Quiz", QuizSchema);
```

We have two different schemas in this model, one schema is for the Questions which will be contained inside the Quiz, and the other schema is for the Quiz itself, which is the main schema here.

Question Schema (Embedded Document)

- **question (String, Required)-** The text of the quiz question.

- **options (Array of Strings, Required)-** A list of answer choices (must have at least two options).

- **correctAnswer (String, Required)-** The correct answer, which must be one of the provided options.

- **Validation Rules-**

1. Ensures there are at least **four** options.

2. Ensures the **correctAnswer** matches one of the options.

Quiz Schema (Main Document)

- **course (ObjectId, Required)-** Links the quiz to a specific course.

- **questions (Array of QuestionSchema, Required)-** A list of multiple-choice questions.

- **timestamps-** Automatically stores the quiz creation and update times.

- **Validation Rules:**

1. Ensures the quiz is **associated with a course**.

2. Ensures the quiz has **at least one question**.

Enrollment Model- Which students are enrolled in which courses is monitored by the Enrollment Model. Additionally, it keeps track of each enrollment's progress and completion status.

- **student (ObjectId,**

```js
JS Enrollment.js U ✕

online-learning-backend > models > JS Enrollment.js > ...
  1   import mongoose from "mongoose";
  2
  3   const EnrollmentSchema = new mongoose.Schema(
  4     {
  5       student: {
  6         type: mongoose.Schema.Types.ObjectId,
  7         ref: "User",
  8         required: true,
  9       },
 10       course: {
 11         type: mongoose.Schema.Types.ObjectId,
 12         ref: "Course",
 13         required: true,
 14       },
 15       progress: { type: Number, default: 0 },
 16       completed: { type: Boolean, default: false },
 17     },
 18     { timestamps: true }
 19   );
 20
 21   export default mongoose.model("Enrollment", EnrollmentSchema);
```

Required)-

1. References the User model.

2. Identifies the student who is enrolled in the course.

- **course (ObjectId, Required)-**

1. References the Course model.

2. Identifies which course the student is enrolled in.

- **progress (Number, Default: 0)-**

1. Represents how much of the course the student has completed (e.g., percentage-based progress).

2. Starts at 0% and can be updated as the student progresses.
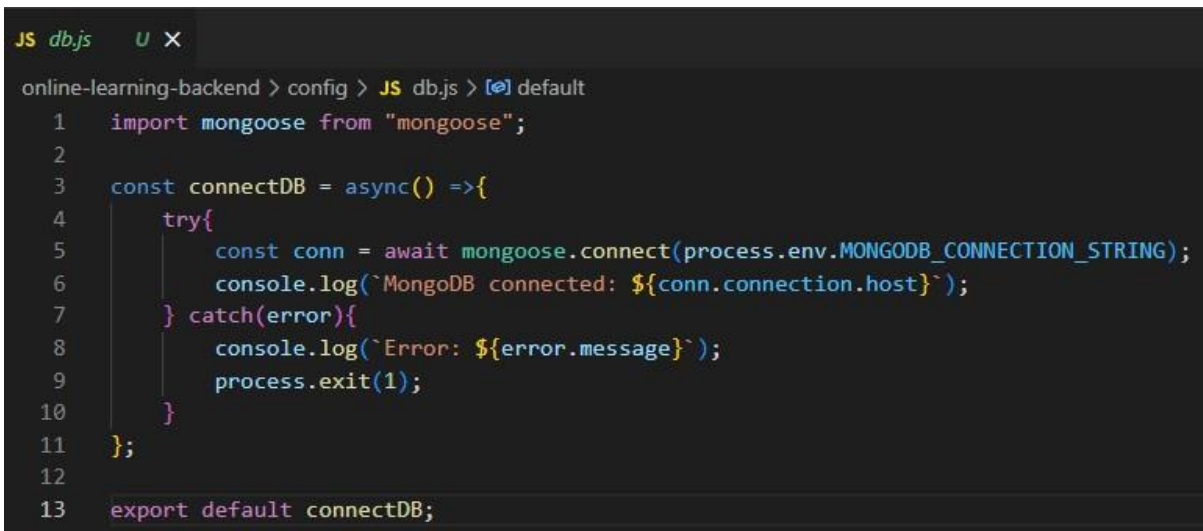
- **completed (Boolean, Default: false)-**

1. Indicates whether the student has fully completed the course.

2. Defaults to false and becomes true when the student finishes all lessons/quizzes.

- **timestamps (Auto-generated)-**

1. Stores the createdAt and updatedAt timestamps automatically.

Now our database schema models are ready, lets establish a connection between our backend ExpressJS server and our MongoDB database. For this we created a file in config folder with name 'db.js'.

The db.js file is responsible for establishing a connection between the Node.js backend server and the MongoDB database. It ensures a stable connection using Mongoose and handles potential errors during the connection process.

```
JS db.js    U ×

online-learning-backend > config > JS db.js > [∅] default
   1    import mongoose from "mongoose";
   2
   3    const connectDB = async() =>{
   4        try{
   5            const conn = await mongoose.connect(process.env.MONGODB_CONNECTION_STRING);
   6            console.log(`MongoDB connected: ${conn.connection.host}`);
   7        } catch(error){
   8            console.log(`Error: ${error.message}`);
   9            process.exit(1);
  10        }
  11    };
  12
  13    export default connectDB;
```

- Uses **Mongoose** to connect to the MongoDB database.

- Reads the **MongoDB connection string** from the environment variables.

- Logs a success message upon successful connection.

- Catches and logs errors if the connection fails, then exits the process.

- Lets look at it step by step:

1. Mongoose is imported to handle MongoDB operations efficiently.

2. Then we create a async function to handle the database connection process efficiently.

3. Inside the Try block **mongoose.connect()** establishes the connection using the MongoDB connection string from environment variables. We can get the connection string to our cluster from the MongoDB Atlas platform, it looks like this:

"mongodb+srv://<username>:<db_password>@[cluster2.aciw9.mongodb.net/?retryWrit](es=true&w=majority&appName=Cluster)es=true&w=majority&appName=Cluster0"

We just need to replace **<username>** and **<db_password>** with our username and database password.

4. On successful connection, it logs the host to which MongoDB is connected.

5. If an error occurs during connection, it logs the error message and **exits the process** with a failure code (1) in the Catch block.

6. And at the last line we export our function as default.