

## **Index.js (old version, without cloundinary)**

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

const mongoose = require("mongoose");

const cors = require("cors");

const jwt = require("jsonwebtoken");

const bcrypt = require("bcryptjs");

const axios = require("axios");

const http = require("http");

const { Server } = require("socket.io");

const { Tupath_usersModel, Employer_usersModel, Project, Admin, SubjectTagMapping } =
require("./models/Tupath_users");

const nodemailer = require("nodemailer");

const crypto = require("crypto");

const cookieParser = require('cookie-parser');

const session = require("express-session");

const MongoStore = require("connect-mongo");

//pushin purposes

require('dotenv').config()


const adminsignup = require("./routes/adminsignup");

const adminLogin = require("./routes/adminLogin");

const questions = require("./routes/questions")

const userStats = require("./routes/userStats")

const studentTags = require("./routes/studentTags");

const studentByTags = require("./routes/studentsByTag")

const users = require("./routes/users");
```

```
const adminDelete = require("./routes/adminDelete");
const corRoutes = require("./routes/corRoutes");

const checkAuth = require('./middleware/authv2')
const adminLogout = require('./routes/adminLogout')

const JWT_SECRET = "your-secret-key";
const GOOGLE_CLIENT_ID = "625352349873-
hrob3g09um6f92jscfb672fb87cn4kw.apps.googleusercontent.com";

const app = express();
const server = http.createServer(app);
const multer = require("multer");
const path = require("path");

// Middleware setup
app.use(cors({ origin: 'http://localhost:5173', credentials: true, })); // Updated CORS for
specific origin // SET CREDENTIALS AS TRUE
app.use('/uploads', express.static('uploads'));
app.use("/certificates", express.static(path.join(__dirname, "certificates")));
app.use('/cor', express.static('cor'));

app.use(express.json({ limit: '50mb' })); // Increase the limit to 50 MB
app.use(express.urlencoded({ limit: '50mb', extended: true }));
app.use(cookieParser());

app.use(session({
```

```
    secret: "your_secret_key",
    resave: false,
    saveUninitialized: false,
    store: MongoStore.create({ mongoUrl: "mongodb://127.0.0.1:27017/tupath_users" }), //
    Persistent session storage
    cookie: { maxAge: 24 * 60 * 60 * 1000 } // 1 day
  }));
```

```
//ROUTES
```

```
app.use('/', users);
app.use('/', adminsignup);
app.use('/', adminLogin);
app.use('/', questions);
app.use('/', userStats);
app.use('/', studentTags);
app.use('/', studentByTags);
app.use('/', adminDelete);
app.use('/', checkAuth);
app.use('/', adminLogout);
app.use('/', corRoutes);
```

```
// Middleware for setting COOP headers
```

```
app.use((req, res, next) => {
  res.setHeader('Cross-Origin-Opener-Policy', 'same-origin-allow-popups'); // Added COOP
  header
  res.setHeader('Cross-Origin-Resource-Policy', 'cross-origin'); // Added CORP header
  next();
});
```

```
});
```

```
// MongoDB connection
```

```
mongoose
```

```
.connect("mongodb://127.0.0.1:27017/tupath_users")
```

```
.then(() => console.log("MongoDB connected successfully"))
```

```
.catch((err) => console.error("MongoDB connection error:", err));
```

```
/*
```

```
mongoose.connect(
```

```
  "mongodb+srv://ali123:ali123@cluster0.wfrb9.mongodb.net/tupath_users?retryWrites=true&w=majority"
```

```
)
```

```
.then(() => console.log("Connected to MongoDB Atlas successfully"))
```

```
.catch((err) => console.error("MongoDB connection error:", err));
```

```
*/
```

```
// Configure multer for file uploads
```

```
const storage = multer.diskStorage({
```

```
  destination: (req, file, cb) => {
```

```
    cb(null, 'uploads/'); // Define where to store the files
```

```
  },
```

```
  filename: (req, file, cb) => {
```

```
    cb(null, Date.now() + '-' + file.originalname); // Define how files are named
```

```
  }
```

```
});
```

```
const upload = multer({  
  storage: storage,  
  limits: { fileSize: 50 * 1024 * 1024 } // 50 MB limit  
});
```

```
// JWT verification middleware
```

```
// JWT verification middleware with added debugging and error handling
```

```
const verifyToken = (req, res, next) => {  
  const authHeader = req.headers["authorization"];  
  if (!authHeader) {  
    console.error("Authorization header is missing.");  
    return res.status(401).json({ message: "Access Denied: Authorization header missing" });  
  }  
}
```

```
const token = authHeader.split(" ")[1];  
if (!token) {  
  console.error("Token not found in Authorization header.");  
  return res.status(401).json({ message: "Access Denied: Token missing" });  
}
```

```
jwt.verify(token, JWT_SECRET, (err, user) => {  
  if (err) {  
    console.error("Token verification failed:", err);  
    return res.status(403).json({ message: "Invalid Token" });  
  }  
}
```

```
}
```

```
req.user = user;
```

```
next();
```

```
});
```

```
};
```

```
// Socket.IO setup
```

```
const io = new Server(server, {
```

```
  cors: {
```

```
    origin: "http://localhost:5173",
```

```
    methods: ["GET", "POST"],
```

```
  },
```

```
});
```

```
// Chat message schema
```

```
const messageSchema = new mongoose.Schema({
```

```
  sender: {
```

```
    senderId: { type: String, required: true },
```

```
    name: { type: String, required: true },
```

```
    profileImg: { type: String, default: "" },
```

```
  },
```

```
  receiver: {
```

```
    receiverId: { type: String, required: true },
```

```
    name: { type: String, required: true },
```

```
    profileImg: { type: String, default: "" },
```

```
    },
    messageContent: {
      text: { type: String, required: true },
      attachments: [{ type: String }], // URLs or file paths
    },
    status: {
      read: { type: Boolean, default: false },
      delivered: { type: Boolean, default: false },
    },
    timestamp: { type: Date, default: Date.now },
    direction: { type: String, enum: ['sent', 'received'], required: true }, // Add direction field
  });
```

```
// Add indexes for optimization
```

```
messageSchema.index({ "sender.senderId": 1 });
messageSchema.index({ "receiver.receiverId": 1 });
messageSchema.index({ timestamp: -1 });
messageSchema.index({ "sender.senderId": 1, "receiver.receiverId": 1 });
messageSchema.index({ "receiver.receiverId": 1, "status.read": 1 });
```

```
const Message = mongoose.model("Message", messageSchema);
```

```
// Add this endpoint to fetch users
```

```
app.get('/api/userss', verifyToken, async (req, res) => {
  try {
```

```

    const students = await Tupath_usersModel.find().select('profileDetails.firstName
profileDetails.lastName profileDetails.profileImg');

    const employers = await Employer_usersModel.find().select('profileDetails.firstName
profileDetails.lastName profileDetails.profileImg');

    const users = [...students, ...employers];

    res.status(200).json({ success: true, users });

  } catch (error) {

    console.error('Error fetching users:', error);

    res.status(500).json({ success: false, message: 'Internal server error' });

  }

});

```

// REST endpoint to fetch chat messages

```

app.get("/api/messages", verifyToken, async (req, res) => {

  try {

    const userId = req.user.id; // Extract userId from the token

    const messages = await Message.find({

      $or: [

        { "sender.senderId": userId },

        { "receiver.receiverId": userId }

      ]

    }).sort({ timestamp: -1 });

    // Transform messages to add correct direction for each user

    const transformedMessages = messages.map(msg => {

      const isSender = msg.sender.senderId === userId;

      return {

```



```
...msg.toObject(),  
  direction: isSender ? 'sent' : 'received'  
};  
});
```

```
res.json(transformedMessages);  
} catch (err) {  
  console.error("Error fetching messages:", err);  
  res.status(500).json({ success: false, message: "Internal server error" });  
}  
});
```

// REST endpoint to fetch unread messages

```
app.get("/api/unread-messages", verifyToken, async (req, res) => {  
  try {  
    const userId = req.user.id; // Extract userId from the token  
  
    const messages = await Message.find({ "receiver.receiverId": userId, "status.read": false  
  }).sort({ timestamp: 1 });  
  
    res.json(messages);  
  } catch (err) {  
    console.error("Error fetching unread messages:", err);  
    res.status(500).json({ success: false, message: "Internal server error" });  
  }  
});
```

// REST endpoint to mark a message as read

```
app.put("/api/messages/:id/read", verifyToken, async (req, res) => {  
  try {  
    const messageId = req.params.id;  
    const userId = req.user.id; // Extract userId from the token  
  
    const message = await Message.findById(messageId);  
    if (!message) {  
      return res.status(404).json({ success: false, message: "Message not found" });  
    }  
  
    if (message.receiver.receiverId !== userId) {  
      return res.status(403).json({ success: false, message: "Unauthorized" });  
    }  
  
    message.status.read = true;  
    await message.save();  
  
    // Emit the message read event  
    io.emit("message_read", { messageId });  
  
    res.status(200).json({ success: true, message: "Message marked as read" });  
  } catch (err) {  
    console.error("Error marking message as read:", err);  
    res.status(500).json({ success: false, message: "Internal server error" });  
  }  
});
```

```
// Add a comment to a post

app.post("/api/posts/:id/comment", verifyToken, async (req, res) => {

  const userId = req.user.id; // Extract userId from the verified token

  const postId = req.params.id;

  const { profileImg, name, comment } = req.body;

  if (!comment || comment.trim() === "") {

    return res.status(400).json({ success: false, message: "Comment cannot be empty" });

  }

  try {

    const post = await Post.findById(postId);

    if (!post) {

      return res.status(404).json({ success: false, message: "Post not found" });

    }

    const newComment = {

      profileImg,

      username: name,

      userId, // Include userId in the comment

      comment,

      createdAt: new Date(),

    };

  }
```

```
post.comments.push(newComment);

await post.save();

io.emit("new_comment", { postId, comment: newComment });

res.status(201).json({ success: true, comment: newComment });
} catch (err) {
  console.error("Error adding comment:", err);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});

// Soft delete a comment from a post
app.delete("/api/posts/:postId/comment/:commentId", verifyToken, async (req, res) => {
  const userId = req.user.id; // Extract userId from the verified token
  const postId = req.params.postId;
  const commentId = req.params.commentId;

  try {
    // Find the post by ID, but only if it is not soft-deleted
    const post = await Post.findOne({ _id: postId, deletedAt: null });

    if (!post) {
      return res.status(404).json({ success: false, message: "Post not found or deleted" });
    }
  }
});
```

```
// Find the comment to soft delete

const comment = post.comments.find(
  (comment) => comment._id.toString() === commentId && comment.userId === userId
);

if (!comment) {
  return res.status(404).json({ success: false, message: "Comment not found or
unauthorized" });
}

// Set deletedAt timestamp instead of removing the comment
comment.deletedAt = new Date();

// Save the updated post
await post.save();

// Emit the comment deletion event
io.emit("delete_comment", { postId, commentId });

res.status(200).json({ success: true, message: "Comment soft deleted successfully" });
} catch (err) {
  console.error("Error deleting comment:", err);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});
```

```

// Edit a comment on a post

app.put("/api/posts/:postId/comment/:commentId", verifyToken, async (req, res) => {

  const userId = req.user.id; // Extract userId from the verified token

  const postId = req.params.postId;

  const commentId = req.params.commentId;

  const { comment } = req.body;

  if (!comment || comment.trim() === "") {

    return res.status(400).json({ success: false, message: "Comment cannot be empty" });

  }

  try {

    // Find the post by ID

    const post = await Post.findById(postId);

    if (!post) {

      return res.status(404).json({ success: false, message: "Post not found" });

    }

    // Find the comment to edit

    const existingComment = post.comments.find(

      (commentItem) => commentItem._id.toString() === commentId &&

      commentItem.userId === userId

    );

    if (!existingComment) {

```

```
    return res.status(404).json({ success: false, message: "Comment not found or unauthorized" });  
  }  
}
```

```
// Update the comment text
```

```
existingComment.comment = comment;
```

```
existingComment.updatedAt = new Date();
```

```
// Save the updated post
```

```
await post.save();
```

```
// Emit the comment edit event
```

```
io.emit("edit_comment", { postId, comment: existingComment });
```

```
res.status(200).json({ success: true, comment: existingComment });
```

```
} catch (err) {
```

```
  console.error("Error editing comment:", err);
```

```
  res.status(500).json({ success: false, message: "Internal server error" });
```

```
}
```

```
});
```

```
// Increment upvotes for a post
```

```
app.post("/api/posts/:id/upvote", verifyToken, async (req, res) => {
```

```
  const postId = req.params.id;
```

```
  const { id: userId, username, lastName } = req.user; // Extract user info from token
```

```
try {  
  const post = await Post.findById(postId);  
  
  if (!post) {  
    return res.status(404).json({ success: false, message: "Post not found" });  
  }  
  
  const userIndex = post.votedUsers.findIndex((user) => user.userId === userId);  
  
  if (userIndex > -1) {  
    // User already upvoted, remove upvote  
    post.votedUsers.splice(userIndex, 1);  
    post.upvotes -= 1;  
  } else {  
    // User has not upvoted, add upvote  
    post.votedUsers.push({ userId, username, lastName });  
    post.upvotes += 1;  
  }  
  
  await post.save();  
  
  res.status(200).json({ success: true, post });  
} catch (err) {  
  console.error("Error toggling upvote:", err);  
  res.status(500).json({ success: false, message: "Internal server error" });  
}
```



```
});
```

```
// Define Post schema
```

```
const postSchema = new mongoose.Schema({  
  userId: String,  
  profileImg: String,  
  name: String,  
  timestamp: { type: Date, default: Date.now },  
  content: String,  
  postImg: String,  
  upvotes: { type: Number, default: 0 },  
  deletedAt: { type: Date, default: null },  
  votedUsers: [  
    {  
      userId: String,  
      username: String,  
      lastName: String,  
    },  
  ], // Array of users who upvoted  
  comments: [  
    {  
      userId: String,  
      profileImg: String,  
      username: String,  
      comment: String,  
      createdAt: Date,
```

```

    deletedAt: { type: Date, default: null }, // Soft delete field
  },
],
});

const Post = mongoose.model("Post", postSchema);

// Get all non-deleted posts with non-deleted comments
app.get("/api/posts", async (req, res) => {
  try {
    const posts = await Post.find({ deletedAt: null }) // Exclude soft-deleted posts
      .sort({ timestamp: -1 })
      .lean(); // Convert to a plain JavaScript object for manipulation

    // Filter out soft-deleted comments
    const filteredPosts = posts.map(post => ({
      ...post,
      comments: post.comments.filter(comment => !comment.deletedAt), // Only include
non-deleted comments
    }));

    res.json(filteredPosts);
  } catch (err) {
    console.error("Error fetching posts:", err);
    res.status(500).json({ success: false, message: "Internal server error" });
  }
}

```

```
});
```

```
// Create a new post
```

```
app.post("/api/posts", verifyToken, async (req, res) => {
```

```
  const userId = req.user.id; // Extract userId from the verified token
```

```
  const userRole = req.user.role; // Extract role from the user token
```

```
  if (userRole !== "employer") {
```

```
    return res.status(403).json({ success: false, message: "Only employers can post." });
```

```
  }
```

```
  const { profileImg, name, content, postImg } = req.body;
```

```
  try {
```

```
    const newPost = new Post({
```

```
      profileImg,
```

```
      name,
```

```
      content,
```

```
      postImg,
```

```
      userId, // Save userId for the post
```

```
    });
```

```
    await newPost.save();
```

```
    res.status(201).json({ success: true, post: newPost });
```

```
    io.emit("new_post", newPost);
```

```
    } catch (err) {  
      console.error("Error creating post:", err);  
      res.status(500).json({ success: false, message: "Internal server error" });  
    }  
  });
```

// update a post

```
app.put("/api/posts/:postId", verifyToken, async (req, res) => {  
  const userId = req.user.id; // Extract userId from the verified token  
  const postId = req.params.postId;  
  const { content } = req.body;  
  
  if (!content || content.trim() === "") {  
    return res.status(400).json({ success: false, message: "Post content cannot be empty" });  
  }  
  
  try {  
    const post = await Post.findById(postId);  
  
    if (!post) {  
      return res.status(404).json({ success: false, message: "Post not found" });  
    }  
  
    if (post.userId.toString() !== userId) {  
      return res.status(403).json({ success: false, message: "Unauthorized" });  
    }  
  }  
});
```

```
}
```

```
post.content = content;
```

```
post.postImg = req.body.postImg;
```

```
post.updatedAt = new Date();
```

```
await post.save();
```

```
res.status(200).json({ success: true, post });
```

```
} catch (err) {
```

```
  console.error("Error editing post:", err);
```

```
  res.status(500).json({ success: false, message: "Internal server error" });
```

```
}
```

```
});
```

```
// delete a post with soft delete
```

```
app.delete("/api/posts/:postId", verifyToken, async (req, res) => {
```

```
  const userId = req.user.id; // Extract userId from the verified token
```

```
  const postId = req.params.postId;
```

```
  try {
```

```
    // Find the post by ID
```

```
    const post = await Post.findById(postId);
```

```
    if (!post) {
```

```
      return res.status(404).json({ success: false, message: "Post not found" });
```

```
}
```

```
// Check if the user is the one who created the post
```

```
if (post.userId.toString() !== userId) {
```

```
  return res.status(403).json({ success: false, message: "Unauthorized" });
```

```
}
```

```
// Perform a soft delete by setting the deletedAt field
```

```
post.deletedAt = new Date();
```

```
await post.save();
```

```
// Emit post deletion event (optional)
```

```
io.emit("delete_post", { postId });
```

```
// Respond with a success message
```

```
res.status(200).json({ success: true, message: "Post soft deleted successfully" });
```

```
} catch (err) {
```

```
  console.error("Error soft deleting post:", err);
```

```
  res.status(500).json({ success: false, message: "Internal server error", error: err.message  
});
```

```
}
```

```
});
```

```
// Socket.IO events for real-time chat and certificates
```

```
io.on("connection", (socket) => {
```

```
socket.on("send_message", async (data) => {  
  try {  
    const token = data.token; // Extract token from the data  
    if (!token) {  
      console.error("Token not provided");  
      return;  
    }  
  
    jwt.verify(token, JWT_SECRET, async (err, user) => {  
      if (err) {  
        console.error("Token verification failed:", err);  
        return;  
      }  
  
      const userId = user.id; // Extract userId from the token  
      console.log("Sender ID:", userId); // Log the senderId for debugging  
  
      const senderUser = await Tupath_usersModel.findById(userId) || await  
      Employer_usersModel.findById(userId);  
  
      if (!senderUser) {  
        console.error("User not found for ID:", userId); // Log the userId if user is not found  
        return;  
      }  
  
      // Create a single message with direction
```

```
const message = new Message({
  sender: {
    senderId: userId,
    name: `${senderUser.profileDetails.firstName}
${senderUser.profileDetails.lastName}`,
    profileImg: senderUser.profileDetails.profileImg,
  },
  receiver: {
    receiverId: data.receiverId,
    name: data.receiverName,
    profileImg: data.receiverProfileImg,
  },
  messageContent: {
    text: data.messageContent.text,
    attachments: data.messageContent.attachments || [],
  },
  status: {
    read: false,
    delivered: true,
  },
  timestamp: new Date(),
  direction: 'sent'
});

await message.save();

// Only emit to the specific receiver
```



```

    socket.to(data.receiverId).emit("receive_message", {
      ...message.toObject(),
      direction: 'received'
    });

    // Send confirmation back to sender
    socket.emit("message_sent", message);
  });
} catch (err) {
  console.error("Error saving message:", err);
}
});

socket.on("disconnect", () => {
  // console.log(` User disconnected: ${socket.id}`);
});
}); // Add this closing bracket

// Student Certificate Schema
const StudentCert = new mongoose.Schema({
  StudId: { type: String, required: true }, // Unique identifier for the student
  StudName: { type: String, required: true }, // Full name of the student
  timestamp: { type: Date, default: Date.now }, // Record creation timestamp
  Certificate: {
    CertName: { type: String, required: true }, // Name/title of the certificate
    CertDescription: { type: String, required: true }, // Detailed description of the certificate
  }
});

```

```

CertThumbnail: { type: String, default: "" }, // URL or path to the certificate thumbnail
Attachments: [{
  type: String,
  validate: {
    validator: function (v) {
      // Ensure each attachment has an allowed file extension
      const allowedExtensions = /\. (jpg|jpeg|png|pdf|docx|txt)$/i;
      return allowedExtensions.test(v);
    },
    message: "Attachments must be valid file URLs with extensions jpg, jpeg, png, pdf, docx, or txt.",
  },
}],
},
});

```

```

const StudentCertificate = mongoose.model("StudentCertificate", StudentCert);

```

```

// Endpoint to handle certificate uploads

```

```

app.post("/api/uploadCertificate", verifyToken, upload.fields([
  { name: "thumbnail", maxCount: 1 },
  { name: "attachments", maxCount: 10 }
]), async (req, res) => {
  try {
    const userId = req.user.id;
    const userName = req.user.name; // Ensure user name is extracted from the token

```

```

const { CertName, CertDescription } = req.body;

if (!CertName || !CertDescription) {

    return res.status(400).json({ success: false, message: "Certificate name and description
are required." });

}

const thumbnail = req.files["thumbnail"] ?
`/uploads/${req.files["thumbnail"][0].filename}` : "";

const attachments = req.files["attachments"] ? req.files["attachments"].map(file =>
`/uploads/${file.filename}`) : [];

const newCertificate = new StudentCertificate({

    StudId: userId,

    StudName: userName, // Use the extracted user name

    Certificate: {

        CertName,

        CertDescription,

        CertThumbnail: thumbnail,

        Attachments: attachments,

    },

});

await newCertificate.save();

// Emit the new certificate event
io.emit("new_certificate", newCertificate);

```

```
    res.status(201).json({ success: true, message: "Certificate uploaded successfully",
certificate: newCertificate });

    } catch (error) {

        console.error("Error uploading certificate:", error);

        res.status(500).json({ success: false, message: "Internal server error", error:
error.message });

    }

});
```

// Endpoint to fetch certificates for a user

```
app.get('/api/certificates', verifyToken, async (req, res) => {

    try {

        const userId = req.user.id;

        const certificates = await StudentCertificate.find({ StudId: userId });

        res.status(200).json({ success: true, certificates });

    } catch (error) {

        console.error("Error fetching certificates:", error);

        res.status(500).json({ success: false, message: 'Internal server error' });

    }

});
```

// Endpoint to delete a certificate

```
app.delete('/api/certificates/:id', verifyToken, async (req, res) => {

    try {

        const certificateId = req.params.id;

        const userId = req.user.id;
```

```
const certificate = await StudentCertificate.findOneAndDelete({ _id: certificateId, StudId:
userId });
```

```
if (!certificate) {
  return res.status(404).json({ success: false, message: "Certificate not found" });
}
```

```
// Emit the delete certificate event
io.emit("delete_certificate", { certificateId });
```

```
res.status(200).json({ success: true, message: "Certificate deleted successfully" });
} catch (error) {
  console.error('Error deleting certificate:', error);
  res.status(500).json({ success: false, message: 'Internal server error' });
}
});
```

```
// Endpoint to fetch profile data for a specific user including projects and certificates
```

```
app.get('/api/profile/:id', verifyToken, async (req, res) => {
  const { id } = req.params;
  const requestingUserId = req.user.id;
  const requestingUserRole = req.user.role;

  try {
    const user = await Tupath_usersModel.findById(id).populate({
```

```

    path: 'profileDetails.projects',
    select: 'projectName description tags tools thumbnail projectUrl'
  });

  if (!user) {
    return res.status(404).json({ success: false, message: 'User not found' });
  }

  let certificates = await StudentCertificate.find({ StudId: id });

  // Hide projects and certificates if the requesting user is another student
  if (requestingUserRole === 'student' && requestingUserId.toString() !== id.toString()) {
    user.profileDetails.projects = [];
    certificates = [];
  }

  res.status(200).json({
    success: true,
    profile: {
      ...user.toObject(),
      profileDetails: {
        ...user.profileDetails,
        certificates,
      },
    },
  });

```

```
    } catch (err) {  
      console.error('Error fetching profile:', err);  
      res.status(500).json({ success: false, message: 'Internal server error' });  
    }  
  });
```

// Login endpoint

```
app.post("/login", async (req, res) => {  
  const { email, password, role } = req.body;  
  
  try {  
    // Find the user by email and ensure they are not soft deleted  
    const user = role === "student"  
      ? await Tupath_usersModel.findOne({ email, deletedAt: null })  
      : await Employer_usersModel.findOne({ email, deletedAt: null });  
  
    // If no user is found or the password is incorrect  
    if (!user || !(await bcrypt.compare(password, user.password))) {  
      return res.status(400).json({ success: false, message: "Invalid email or password" });  
    }  
  
    const token = jwt.sign({ id: user._id, role }, JWT_SECRET, { expiresIn: "1h" });  
  
    let redirectPath = user.isNewUser ? "/studentprofilecreation" : "/homepage";  
    if (role === "employer") redirectPath = user.isNewUser ? "/employerprofilecreation" :  
      "/homepage";
```

```
user.isNewUser = false;
await user.save();

res.status(200).json({ success: true, token, message: "Login successful", redirectPath });
} catch (err) {
  res.status(500).json({ success: false, message: "Internal server error" });
}
});

// Google Signup endpoint
app.post("/google-signup", async (req, res) => {
  const { token, role } = req.body;

  // Validate role
  if (!['student', 'employer'].includes(role)) {
    return res.status(400).json({ success: false, message: 'Invalid role specified' });
  }

  try {
    // Verify the Google token using Google API
    const googleResponse = await
    axios.get(`https://oauth2.googleapis.com/tokeninfo?id_token=${token}`);

    if (googleResponse.data.aud !== GOOGLE_CLIENT_ID) {
      return res.status(400).json({ success: false, message: 'Invalid Google token' });
    }
  }
});
```



```
}
```

```
const { email, sub: googleId, name } = googleResponse.data;
```

```
// Select the correct model based on the role
```

```
const UserModel = role === 'student' ? Tupath_usersModel : Employer_usersModel;
```

```
// Check if the user already exists
```

```
const existingUser = await UserModel.findOne({ email });
```

```
if (existingUser) {
```

```
  return res.status(409).json({ success: false, message: 'Account already exists. Please  
log in.' });
```

```
}
```

```
// Create a new user
```

```
const newUser = await UserModel.create({
```

```
  name,
```

```
  email,
```

```
  password: googleId, // Placeholder for password
```

```
  isNewUser: true,
```

```
  googleSignup: true,
```

```
  role, // Add role explicitly
```

```
});
```

```
// Generate JWT token
```

```
const jwtToken = jwt.sign(
```

```

    { email, googleId, name, id: newUser._id, role },
    JWT_SECRET,
    { expiresIn: '1h' }
  );

  const redirectPath = role === 'student' ? '/studentprofilecreation' :
  '/employerprofilecreation';

  res.json({ success: true, token: jwtToken, redirectPath });
} catch (error) {
  console.error('Google sign-up error:', error);
  res.status(500).json({ success: false, message: 'Google sign-up failed' });
}
});

// Google login endpoint
app.post("/google-login", async (req, res) => {
  const { token, role } = req.body;

  try {
    const googleResponse = await
    axios.get(`https://oauth2.googleapis.com/tokeninfo?id_token=${token}`);

    if (googleResponse.data.aud !== GOOGLE_CLIENT_ID) {

```

```

    return res.status(400).json({ success: false, message: "Invalid Google token" });
  }

  const { email, sub: googleId, name } = googleResponse.data;
  const UserModel = role === "student" ? Tupath_usersModel : Employer_usersModel;

  // Check if the user exists and is not soft-deleted
  const user = await UserModel.findOne({ email, deletedAt: null });

  if (!user) {
    return res.status(404).json({ success: false, message: "User not registered or has been
deleted." });
  }

  // Generate JWT token
  const jwtToken = jwt.sign(
    { email, googleId, name, id: user._id, role },
    JWT_SECRET,
    { expiresIn: "1h" }
  );

  const redirectPath = "/homepage"; // Same for both roles

  res.json({ success: true, token: jwtToken, redirectPath });
} catch (error) {
  console.error("Google login error:", error);
}

```

```
    res.status(500).json({ success: false, message: "Google login failed" });
  }
});
```

```
// Student signup endpoint
```

```
app.post("/studentsignup", async (req, res) => {
  const { firstName, lastName, email, password } = req.body;

  try {
    const existingUser = await Tupath_usersModel.findOne({ email });

    if (existingUser) {
      return res.status(400).json({ success: false, message: "User already exists." });
    }
  }
```

```
  const hashedPassword = await bcrypt.hash(password, 10);
```

```
  const newUser = await Tupath_usersModel.create({
    name: `${firstName} ${lastName}`,
    email,
    password: hashedPassword,
    isNewUser: true,
    role: 'student', // Explicitly set the role
  });
```

```
  const token = jwt.sign({ id: newUser._id, role: 'student' }, JWT_SECRET, { expiresIn: '1h' });
```

```
return res.status(201).json({
  success: true,
  token,
  message: "Signup successful",
  redirectPath: "/studentprofilecreation",
});
} catch (err) {
  console.error("Error during signup:", err);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});

// Employer signup endpoint
app.post("/employersignup", async (req, res) => {
  const { firstName, lastName, email, password } = req.body;

  try {
    const existingUser = await Employer_usersModel.findOne({ email });

    if (existingUser) {
      return res.status(400).json({ success: false, message: "User already exists." });
    }

    const hashedPassword = await bcrypt.hash(password, 10);
```

```
const newUser = await Employer_usersModel.create({
  name: `${firstName} ${lastName}`,
  email,
  password: hashedPassword,
  isNewUser: true,
  role: 'employer', // Explicitly set the role
});

const token = jwt.sign({ id: newUser._id, role: 'employer' }, JWT_SECRET, { expiresIn: '1h'
});

return res.status(201).json({
  success: true,
  token,
  message: "Signup successful",
  redirectPath: "/employerprofilecreation",
});
} catch (err) {
  console.error("Error during signup:", err);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});
```

```
//-----NEWLY ADDED-----
-----
```

```
app.post('/api/updateStudentProfile', verifyToken, async (req, res) => {  
  try {  
    const userId = req.user.id;  
  
    const {  
      studentId,  
      firstName,  
      lastName,  
      middleName,  
      department,  
      yearLevel,  
      dob,  
      profileImg,  
      gender,  
      address,  
      techSkills,  
      softSkills,  
      contact } = req.body;  
  
    // email  
  
  
    const updatedUser = await Tupath_usersModel.findByIdAndUpdate(  
      userId,  
      {  
        $set: {  
          profileDetails: {  
            studentId,  
            firstName,  

```

```
    lastName,  
    middleName,  
    department,  
    yearLevel,  
    dob,  
    profileImg,  
    gender,  
    address,  
    techSkills,  
    softSkills,  
    contact  
    // email  
  }  
}  
},  
{ new: true, upsert: true }  
);
```

```
if (!updatedUser) {  
  return res.status(404).json({ success: false, message: 'User not found' });  
}
```

```
  res.status(200).json({ success: true, message: 'Profile updated successfully',  
updatedUser });  
} catch (error) {  
  res.status(500).json({ success: false, message: 'Internal server error' });
```



```
}  
});
```

```
app.post('/api/updateEmployerProfile', verifyToken, async (req, res) => {  
  try {  
    const userId = req.user.id;  
    const {  
      firstName,  
      lastName,  
      middleName,  
      dob,  
      gender,  
      nationality,  
      address,  
      profileImg,  
      companyName,  
      industry,  
      location,  
      aboutCompany,  
      contactPersonName,  
      position,  
      // email,  
      phoneNumber,  
      preferredRoles,  
      internshipOpportunities,  
      preferredSkills } = req.body;
```

```
const updatedUser = await Employer_usersModel.findByIdAndUpdate(
  userId,
  {
    $set: {
      profileDetails: {
        firstName,
        lastName,
        middleName,
        dob,
        gender,
        nationality,
        address,
        profileImg,
        companyName,
        industry,
        location,
        aboutCompany,
        contactPersonName,
        position,
        // email,
        phoneNumber,
        preferredRoles,
        internshipOpportunities,
        preferredSkills
      }
    }
  }
)
```

```

    }
  },
  { new: true, upsert: true }
);

if (!updatedUser) {
  return res.status(404).json({ success: false, message: 'User not found' });
}

res.status(200).json({ success: true, message: 'Profile updated successfully',
updatedUser });
} catch (error) {
  res.status(500).json({ success: false, message: 'Internal server error' });
}
});

// Profile fetching endpoint
app.get('/api/profile', verifyToken, async (req, res) => {
  try {
    const userId = req.user.id;
    const role = req.user.role; // Extract role from the token

    const userModel = role === 'student' ? Tupath_usersModel : Employer_usersModel;
    const user = await userModel.findById(userId)
      .select('email role profileDetails createdAt googleSignup')
      .populate({

```

```
    path: 'profileDetails.projects',
    select: 'projectName description tags tools thumbnail projectUrl',
    strictPopulate: false, // Allow flexible population
  });

if (!user) {
  return res.status(404).json({ success: false, profile: 'User not Found' });
}

// Fetch certificates for the user
const certificates = await StudentCertificate.find({ StudId: userId });

// Return profile details tailored to the role
const profile = {
  email: user.email,
  role: user.role,
  profileDetails: user.role === 'student' ? {
    ...user.profileDetails,
    certificates, // Include certificates in the profile details
  } : {
    ...user.profileDetails,
    companyName: user.profileDetails.companyName || null,
    industry: user.profileDetails.industry || null,
    aboutCompany: user.profileDetails.aboutCompany || null,
    preferredRoles: user.profileDetails.preferredRoles || [],
    internshipOpportunities: user.profileDetails.internshipOpportunities || false,
```

```

    },
    createdAt: user.createdAt,
    googleSignup: user.googleSignup,
  };

  res.status(200).json({ success: true, profile });
} catch (error) {
  console.error('Error fetching profile:', error);
  res.status(500).json({ success: false, message: 'Internal server error' });
}
});

/*app.post("/api/uploadProfileImage", verifyToken, upload.single("profileImg"), async (req,
res) => {
  try {
    const userId = req.user.id;

    // Validate if file exists
    if (!req.file) {
      return res.status(400).json({ success: false, message: "No file uploaded" });
    }

    const profileImgPath = `/uploads/${req.file.filename}`;
    console.log("Uploaded file path:", profileImgPath); // Debugging

    // Update for both student and employer models

```

```

const updatedStudent = await Tupath_usersModel.findByIdAndUpdate(
  userId,
  { $set: { "profileDetails.profileImg": profileImgPath } },
  { new: true }
);

const updatedEmployer = await Employer_usersModel.findByIdAndUpdate(
  userId,
  { $set: { "profileDetails.profileImg": profileImgPath } },
  { new: true }
);

// If no user was updated, return an error
if (!updatedStudent && !updatedEmployer) {
  console.log("User not found for ID:", userId); // Debugging
  return res.status(404).json({ success: false, message: "User not found" });
}

res.status(200).json({
  success: true,
  message: "Profile image uploaded successfully",
  profileImg: profileImgPath,
});
} catch (error) {
  console.error("Error uploading profile image:", error);
  res.status(500).json({ success: false, message: "Internal server error" });
}

```

```

    }
  });
  */

// api upload image endpoint
app.post("/api/uploadProfileImage", verifyToken, upload.single("profileImg"), async (req, res) => {
  try {
    const userId = req.user.id;

    if (!req.file) {
      return res.status(400).json({ success: false, message: "No file uploaded" });
    }

    const profileImgPath = `/uploads/${req.file.filename}`;

    const userModel = req.user.role === "student" ? Tupath_usersModel :
Employer_usersModel;

    const updatedUser = await userModel.findByIdAndUpdate(
      userId,
      { $set: { "profileDetails.profileImg": profileImgPath } },
      { new: true }
    );

    if (!updatedUser) {
      return res.status(404).json({ success: false, message: "User not found" });
    }
  }

```

```

    res.status(200).json({
      success: true,
      message: "Profile image uploaded successfully",
      profileImg: profileImgPath,
    });
  } catch (error) {
    console.error("Error uploading profile image:", error);
    res.status(500).json({ success: false, message: "Internal server error" });
  }
});

app.post("/api/uploadProject", verifyToken, upload.fields([
  { name: "thumbnail", maxCount: 1 },
  { name: "selectedFiles", maxCount: 10 },
  { name: "ratingSlip", maxCount: 1 }
]), async (req, res) => {
  try {
    console.log("Received project upload request with data:", req.body);

    // Retrieve saved subject & grade from session
    const { subject, grade, ratingSlip, year, term } = req.session.assessmentData || {};

    if (!subject || !grade) {
      console.error("Missing subject or grade in session.");
      return res.status(400).json({ success: false, message: "Missing required fields." });
    }
  }
});

```



```
}
```

```
const thumbnail = req.files?.["thumbnail"]?.[0]?.filename ?  
`/uploads/${req.files["thumbnail"][0].filename}` : null;  
  
const selectedFiles = req.files?.["selectedFiles"] ? req.files["selectedFiles"].map(file =>  
file.path) : [];  
  
const ratingSlipPath = req.files?.["ratingSlip"]?.[0]?.filename  
? `/uploads/${req.files["ratingSlip"][0].filename}`  
: ratingSlip;
```

```
// Create and save the new project
```

```
const newProject = new Project({  
  user: req.user.id, // Add this line  
  projectName: req.body.projectName,  
  description: req.body.description,  
  tag: req.body.tag,  
  tools: req.body.tools,  
  projectUrl: req.body.projectUrl,  
  roles: req.body.roles,  
  subject,  
  grade,  
  ratingSlip: ratingSlipPath,  
  status: "pending",  
  thumbnail,  
  selectedFiles,  
  year,  
  term,
```

```
});
```

```
await newProject.save();
```

```
// Find student by ID and update their projects list
```

```
const userId = req.user?.id || req.body.userId; // Get user ID from auth or request body
```

```
if (!userId) {
```

```
  console.error("User ID is missing.");
```

```
  return res.status(400).json({ success: false, message: "User ID is required." });
```

```
}
```

```
const user = await Tupath_usersModel.findOneAndUpdate(
```

```
  { _id: userId },
```

```
  { $addToSet: { "profileDetails.projects": newProject._id } }, // Ensure project is stored in  
profile
```

```
  { new: true } 
```

```
);
```

```
if (!user) {
```

```
  console.error("User not found.");
```

```
  return res.status(404).json({ success: false, message: "User not found." });
```

```
}
```

```
// Update best tag dynamically
```

```
await user.calculateBestTag();
```

```

// Clear session after successful save
delete req.session.assessmentData;

console.log("Project successfully saved and linked to user:", newProject);
res.status(201).json({ success: true, project: newProject });

} catch (error) {
  console.error("Error uploading project:", error);
  res.status(500).json({ success: false, message: "Server error" });
}
});

/* //BACKUP LATEST API

app.post("/api/uploadProject", verifyToken, upload.fields([
  { name: "thumbnail", maxCount: 1 },
  { name: "selectedFiles", maxCount: 10 },
  { name: "ratingSlip", maxCount: 1 }
]), async (req, res) => {
  try {
    console.log("Received project upload request with data:", req.body);
    console.log("Session before accessing assessmentData:", req.session);

    // Retrieve saved subject & grade from session
    const { subject, grade, ratingSlip } = req.session.assessmentData || {};

```

```
console.log("Extracted assessment data from session:", { subject, grade, ratingSlip });
```

```
if (!subject || !grade) {
```

```
  console.error("Missing subject or grade in session.");
```

```
  return res.status(400).json({ success: false, message: "Missing required fields." });
```

```
}
```

```
  const thumbnail = req.files?.["thumbnail"]?.[0]?.filename ?
```

```
  `/uploads/${req.files["thumbnail"][0].filename}` : null;
```

```
  const selectedFiles = req.files?.["selectedFiles"] ? req.files["selectedFiles"].map(file =>  
file.path) : [];
```

```
  const ratingSlipPath = req.files?.["ratingSlip"]?.[0]?.filename
```

```
  ? `/uploads/${req.files["ratingSlip"][0].filename}`
```

```
  : ratingSlip;
```

```
const newProject = new Project({
```

```
  projectName: req.body.projectName,
```

```
  description: req.body.description,
```

```
  tag: req.body.tag,
```

```
  tools: req.body.tools,
```

```
  projectUrl: req.body.projectUrl,
```

```
  roles: req.body.roles,
```

```
  subject,
```

```
  grade,
```

```
  ratingSlip: ratingSlipPath, // <-- Save ratingSlip instead of corFile
```

```
  status: "pending",
```

```
  thumbnail,
```

```

        selectedFiles
    });

    await newProject.save();

    // Clear session after successful save
    delete req.session.assessmentData;

    console.log("Project successfully saved:", newProject);

    res.status(201).json({ success: true, project: newProject });
  } catch (error) {
    console.error("Error saving project:", error);
    res.status(500).json({ success: false, message: "Server error" });
  }
});

*/

app.get("/api/projects", verifyToken, async (req, res) => {
  try {
    const userId = req.user.id;

    // Fetch user with populated projects
    const user = await
    Tupath_usersModel.findById(userId).populate("profileDetails.projects");

```

```
if (!user) {  
  return res.status(404).json({ success: false, message: "User not found" });  
}  
  
// Add scores and tag summary for each project  
const projectsWithScores = user.profileDetails.projects.map((project) => {  
  const totalScore = project.assessment.reduce((sum, question) => sum +  
(question.weightedScore || 0), 0);  
  
  return {  
    _id: project._id,  
    projectName: project.projectName,  
    description: project.description,  
    tag: project.tag,  
    totalScore, // Sum of all weighted scores for the project  
    tools: project.tools,  
    status: project.status,  
    assessment: project.assessment, // Include detailed assessment  
    createdAt: project.createdAt,  
    roles: project.roles,  
  };  
});  
  
res.status(200).json({  
  success: true,
```

```
    projects: projectsWithScores,
  });
} catch (error) {
  console.error("Error fetching projects:", error);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});
```

```
app.delete("/api/projects/:projectId", verifyToken, async (req, res) => {
  try {
    const userId = req.user.id; // Extract user ID from the token
    const { projectId } = req.params;

    // Find the user
    const user = await Tupath_usersModel.findById(userId);
    if (!user) {
      return res.status(404).json({ success: false, message: "User not found" });
    }

    // Find the project in the user's profile and remove it
    const projectIndex = user.profileDetails.projects.findIndex(
      (project) => project._id.toString() === projectId
    );
    if (projectIndex === -1) {
      return res.status(404).json({ success: false, message: "Project not found in user's profile" });
    }
  }
});
```

```

    }

    // Remove the project reference from the user's profile
    user.profileDetails.projects.splice(projectIndex, 1);
    await user.save();

    // Delete the project document from the 'projects' collection
    const deletedProject = await Project.findByIdAndDelete(projectId);
    if (!deletedProject) {
        return res.status(404).json({ success: false, message: "Project not found in projects collection" });
    }

    res.status(200).json({
        success: true,
        message: "Project deleted successfully from user's profile and projects collection",
    });
} catch (error) {
    console.error("Error deleting project:", error);
    res.status(500).json({ success: false, message: "Internal server error" });
}
});

// Endpoint for uploading certificate photos

```



```
app.post("/api/uploadCertificate", verifyToken, upload.array("certificatePhotos", 3), async (req, res) => {
```

```
  try {
```

```
    const userId = req.user.id;
```

```
    const filePaths = req.files.map(file => `/certificates/${file.filename}`);
```

```
    const updatedUser = await Tupath_usersModel.findByIdAndUpdate(
```

```
      userId,
```

```
      { $push: { "profileDetails.certificatePhotos": { $each: filePaths } } },
```

```
      { new: true } 
```

```
    );
```

```
    if (!updatedUser) {
```

```
      return res.status(404).json({ success: false, message: "User not found" });
```

```
    }
```

```
    res.status(200).json({ success: true, message: "Certificate photos uploaded successfully", certificatePhotos: filePaths });
```

```
  } catch (error) {
```

```
    console.error("Error uploading certificate photos:", error);
```

```
    res.status(500).json({ success: false, message: "Internal server error" });
```

```
  }
```

```
});
```

```
// -----api for dynamic search-----
```

```
app.get('/api/search', verifyToken, async (req, res) => {
```

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

```
if (!query) {
  return res.status(400).json({ success: false, message: 'Query parameter is required' });
}

try {
  const regex = new RegExp(query, 'i'); // Case-insensitive regex

  const loggedInUserId = req.user.id; // Assuming verifyToken populates req.user with user
  details

  // Search students only
  const studentResults = await Tupath_usersModel.find({
    $and: [
      {
        $or: [
          { 'profileDetails.firstName': regex },
          { 'profileDetails.middleName': regex },
          { 'profileDetails.lastName': regex },
          { bestTag: regex } // Search by `bestTag`
        ]
      },
      { _id: { $ne: loggedInUserId } } // Exclude the logged-in user
    ]
  }).select('profileDetails.firstName profileDetails.middleName profileDetails.lastName
profileDetails.profileImg bestTag');

  res.status(200).json({ success: true, results: studentResults });
}
```

```
    } catch (err) {  
      console.error('Error during search:', err);  
      res.status(500).json({ success: false, message: 'Internal server error' });  
    }  
  });
```

```
app.put("/api/updateProfile", verifyToken, upload.single("profileImg"), async (req, res) => {  
  try {  
    const userId = req.user.id;  
    const { role } = req.user;  
    const userModel = role === "student" ? Tupath_usersModel : Employer_usersModel;  
  
    const profileData = req.body;  
  
    // Handle file upload (if any)  
    if (req.file) {  
      profileData.profileImg = `/uploads/${req.file.filename}`;  
    }  
  
    // Ensure we preserve the existing projects data  
    const existingUser = await userModel.findById(userId);  
    if (!existingUser) {  
      return res.status(404).json({ success: false, message: "User not found" });  
    }  
  }  
});
```

```
// Preserve projects in the profileData (if no projects are passed, keep the existing ones)
const updatedProfile = {
  ...existingUser.profileDetails,
  ...profileData,
  projects: existingUser.profileDetails.projects || [] // Ensure existing projects are kept
};

// Update the user's profile details
const updatedUser = await userModel.findByIdAndUpdate(
  userId,
  { $set: { profileDetails: updatedProfile } },
  { new: true }
);

if (!updatedUser) {
  return res.status(404).json({ success: false, message: "User not found" });
}

res.status(200).json({ success: true, message: "Profile updated successfully",
updatedUser });
} catch (error) {
  console.error("Error updating profile:", error);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});
```

```
//-----DECEMBER 13
```

```
// Step 1: Add a reset token field to the user schemas
```

```
// Step 2: Endpoint to request password reset
```

```
app.post("/api/forgot-password", async (req, res) => {
```

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

```
  try {
```

```
    const user = await Tupath_usersModel.findOne({ email }) ||  
    Employer_usersModel.findOne({ email });
```

```
    if (!user) {
```

```
      return res.status(404).json({ success: false, message: "User not found" });
```

```
    }
```

```
    // Generate a reset token
```

```
    const resetToken = crypto.randomBytes(20).toString("hex");
```

```
    user.resetPasswordToken = resetToken;
```

```
    user.resetPasswordExpires = Date.now() + 3600000; // Token valid for 1 hour
```

```
    await user.save();
```

```
    // Send email
```

```
    const transporter = nodemailer.createTransport({
```

```
      service: "Gmail",
```

```
      auth: {
```

```
        user: "woojohnhenry2@gmail.com",
```

```
        pass: "efqk hxyw jpeq sndo",
```

```
},  
});
```

```
const resetLink = `http://localhost:5173/reset-password/${resetToken}`;
```

```
const mailOptions = {
```

```
  to: user.email,
```

```
  from: "no-reply@yourdomain.com",
```

```
  subject: "Password Reset Request",
```

```
  text: `You are receiving this because you (or someone else) requested the reset of your  
account's password.\n\nPlease click on the following link, or paste it into your browser to  
complete the process within one hour of receiving it:\n\n${resetLink}\n\nIf you did not  
request this, please ignore this email and your password will remain unchanged.` ,
```

```
};
```

```
await transporter.sendMail(mailOptions);
```

```
res.status(200).json({ success: true, message: "Reset link sent to email" });
```

```
} catch (error) {
```

```
  console.error("Error in forgot password endpoint:", error);
```

```
  res.status(500).json({ success: false, message: "Internal server error" });
```

```
}
```

```
});
```

```
// Step 3: Endpoint to reset password
```

```
app.post("/api/reset-password/:token", async (req, res) => {
```

```
  const { token } = req.params;
```

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

```
try {  
  const user = await Tupath_usersModel.findOne({  
    resetPasswordToken: token,  
    resetPasswordExpires: { $gt: Date.now() },  
  }) || Employer_usersModel.findOne({  
    resetPasswordToken: token,  
    resetPasswordExpires: { $gt: Date.now() },  
  });  
  
  if (!user) {  
    return res.status(400).json({ success: false, message: "Invalid or expired token" });  
  }  
  
  // Update password and clear reset token  
  user.password = await bcrypt.hash(newPassword, 10);  
  user.resetPasswordToken = undefined;  
  user.resetPasswordExpires = undefined;  
  await user.save();  
  
  res.status(200).json({ success: true, message: "Password reset successful" });  
} catch (error) {  
  console.error("Error in reset password endpoint:", error);  
  res.status(500).json({ success: false, message: "Internal server error" });  
}  
});
```

```
//for pushing purposes, please delete this comment later
```

```
//=====FOR ASSESSMENT  
QUESTIONS
```

```
// Add authentication check endpoint
```

```
app.get('/check-auth', async (req, res) => {
```

```
  try {
```

```
    const token = req.cookies.adminToken; // Assuming you're using cookies for admin auth
```

```
    if (!token) {
```

```
      return res.status(401).json({ success: false, message: 'No token found' });
```

```
    }
```

```
    const verified = jwt.verify(token, JWT_SECRET);
```

```
    if (!verified) {
```

```
      return res.status(401).json({ success: false, message: 'Invalid token' });
```

```
    }
```

```
    res.json({ success: true });
```

```
  } catch (error) {
```

```
    res.status(401).json({ success: false, message: 'Authentication failed' });
```

```
  }
```

```
});
```

```
// Update logout endpoint to clear cookie
```

```
app.post('/api/admin/logout', (req, res) => {
```



```
res.clearCookie('adminToken');  
res.json({ success: true, message: 'Logged out successfully' });  
});
```

```
// NEW API- HIWALAY KO LANG KASI BABAKLASIN KO TO
```

```
app.post("/api/saveAssessment", verifyToken, upload.single("ratingSlip"), async (req, res)  
=> {  
  try {  
    const { subject, grade, year, term } = req.body;  
    if (!subject || !grade || !year || !term) {  
      return res.status(400).json({ success: false, message: "Subject, grade, year level, and  
term are required." });  
    }  
  
    const ratingSlipPath = req.file ? `/uploads/${req.file.filename}` : null;  
  
    req.session.assessmentData = { subject, grade, ratingSlip: ratingSlipPath, year, term };  
  
    // Debugging: Log the stored session data  
    console.log("Saved assessment data in session:", req.session.assessmentData);
```

```
    res.status(200).json({ success: true, message: "Assessment saved successfully." });  
  } catch (error) {  
    console.error("Error saving assessment:", error);  
    res.status(500).json({ success: false, message: "Server error" });  
  }  
});
```

```
app.get("/api/topStudentsByTag", async (req, res) => {  
  try {  
    const topStudents = await Tupath_usersModel.find({ bestTag: { $exists: true } })  
      .sort({ "bestTagScores": -1 })  
      .limit(10); // Get top 10 students  
  
    res.status(200).json(topStudents);  
  } catch (error) {  
    console.error("Error fetching top students:", error);  
    res.status(500).json({ message: "Server error", error });  
  }  
});
```

```
app.get("/api/getSubjectByTag", async (req, res) => {  
  try {  
    const { tag } = req.query;  
  
    // Find the mapping document by tag
```

```

const mapping = await SubjectTagMapping.findOne({ tag });

if (!mapping || !mapping.subjects || mapping.subjects.length === 0) {
  return res.json({ success: false, message: "No subjects found for this tag." });
}

// Return subjects as an array of objects
res.json({ success: true, subjects: mapping.subjects });
} catch (error) {
  console.error("Error fetching subjects:", error);
  res.status(500).json({ success: false, message: "Server error" });
}
});

app.get('/api/grades', verifyToken, async (req, res) => {
  try {
    const userId = req.user.id;

    // Find all projects for the user
    const projects = await Project.find({
      // Assuming projects are linked to users in your schema
      // You may need to adjust this query based on your actual schema
    }).select('subject grade');

    // Transform projects into grades format
    const grades = projects.map(project => ({

```

```
code: project.subject,  
description: 'Project submission', // Or get from subject mapping  
grade: project.grade,  
corFile: null // Or include if you have this data  
}});
```

```
res.status(200).json({ success: true, grades });  
} catch (error) {  
  console.error('Error fetching grades:', error);  
  res.status(500).json({ success: false, message: 'Server error' });  
}  
});
```

```
// Update this route in your index.js file  
app.get("/api/checkExistingGrade", verifyToken, async (req, res) => {  
  try {  
    const { subject, year, term } = req.query;  
  
    if (!subject || !year || !term) {  
      return res.status(400).json({  
        success: false,  
        message: "Subject, year, and term are required."  
      });  
    }  
  }  
}
```

```
// Check if grade exists in any of the user's projects
```

```
const existingProject = await Project.findOne({
  user: req.user.id,
  subject,
  year,
  term
}).select("grade");

if (existingProject && existingProject.grade) {
  return res.status(200).json({
    success: true,
    grade: existingProject.grade
  });
}

return res.status(200).json({
  success: true,
  grade: null
});
} catch (error) {
  console.error("Error checking existing grade:", error);
  res.status(500).json({
    success: false,
    message: "Server error"
  });
}
});
```

```
// Server setup  
const PORT = process.env.PORT || 3001;  
server.listen(PORT, () => {  
  console.log(`Server running on port ${PORT}`);  
});
```

### **Index.js ( latest version with cloudinary):**

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const axios = require("axios");
const http = require("http");
const { Server } = require("socket.io");
const {
  Tupath_usersModel,
  Employer_usersModel,
  Project,
  Admin,
  SubjectTagMapping,
} = require("../models/Tupath_users");
const Post = require("../models/Post"); // Import Post model from models folder
const nodemailer = require("nodemailer");
const crypto = require("crypto");
const cookieParser = require("cookie-parser");
const session = require("express-session");
const MongoStore = require("connect-mongo");
const cloudinary = require("cloudinary").v2;
const { CloudinaryStorage } = require("multer-storage-cloudinary");
require("dotenv").config();
```

```
const adminsignup = require("./routes/adminsignup");
const adminLogin = require("./routes/adminLogin");
const questions = require("./routes/questions");
const userStats = require("./routes/userStats");
const studentTags = require("./routes/studentTags");
const studentByTags = require("./routes/studentsByTag");
const users = require("./routes/users");
const adminDelete = require("./routes/adminDelete");
const corRoutes = require("./routes/corRoutes");
const postRoutes = require("./routes/postRoutes"); // Import post routes
const adminRoutes = require("./routes/adminRoutes"); // Import admin routes
const messagingRoute = require("./routes/messagingRoute"); // Import messaging routes

// const checkAuth = require('./middleware/authv2')
const adminLogout = require("./routes/adminLogout");
const authRoute = require("./routes/authRoute"); // Add the auth routes
const verifyToken = require("./middleware/verifyToken"); // Use the separated middleware

// Environment variables
const JWT_SECRET = process.env.JWT_SECRET;
const GOOGLE_CLIENT_ID = process.env.GOOGLE_CLIENT_ID;
const CLIENT_URL = process.env.CLIENT_URL;
const PORT = process.env.PORT || 3001;
const MONGO_URI = process.env.MONGO_URI;
// const MONGO_LOCAL_URI = process.env.MONGO_LOCAL_URI;
const SESSION_SECRET = process.env.SESSION_SECRET;
```



```
const EMAIL_USER = process.env.EMAIL_USER;

const EMAIL_PASS = process.env.EMAIL_PASS;


const app = express();

const server = http.createServer(app);

const multer = require("multer");

const path = require("path");


// Middleware setup

app.use(cors({ origin: CLIENT_URL, credentials: true })); // Use environment variable for
CORS

app.use("/uploads", express.static("uploads"));

app.use("/certificates", express.static(path.join(__dirname, "certificates")));

app.use('/cor', express.static('cor'));


// Add this middleware first in your Express app

// app.use((req, res, next) => {

//   console.log(` Incoming ${req.method} request to ${req.path}` );

//   console.log("Headers:", req.headers);

//   console.log("Body:", req.body);

//   next();

// });


app.use(express.json({ limit: "50mb" }));

app.use(express.urlencoded({ limit: "50mb", extended: true }));

app.use(cookieParser());
```

```
app.use(
  session({
    secret: SESSION_SECRET,
    resave: false,
    saveUninitialized: false,
    store: MongoStore.create({ mongoUrl: MONGO_URI }),
    cookie: { maxAge: 24 * 60 * 60 * 1000 }, // 1 day
  })
);
```

```
// Make io available to routes
const io = new Server(server, {
  cors: {
    origin: CLIENT_URL,
    methods: ["GET", "POST"],
  },
});
```

```
// Add io to req object
app.use((req, res, next) => {
  req.io = io;
  next();
});
```

```
// ROUTES
```

```
app.use("/", users);
app.use("/", adminsignup);
app.use("/", adminLogin);
app.use("/", questions);
app.use("/", userStats);
app.use("/", studentTags);
app.use("/", studentByTags);
app.use("/", adminDelete);
// app.use('/', checkAuth);
app.use("/", adminLogout);
app.use("/", corRoutes);
app.use("/api", authRoute); // Use the auth routes
app.use("/api/posts", postRoutes); // Use post routes
app.use("/api/admin", adminRoutes); // Use admin routes
app.use('/adminsubjects', require('./routes/adminSubjects')); // use admin subjects routes
app.use("/api/messaging", messagingRoute); // Use messaging routes

// Middleware for setting COOP headers
app.use((req, res, next) => {
  res.setHeader("Cross-Origin-Opener-Policy", "same-origin-allow-popups"); // Added
  COOP header
  res.setHeader("Cross-Origin-Resource-Policy", "cross-origin"); // Added CORP header
  next();
});

// Connect to MongoDB using environment variables
```

mongoose

```
.connect(MONGO_URI)

.then(() => console.log("Connected to MongoDB successfully"))

.catch((err) => console.error("MongoDB connection error:", err));
```

cloudinary.config({

```
  cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
  api_key: process.env.CLOUDINARY_API_KEY,
  api_secret: process.env.CLOUDINARY_API_SECRET,
});
```

const storage = new CloudinaryStorage({

```
  cloudinary,
  params: {
    folder: 'TUPath_global', // Change this to your preferred folder name
    allowed_formats: ['jpg', 'png', 'jpeg'],
    transformation: [{ width: 500, height: 500, crop: 'limit' }],
  },
});
```

const Projectstorage = new CloudinaryStorage({

```
  cloudinary,
  params: {
    folder: 'TUPath_Proj', // Change this to your preferred folder name
    allowed_formats: ['jpg', 'png', 'jpeg'],
    transformation: [{ width: 500, height: 500, crop: 'limit' }],
```

```
},  
});
```

```
const CertThumbStorage = new CloudinaryStorage({  
  cloudinary: cloudinary,  
  params: (req, file) => {  
    return {  
      folder: 'TUPath_Cert_Thumbnails',  
      public_id: `${Date.now()}-${file.originalname}`,  
      allowed_formats: ['jpg', 'jpeg', 'png'],  
      transformation: [  
        { width: 300, height: 300, crop: 'limit' },  
        { quality: 'auto' }  
      ]  
    };  
  }  
});
```

```
const CertFileStorage = new CloudinaryStorage({  
  cloudinary: cloudinary,  
  params: async (req, file) => {  
    // Sanitize filename  
    const sanitizedName = file.originalname.replace(/^[^w.-]/g, "");  
    const ext = sanitizedName.split('.').pop().toLowerCase();  
  
    console.log(` Processing file: ${sanitizedName} (${ext}) `);
```

```

return {
  folder: 'TUPath_Cert_Attachments',
  public_id: `${Date.now()}-${sanitizedName}`,
  resource_type: ext === "docx" || ext === "txt" || ext === "pdf" ? "raw" : "auto", // Use 'raw'
  for non-images
};
}
});

```

```

const Profilestorage = new CloudinaryStorage({
  cloudinary,
  params: {
    folder: "TUPath_Profile",
    allowed_formats: ["jpg", "png", "jpeg"],
    transformation: [{ width: 500, height: 500, crop: "limit" }],
  },
});


```

//  Correct Multer Setup

```

const uploadImageProfile = multer({ storage: Profilestorage });
const upload = multer({ storage: storage });
const UploadImageProjects = multer({ storage: Projectstorage });

```

```
//  Multer Setup for Certificates

const uploadThumbnail = multer({
  storage: CertThumbStorage,
  limits: { fileSize: 5 * 1024 * 1024 } // 5MB
});

const uploadCertFiles = multer({
  storage: CertFileStorage,
  fileFilter: (req, file, cb) => {
    const allowedMimes = [
      'image/jpeg',
      'image/png',
      'application/pdf',
      'application/vnd.openxmlformats-officedocument.wordprocessingml.document', //
DOCX
      'text/plain'
    ];

    if (allowedMimes.includes(file.mimetype)) {
      cb(null, true);
    } else {
      cb(new Error(`Invalid file type: ${file.mimetype}`), false);
    }
  },
  limits: { fileSize: 15 * 1024 * 1024 } // 15MB limit
});
```

```
// Socket.IO events for real-time chat and certificates

io.on("connection", (socket) => {

  // Join a specific conversation room

  socket.on("join_conversation", (conversationId) => {

    socket.join(conversationId);

    console.log(` User joined conversation: ${conversationId} `);

  });


  // Handle sending messages

  socket.on("send_message", async (data) => {

    try {

      const { conversationId, message } = data;


      // Broadcast the message to everyone in the conversation except the sender

      socket.to(conversationId).emit("new_message", {

        conversationId,

        message

      });

    } catch (err) {

      console.error("Error handling send_message event:", err);

    }

  });


  // Handle typing indicators

  socket.on("typing", async (data) => {

    try {
```



```
const { conversationId, isTyping } = data;

const userId = socket.userId || "unknown"; // You may need to set this when user
connects

// Broadcast typing status to everyone in the conversation except the typer
socket.to(conversationId).emit("user_typing", {
  conversationId,
  isTyping,
  userId
});
} catch (err) {
  console.error("Error handling typing event:", err);
}
});

// Handle when a user disconnects
socket.on("disconnect", () => {
  console.log(` User disconnected: ${socket.id}` );
});

// ...existing socket.io code if any...
});

const validateAttachment = (url) => {
  const allowedExtensions = /\. (jpg|jpeg|png|pdf|docx|txt)$/i;
  return allowedExtensions.test(url);
}
```

```
};
```

```
const StudentCert = new mongoose.Schema({  
  StudId: { type: String, required: true },  
  StudName: { type: String, required: true },  
  timestamp: { type: Date, default: Date.now },  
  Certificate: {  
    CertName: { type: String, required: true },  
    CertDescription: { type: String, required: true },  
    CertThumbnail: { type: String, default: "" },  
    Attachments: [  
      {  
        type: String,  
        validate: {  
          validator: validateAttachment,  
          message:  
            "Attachments must be valid file URLs with extensions jpg, jpeg, png, pdf, docx, or txt.",  
        },  
      },  
    ],  
  },  
});
```

```
const StudentCertificate = mongoose.model("StudentCertificate", StudentCert);
```

```
/// Update your uploadCertificate endpoint
```

```
app.post("/api/uploadCertificate", verifyToken, async (req, res) => {  
  try {  
    console.log("Processing certificate upload...");  
  
    const { CertName, CertDescription, CertThumbnail, Attachments } = req.body;  
    const userId = req.user.id;  
    const userName = req.user.name;  
  
    console.log("Received data:", {  
      CertName,  
      CertDescription,  
      CertThumbnail,  
      Attachments,  
      userId,  
      userName  
    });  
  
    // Validate inputs  
    if (!CertName || !CertDescription) {  
      throw new Error("Certificate name and description are required");  
    }  
  
    if (!CertThumbnail) {  
      throw new Error("Thumbnail URL is required");  
    }  
  }  
}
```

```
if (!Attachments || !Array.isArray(Attachments) || Attachments.length === 0) {  
  throw new Error("At least one attachment is required");  
}
```

```
// Filter valid attachments
```

```
const validAttachments = Attachments.filter(url =>  
  url && /\. (jpg|jpeg|png|pdf|docx|txt)$/i.test(url)  
);
```

```
if (validAttachments.length === 0) {  
  throw new Error("No valid attachments provided");  
}
```

```
console.log("Creating new certificate document...");
```

```
const newCertificate = new StudentCertificate({  
  StudId: userId,  
  StudName: userName,  
  Certificate: {  
    CertName,  
    CertDescription,  
    CertThumbnail,  
    Attachments: validAttachments  
  }  
});
```

```
console.log("Saving to database...");
```

```
await newCertificate.save();
```

```
console.log("Emission new_certificate event");
```

```
io.emit("new_certificate", newCertificate);
```

```
console.log("Certificate saved successfully");
```

```
res.status(201).json({
```

```
  success: true,
```

```
  message: "Certificate saved successfully",
```

```
  certificate: newCertificate
```

```
});
```

```
} catch (error) {
```

```
  console.error("Certificate save error:", {
```

```
    message: error.message,
```

```
    stack: error.stack,
```

```
    requestBody: req.body,
```

```
    user: req.user
```

```
});
```

```
res.status(500).json({
```

```
  success: false,
```

```
  message: error.message || "Failed to save certificate",
```

```
  error: process.env.NODE_ENV === 'development' ? error.stack : undefined
```

```
});
```

```
}
```

```

});

app.post("/api/uploadThumbnail", verifyToken, uploadThumbnail.single('thumbnail'), async
(req, res) => {

  try {

    if (!req.file) {

      console.error("No file received in uploadThumbnail");

      return res.status(400).json({

        success: false,

        message: "No thumbnail file received"

      });

    }

    console.log("Cloudinary upload result:", req.file);

    if (!req.file.path) {

      throw new Error("Cloudinary upload failed - no URL returned");

    }

    res.status(201).json({

      success: true,

      message: "Thumbnail uploaded successfully",

      thumbnailUrl: req.file.path

    });

  } catch (error) {

    console.error("Thumbnail upload error:", {

```

```
    message: error.message,
    stack: error.stack,
    file: req.file
  });

  res.status(500).json({
    success: false,
    message: "Thumbnail upload failed",
    error: process.env.NODE_ENV === 'development' ? error.message : undefined
  });
}
});
```

// File validation middleware

```
const validateFileType = (req, res, next) => {
  if (!req.file) return next();

  const ext = req.file.originalname.split('.').pop().toLowerCase();
  const allowed = ['jpg', 'jpeg', 'png', 'pdf', 'docx', 'txt'];

  if (!allowed.includes(ext)) {
    return res.status(400).json({
      success: false,
      message: `Invalid file type .${ext}. Allowed: ${allowed.join(', ')}`
    });
  }
}
```

```
}  
next();  
};  
  
app.post("/api/uploadAttachments",  
  verifyToken,  
  uploadCertFiles.single('attachment'),  
  validateFileType,  
  async (req, res) => {  
    try {  
      if (!req.file) throw new Error("No file uploaded");  
  
      console.log("Upload successful:", req.file.path);  
      res.json({  
        success: true,  
        url: req.file.path,  
        filename: req.file.originalname  
      });  
  
    } catch (error) {  
      console.error("Upload failed:", error.message);  
      res.status(500).json({  
        success: false,  
        message: error.message  
      });  
    }  
  }  
}
```



```
}  
);
```

```
// Endpoint to fetch certificates for a user
```

```
app.get("/api/certificates", verifyToken, async (req, res) => {  
  try {  
    const userId = req.user.id;  
    const certificates = await StudentCertificate.find({ StudId: userId });  
    res.status(200).json({ success: true, certificates });  
  } catch (error) {  
    console.error("Error fetching certificates:", error);  
    res.status(500).json({ success: false, message: "Internal server error" });  
  }  
});
```

```
// Endpoint to delete a certificate
```

```
app.delete("/api/certificates/:id", verifyToken, async (req, res) => {  
  try {  
    const certificateId = req.params.id;  
    const userId = req.user.id;  
  
    const certificate = await StudentCertificate.findOneAndDelete({  
      _id: certificateId,  
      StudId: userId,  
    });  
  });
```

```

    if (!certificate) {
      return res
        .status(404)
        .json({ success: false, message: "Certificate not found" });
    }

    // Emit the delete certificate event
    io.emit("delete_certificate", { certificateId });

    res
      .status(200)
      .json({ success: true, message: "Certificate deleted successfully" });
  } catch (error) {
    console.error("Error deleting certificate:", error);
    res.status(500).json({ success: false, message: "Internal server error" });
  }
});

// Endpoint to fetch profile data for a specific user including projects and certificates
app.get("/api/profile/:id", verifyToken, async (req, res) => {
  const { id } = req.params;

  const requestingUserId = req.user.id;
  const requestingUserRole = req.user.role;

  try {
    // First check if the profile is a student

```

```

let user = await Tupath_usersModel.findById(id).populate({
  path: "profileDetails.projects",
  select: "projectName description tags tools thumbnail projectUrl",
});

// If not a student, check if it's an employer
if (!user) {
  user = await Employer_usersModel.findById(id);

  if (!user) {
    return res
      .status(404)
      .json({ success: false, message: "User not found" });
  }
}

let certificates = [];

// Only fetch certificates for student profiles
if (user.constructor.modelName === 'Tupath_user') {
  certificates = await StudentCertificate.find({ StudId: id });

  // Hide projects and certificates if the requesting user is another student
  if (
    requestingUserRole === "student" &&
    requestingUserId.toString() !== id.toString()
  )

```

```

){
  user.profileDetails.projects = [];
  certificates = [];
}
}

// Add role information to the response
const role = user.constructor.modelName === 'Tupath_user' ? 'student' : 'employer';

res.status(200).json({
  success: true,
  profile: {
    ...user.toObject(),
    role,
    profileDetails: {
      ...user.profileDetails,
      certificates,
    },
  },
});
} catch (err) {
  console.error("Error fetching profile:", err);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});

```

```
//-----NEWLY ADDED-----  
-----
```

```
app.post("/api/updateStudentProfile", verifyToken, async (req, res) => {  
  try {  
    const userId = req.user.id;  
    const {  
      studentId,  
      firstName,  
      lastName,  
      middleName,  
      department,  
      yearLevel,  
      dob,  
      profileImg,  
      gender,  
      address,  
      techSkills,  
      softSkills,  
      contact,  
    } = req.body;  
    // email  
  
    const updatedUser = await Tupath_usersModel.findByIdAndUpdate(  
      userId,  
      {
```

```
$set: {  
  profileDetails: {  
    studentId,  
    firstName,  
    lastName,  
    middleName,  
    department,  
    yearLevel,  
    dob,  
    profileImg,  
    gender,  
    address,  
    techSkills,  
    softSkills,  
    contact,  
    // email  
  },  
},  
},  
{ new: true, upsert: true }  
);
```

```
if (!updatedUser) {  
  return res  
    .status(404)  
    .json({ success: false, message: "User not found" });  
}
```

```
}
```

```
res.status(200).json({
```

```
  success: true,
```

```
  message: "Profile updated successfully",
```

```
  updatedUser,
```

```
});
```

```
} catch (error) {
```

```
  res.status(500).json({ success: false, message: "Internal server error" });
```

```
}
```

```
});
```

```
app.post("/api/updateEmployerProfile", verifyToken, async (req, res) => {
```

```
  try {
```

```
    const userId = req.user.id;
```

```
    const {
```

```
      firstName,
```

```
      lastName,
```

```
      middleName,
```

```
      dob,
```

```
      gender,
```

```
      nationality,
```

```
      address,
```

```
      profileImg,
```

```
      companyName,
```

```
      industry,
```

```
location,  
aboutCompany,  
contactPersonName,  
position,  
// email,  
phoneNumber,  
preferredRoles,  
internshipOpportunities,  
preferredSkills,  
} = req.body;
```

```
const updatedUser = await Employer_usersModel.findByIdAndUpdate(  
  userId,  
  {  
    $set: {  
      profileDetails: {  
        firstName,  
        lastName,  
        middleName,  
        dob,  
        gender,  
        nationality,  
        address,  
        profileImg,  
        companyName,  
        industry,
```



```
    location,  
    aboutCompany,  
    contactPersonName,  
    position,  
    // email,  
    phoneNumber,  
    preferredRoles,  
    internshipOpportunities,  
    preferredSkills,  
  },  
},  
},  
{ new: true, upsert: true }  
);
```

```
if (!updatedUser) {  
  return res  
    .status(404)  
    .json({ success: false, message: "User not found" });  
}
```

```
res.status(200).json({  
  success: true,  
  message: "Profile updated successfully",  
  updatedUser,  
});
```

```
    } catch (error) {  
      res.status(500).json({ success: false, message: "Internal server error" });  
    }  
  });
```

// Profile fetching endpoint

```
app.get("/api/profile", verifyToken, async (req, res) => {  
  try {  
    const userId = req.user.id;  
    const role = req.user.role; // Extract role from the token  
  
    const userModel =  
      role === "student" ? Tupath_usersModel : Employer_usersModel;  
    const user = await userModel  
      .findById(userId)  
      .select("email role profileDetails createdAt googleSignup")  
      .populate({  
        path: "profileDetails.projects",  
        select: "projectName description tags tools thumbnail projectUrl",  
        strictPopulate: false, // Allow flexible population  
      });  
  
    if (!user) {  
      return res  
        .status(404)  
        .json({ success: false, profile: "User not Found" });  
    }  
  }  
});
```

```
}
```

```
// Fetch certificates for the user
```

```
const certificates = await StudentCertificate.find({ StudId: userId });
```

```
// Return profile details tailored to the role
```

```
const profile = {
```

```
  email: user.email,
```

```
  role: user.role,
```

```
  profileDetails:
```

```
    user.role === "student"
```

```
    ? {
```

```
      ...user.profileDetails,
```

```
      certificates, // Include certificates in the profile details
```

```
    }
```

```
    : {
```

```
      ...user.profileDetails,
```

```
      companyName: user.profileDetails.companyName || null,
```

```
      industry: user.profileDetails.industry || null,
```

```
      aboutCompany: user.profileDetails.aboutCompany || null,
```

```
      preferredRoles: user.profileDetails.preferredRoles || [],
```

```
      internshipOpportunities:
```

```
        user.profileDetails.internshipOpportunities || false,
```

```
    },
```

```
  createdAt: user.createdAt,
```

```
  googleSignup: user.googleSignup,
```

```

};

res.status(200).json({ success: true, profile });
} catch (error) {
  console.error("Error fetching profile:", error);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});

/*app.post("/api/uploadProfileImage", verifyToken, upload.single("profileImg"), async (req,
res) => {
  try {
    const userId = req.user.id;

    // Validate if file exists
    if (!req.file) {
      return res.status(400).json({ success: false, message: "No file uploaded" });
    }

    const profileImgPath = `/uploads/${req.file.filename}`;
    console.log("Uploaded file path:", profileImgPath); // Debugging

    // Update for both student and employer models
    const updatedStudent = await Tupath_usersModel.findByIdAndUpdate(
      userId,
      { $set: { "profileDetails.profileImg": profileImgPath } },

```

```

        { new: true }
    );

    const updatedEmployer = await Employer_usersModel.findByIdAndUpdate(
        userId,
        { $set: { "profileDetails.profileImg": profileImgPath } },
        { new: true }
    );

    // If no user was updated, return an error
    if (!updatedStudent && !updatedEmployer) {
        console.log("User not found for ID:", userId); // Debugging
        return res.status(404).json({ success: false, message: "User not found" });
    }

    res.status(200).json({
        success: true,
        message: "Profile image uploaded successfully",
        profileImg: profileImgPath,
    });
} catch (error) {
    console.error("Error uploading profile image:", error);
    res.status(500).json({ success: false, message: "Internal server error" });
}
});
*/

```


```
// api upload image endpoint
app.post(
  "/api/uploadProfileImage",
  verifyToken,
  uploadImageProfile.single("profileImg"),
  async (req, res) => {
    try {
      const userId = req.user.id;

      if (!req.file) {
        return res.status(400).json({
          success: false,
          message: "No file uploaded or Cloudinary failed",
        });
      }

      console.log("Uploaded File:", req.file); // 🪲 Debugging

      const profileImgUrl = req.file.path; // ✅ Cloudinary should return a URL

      if (!profileImgUrl) {
        return res.status(500).json({
          success: false,
          message: "Cloudinary upload failed, no URL returned",
        });
      }
    }
  }
}
```

```
//  Select the correct user model based on role
```

```
const userModel =
```

```
req.user.role === "student" ? Tupath_usersModel : Employer_usersModel;
```

```
//  Update user profile
```

```
const updatedUser = await userModel.findByIdAndUpdate(
```

```
  userId,
```

```
  { $set: { "profileDetails.profileImg": profileImgUrl } },
```

```
  { new: true } 
```

```
);
```

```
if (!updatedUser) {
```

```
  return res
```

```
    .status(404)
```

```
    .json({ success: false, message: "User not found" });
```

```
}
```

```
res.status(200).json({
```

```
  success: true,
```

```
  message: "Profile image uploaded successfully",
```

```
  profileImg: profileImgUrl,
```

```
});
```

```
} catch (error) {
```

```
  console.error("Error uploading profile image:", error);
```

```
  res.status(500).json({ success: false, message: "Internal server error" });
```

```

    }
  });

  // Modify API Endpoint to Use Cloudinary
  app.post("/api/uploadProject", verifyToken, upload.fields([
    { name: "thumbnail", maxCount: 1 },
    { name: "selectedFiles", maxCount: 10 },
    { name: "ratingSlip", maxCount: 1 }
  ]), async (req, res) => {
    try {
      console.log("Received project upload request with data:", req.body);

      // Retrieve saved subject & grade from session
      const { subject, grade, ratingSlip, year, term } = req.session.assessmentData || {};

      if (!subject || !grade || !year || !term) {
        console.error("Missing subject or grade in session.");
        return res.status(400).json({ success: false, message: "Missing required fields." });
      }

      // Get Cloudinary URL instead of local file path
      const thumbnail = req.files?.["thumbnail"]?.[0]?.filename ?
        `/uploads/${req.files["thumbnail"][0].filename}` : null;

      const selectedFiles = req.files?.["selectedFiles"] ? req.files["selectedFiles"].map(file =>
        file.path) : [];

      const ratingSlipPath = req.files?.["ratingSlip"]?.[0]?.filename
        ? `/uploads/${req.files["ratingSlip"][0].filename}`

```



: ratingSlip;

```
const newProject = new Project({  
  user: req.user.id, // Add this line  
  projectName: req.body.projectName,  
  description: req.body.description,  
  tag: req.body.tag,  
  tools: req.body.tools,  
  projectUrl: req.body.projectUrl,  
  roles: req.body.roles,  
  subject,  
  grade,  
  ratingSlip: ratingSlipPath,  
  status: "pending",  
  thumbnail,  
  selectedFiles,  
  year,  
  term,  
});
```

```
await newProject.save();
```

```
const userId = req.user?.id || req.body.userId; // Get user ID from auth or request body  
if (!userId) {  
  console.error("User ID is missing.");  
  return res.status(400).json({ success: false, message: "User ID is required." });  
}
```

```
}
```

```
const user = await Tupath_usersModel.findOneAndUpdate(  
  { _id: userId },  
  { $addToSet: { "profileDetails.projects": newProject._id } },  
  { new: true }  
);
```

```
if (!user) {  
  console.error("User not found.");  
  return res.status(404).json({ success: false, message: "User not found." });  
}
```

```
await user.calculateBestTag();
```

```
//clear session after a successful save
```

```
delete req.session.assessmentData;
```

```
console.log("Project successfully saved and linked to user:", newProject);
```

```
res.status(201).json({ success: true, project: newProject });
```

```
} catch (error) {  
  //console.error("Error uploading project:", error);  
  console.error("Detailed upload error:", {  
    message: error.message,  
    stack: error.stack,
```

```
    sessionData: req.session.assessmentData,  
    files: req.files,  
    body: req.body  
  });  
  res.status(500).json({ success: false, message: "Server error" });  
}  
});
```

```
/* //BACKUP LATEST API
```

```
app.post("/api/uploadProject", verifyToken, upload.fields([  
  { name: "thumbnail", maxCount: 1 },  
  { name: "selectedFiles", maxCount: 10 },  
  { name: "ratingSlip", maxCount: 1 }  
]), async (req, res) => {  
  try {  
    console.log("Received project upload request with data:", req.body);  
    console.log("Session before accessing assessmentData:", req.session);  
  
    // Retrieve saved subject & grade from session  
    const { subject, grade, ratingSlip } = req.session.assessmentData || {};  
  
    console.log("Extracted assessment data from session:", { subject, grade, ratingSlip });
```

```
if (!subject || !grade) {  
  console.error("Missing subject or grade in session.");  
  return res.status(400).json({ success: false, message: "Missing required fields." });  
}
```

```
const thumbnail = req.files?.["thumbnail"]?.[0]?.filename ?  
`/uploads/${req.files["thumbnail"][0].filename}` : null;  
  
const selectedFiles = req.files?.["selectedFiles"] ? req.files["selectedFiles"].map(file =>  
file.path) : [];  
  
const ratingSlipPath = req.files?.["ratingSlip"]?.[0]?.filename  
? `/uploads/${req.files["ratingSlip"][0].filename}`  
: ratingSlip;
```

```
const newProject = new Project({  
  projectName: req.body.projectName,  
  description: req.body.description,  
  tag: req.body.tag,  
  tools: req.body.tools,  
  projectUrl: req.body.projectUrl,  
  roles: req.body.roles,  
  subject,  
  grade,  
  ratingSlip: ratingSlipPath, // <-- Save ratingSlip instead of corFile  
  status: "pending",  
  thumbnail,  
  selectedFiles  
});
```

```

    await newProject.save();

    // Clear session after successful save
    delete req.session.assessmentData;

    console.log("Project successfully saved:", newProject);

    res.status(201).json({ success: true, project: newProject });
  } catch (error) {
    console.error("Error saving project:", error);
    res.status(500).json({ success: false, message: "Server error" });
  }
});

*/

app.get("/api/projects", verifyToken, async (req, res) => {
  try {
    const userId = req.user.id;

    // Fetch user with populated projects
    const user = await Tupath_usersModel.findById(userId).populate(
      "profileDetails.projects"
    );
  }
});

```

```
if (!user) {  
  return res  
    .status(404)  
    .json({ success: false, message: "User not found" });  
}
```

```
// Add scores and tag summary for each project  
const projectsWithScores = user.profileDetails.projects.map((project) => {  
  const totalScore = project.assessment.reduce(  
    (sum, question) => sum + (question.weightedScore || 0),  
    0  
  );
```

```
  return {  
    _id: project._id,  
    projectName: project.projectName,  
    description: project.description,  
    tag: project.tag,  
    totalScore, // Sum of all weighted scores for the project  
    tools: project.tools,  
    status: project.status,  
    assessment: project.assessment, // Include detailed assessment  
    createdAt: project.createdAt,  
    roles: project.roles,  
  };  
});
```

```
res.status(200).json({
  success: true,
  projects: projectsWithScores,
});
} catch (error) {
  console.error("Error fetching projects:", error);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});
```

```
app.delete("/api/projects/:projectId", verifyToken, async (req, res) => {
  try {
    const userId = req.user.id; // Extract user ID from the token
    const { projectId } = req.params;

    // Find the user
    const user = await Tupath_usersModel.findById(userId);
    if (!user) {
      return res
        .status(404)
        .json({ success: false, message: "User not found" });
    }

    // Find the project in the user's profile and remove it
    const projectIndex = user.profileDetails.projects.findIndex(
```

```
(project) => project._id.toString() === projectId
);
if (projectIndex === -1) {
  return res.status(404).json({
    success: false,
    message: "Project not found in user's profile",
  });
}

// Remove the project reference from the user's profile
user.profileDetails.projects.splice(projectIndex, 1);
await user.save();

// Delete the project document from the 'projects' collection
const deletedProject = await Project.findByIdAndDelete(projectId);
if (!deletedProject) {
  return res.status(404).json({
    success: false,
    message: "Project not found in projects collection",
  });
}

res.status(200).json({
  success: true,
  message:
    "Project deleted successfully from user's profile and projects collection",
```



```

});

} catch (error) {

  console.error("Error deleting project:", error);

  res.status(500).json({ success: false, message: "Internal server error" });

}

});

// // Endpoint for uploading certificate photos

// app.post("/api/uploadCertificate", verifyToken, upload.array("certificatePhotos", 3), async
// (req, res) => {

//   try {

//     const userId = req.user.id;

//     const filePaths = req.files.map(file => `/certificates/${file.filename}`);

//     const updatedUser = await Tupath_usersModel.findByIdAndUpdate(
//       userId,
//       { $push: { "profileDetails.certificatePhotos": { $each: filePaths } } },
//       { new: true }
//     );

//     if (!updatedUser) {

//       return res.status(404).json({ success: false, message: "User not found" });

//     }

//     res.status(200).json({ success: true, message: "Certificate photos uploaded
// successfully", certificatePhotos: filePaths });

//   } catch (error) {

```

```

// console.error("Error uploading certificate photos:", error);

// res.status(500).json({ success: false, message: "Internal server error" });

// }

// });

// -----api for dynamic search-----

app.get("/api/search", verifyToken, async (req, res) => {

  const { query } = req.query;

  if (!query) {

    return res

      .status(400)

      .json({ success: false, message: "Query parameter is required" });

  }

  try {

    const regex = new RegExp(query, "i"); // Case-insensitive regex

    const loggedInUserId = req.user.id; // Assuming verifyToken populates req.user with user
    details

    // Search students only

    const studentResults = await Tupath_usersModel.find({

      $and: [

        {

          $or: [

            { "profileDetails.firstName": regex },

```

```

    { "profileDetails.middleName": regex },
    { "profileDetails.lastName": regex },
    { bestTag: regex }, // Search by `bestTag`
  ],
},
{ _id: { $ne: loggedInUserId } }, // Exclude the logged-in user
],
}).select(
  "profileDetails.firstName profileDetails.middleName profileDetails.lastName
profileDetails.profileImg bestTag"
);

```

```

res.status(200).json({ success: true, results: studentResults });
} catch (err) {
  console.error("Error during search:", err);
  res.status(500).json({ success: false, message: "Internal server error" });
}
});

```

```

app.put(
  "/api/updateProfile",
  verifyToken,
  upload.single("profileImg"),
  async (req, res) => {
    try {
      const userId = req.user.id;

```

```
const { role } = req.user;

const userModel =
  role === "student" ? Tupath_usersModel : Employer_usersModel;

const profileData = req.body;

// Find existing user
const existingUser = await userModel.findById(userId);
if (!existingUser) {
  return res
    .status(404)
    .json({ success: false, message: "User not found" });
}

// 🔥 **Delete old profile image from Cloudinary before uploading a new one**
if (existingUser.profileDetails.profileImg) {
  const oldImageUrl = existingUser.profileDetails.profileImg;
  const publicId = oldImageUrl.split("/").pop().split(".")[0]; // Extract publicId
  await cloudinary.uploader.destroy(publicId);
}

// Handle file upload (if any)
if (req.file) {
  const uploadedImage = await cloudinary.uploader.upload(req.file.path);
  profileData.profileImg = uploadedImage.secure_url; // Store the Cloudinary URL
}
```

```
// Preserve existing projects and update profile
```

```
const updatedProfile = {  
  ...existingUser.profileDetails,  
  ...profileData,  
  projects: existingUser.profileDetails.projects || [],  
};
```

```
// Update user profile
```

```
const updatedUser = await userModel.findByIdAndUpdate(  
  userId,  
  { $set: { profileDetails: updatedProfile } },  
  { new: true }  
);
```

```
// 🧑 **Update all posts where userId matches the updated user**
```

```
if (profileData.profileImg) {  
  await Post.updateMany(  
    { userId },  
    { $set: { profileImg: profileData.profileImg } }  
  );  
}
```

```
// 🧑 **Update all comments where userId matches**
```

```
await Post.updateMany(  
  { "comments.userId": userId },  
  { $set: { "comments.$[elem].profileImg": profileData.profileImg } },
```

```

    { arrayFilters: [{ "elem.userId": userId }] }
  );

  // Commented out Message updates since the Message model isn't defined
  // If you need this functionality, import the Message model at the top of the file
  /*
  // 🔥 **Update profile image in messages where user is the sender**
  await Message.updateMany(
    { "sender.senderId": userId },
    { $set: { "sender.profileImg": profileData.profileImg } }
  );

  // 🔥 **Update profile image in messages where user is the receiver**
  await Message.updateMany(
    { "receiver.receiverId": userId },
    { $set: { "receiver.profileImg": profileData.profileImg } }
  );
  */
}

res.status(200).json({
  success: true,
  message: "Profile updated successfully",
  updatedUser,
});
} catch (error) {

```

```

    console.error("Error updating profile:", error);

    res

    .status(500)

    .json({ success: false, message: "Internal server error" });
  }
}
);

```

//-----DECEMBER 13

// Step 1: Add a reset token field to the user schemas

// Step 2: Endpoint to request password reset

```

app.post("/api/forgot-password", async (req, res) => {
  const { email } = req.body;

  try {
    const user =
      (await Tupath_usersModel.findOne({ email })) ||
      Employer_usersModel.findOne({ email });

    if (!user) {
      return res

        .status(404)

        .json({ success: false, message: "User not found" });
    }
  }
}

```

// Generate a reset token

```

const resetToken = crypto.randomBytes(20).toString("hex");

```

```
user.resetPasswordToken = resetToken;
user.resetPasswordExpires = Date.now() + 3600000; // Token valid for 1 hour
await user.save();
```

```
// Send email with environment variables
```

```
const transporter = nodemailer.createTransport({
  service: "Gmail",
  auth: {
    user: EMAIL_USER,
    pass: EMAIL_PASS,
  },
});
```

```
const resetLink = `${CLIENT_URL}/reset-password/${resetToken}`;
```

```
const mailOptions = {
```

```
  to: user.email,
```

```
  from: "no-reply@yourdomain.com",
```

```
  subject: "Password Reset Request",
```

```
  text: `You are receiving this because you (or someone else) requested the reset of your
account's password.\n\nPlease click on the following link, or paste it into your browser to
complete the process within one hour of receiving it:\n\n${resetLink}\n\nIf you did not
request this, please ignore this email and your password will remain unchanged.` ,
```

```
};
```

```
await transporter.sendMail(mailOptions);
```

```
res
```

```
.status(200)
```



```
    .json({ success: true, message: "Reset link sent to email" });  
  } catch (error) {  
    console.error("Error in forgot password endpoint:", error);  
    res.status(500).json({ success: false, message: "Internal server error" });  
  }  
});
```

// Step 3: Endpoint to reset password

```
app.post("/api/reset-password/:token", async (req, res) => {
```

```
  const { token } = req.params;
```

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

```
  try {
```

```
    const user =
```

```
      (await Tupath_usersModel.findOne({
```

```
        resetPasswordToken: token,
```

```
        resetPasswordExpires: { $gt: Date.now() },
```

```
      })) ||
```

```
      Employer_usersModel.findOne({
```

```
        resetPasswordToken: token,
```

```
        resetPasswordExpires: { $gt: Date.now() },
```

```
      });
```

```
  if (!user) {
```

```
    return res
```

```
      .status(400)
```

```

        .json({ success: false, message: "Invalid or expired token" });
    }

    // Update password and clear reset token
    user.password = await bcrypt.hash(newPassword, 10);
    user.resetPasswordToken = undefined;
    user.resetPasswordExpires = undefined;
    await user.save();

    res
        .status(200)
        .json({ success: true, message: "Password reset successful" });
    } catch (error) {
        console.error("Error in reset password endpoint:", error);
        res.status(500).json({ success: false, message: "Internal server error" });
    }
});

//for pushing purposes, please delete this comment later

//=====FOR ASSESSMENT
QUESTIONS

// Add authentication check endpoint
app.get("/check-auth", async (req, res) => {
    try {

```

```
const token = req.cookies.adminToken; // Assuming you're using cookies for admin auth
if (!token) {
  return res
    .status(401)
    .json({ success: false, message: "No token found" });
}

const verified = jwt.verify(token, JWT_SECRET);
if (!verified) {
  return res.status(401).json({ success: false, message: "Invalid token" });
}

res.json({ success: true });
} catch (error) {
  res.status(401).json({ success: false, message: "Authentication failed" });
}
});

// Update logout endpoint to clear cookie
app.post("/api/admin/logout", (req, res) => {
  res.clearCookie("adminToken");
  res.json({ success: true, message: "Logged out successfully" });
});

// NEW API- HIWALAY KO LANG KASI BABAKLASIN KO TO
```

```
app.post(
  "/api/saveAssessment",
  verifyToken,
  upload.single("ratingSlip"),
  async (req, res) => {
    try {
      const { subject, grade, year, term } = req.body;
      if (!subject || !grade || !year || !term) {
        return res.status(400).json({
          success: false,
          message: "Subject, grade, year level, and term are required.",
        });
      }

      const ratingSlipPath = req.file ? `/uploads/${req.file.filename}` : null;

      req.session.assessmentData = {
        subject,
        grade,
        ratingSlip: ratingSlipPath,
        year,
        term,
      };

      // Debugging: Log the stored session data
      console.log(
```

```

    "Saved assessment data in session:",
    req.session.assessmentData
  );

  res
    .status(200)
    .json({ success: true, message: "Assessment saved successfully." });
} catch (error) {
  console.error("Error saving assessment:", error);
  res.status(500).json({ success: false, message: "Server error" });
}
};

app.get("/api/topStudentsByTag", async (req, res) => {
  try {
    const topStudents = await Tupath_usersModel.find({
      bestTag: { $exists: true },
    })
      .sort({ bestTagScores: -1 })
      .limit(10); // Get top 10 students

    res.status(200).json(topStudents);
  } catch (error) {
    console.error("Error fetching top students:", error);
    res.status(500).json({ message: "Server error", error });
  }
});

```

```

    }
  });

app.get("/api/getSubjectByTag", async (req, res) => {
  try {
    const { tag } = req.query;

    // Find the mapping document by tag
    const mapping = await SubjectTagMapping.findOne({ tag });

    if (!mapping || !mapping.subjects || mapping.subjects.length === 0) {
      return res.json({
        success: false,
        message: "No subjects found for this tag.",
      });
    }

    // Return subjects as an array of objects
    res.json({ success: true, subjects: mapping.subjects });
  } catch (error) {
    console.error("Error fetching subjects:", error);
    res.status(500).json({ success: false, message: "Server error" });
  }
});

// studentcount in clientdashboard

```

```
app.get("/api/student-counts", async (req, res) => {
  try {
    const bsitCount = await Tupath_usersModel.countDocuments({
      "profileDetails.department": "Information Technology",
    });
    const bscsCount = await Tupath_usersModel.countDocuments({
      "profileDetails.department": "Computer Science",
    });
    const bsisCount = await Tupath_usersModel.countDocuments({
      "profileDetails.department": "Information System",
    });

    res.json({
      success: true,
      counts: {
        BSIT: bsitCount,
        BSCS: bscsCount,
        BSIS: bsisCount,
      },
    });
  } catch (error) {
    console.error("Error fetching student counts:", error);
    res.status(500).json({ success: false, message: "Server Error" });
  }
});
```

```
// Update this route in your index.js file

app.get("/api/checkExistingGrade", verifyToken, async (req, res) => {
  try {
    const { subject, year, term } = req.query;

    if (!subject || !year || !term) {
      return res.status(400).json({
        success: false,
        message: "Subject, year, and term are required."
      });
    }

    // Check if grade exists in any of the user's projects
    const existingProject = await Project.findOne({
      user: req.user.id,
      subject,
      year,
      term
    }).select("grade");

    if (existingProject && existingProject.grade) {
      return res.status(200).json({
        success: true,
        grade: existingProject.grade
      });
    }
  }
}
```



```
return res.status(200).json({
  success: true,
  grade: null
});
} catch (error) {
  console.error("Error checking existing grade:", error);
  res.status(500).json({
    success: false,
    message: "Server error"
  });
}
});

// Server setup
server.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

**ProjectUpModal.jsx:**

```
import React, { useState, useEffect, useRef } from "react";  
  
import { motion } from "framer-motion";  
  
import ProjectAssessmentModal from "../ProjectAssessmentModal";  
  
import "../popups/ProjectUpModal.css";  
  
import axiosInstance from "../../services/axiosInstance";  
  
import { toast } from "react-toastify"; // Import toastify  
  
import "react-toastify/dist/ReactToastify.css"; // Don't forget to import the CSS for toastify
```

```
const predefinedTags = [  
  "Web Development",  
  "Software Development and Applications",  
];
```

```
const predefinedTools = [  
  "React",  
  "Node.js",  
  "Laravel",  
  "PHP",  
];
```

```
const predefinedRoles = [  
  "Frontend Developer",  
  "UI/UX Designer",  
  "Angular Developer",
```

```
"Backend Developer",  
"HTML/CSS Specialist",
```

```
];
```

```
const ProjectUploadModal = ({ show, onClose, updateGradesTable }) => {  
  const [projectName, setProjectName] = useState("");  
  const [description, setDescription] = useState("");  
  const [selectedFiles, setSelectedFiles] = useState([]);  
  const [tag, setTag] = useState("");  
  const [tools, setTools] = useState([]);  
  const [roles, setRoles] = useState([]); // Add this state variable  
  
  const [thumbnail, setThumbnail] = useState(null);  
  const [projectUrl, setProjectUrl] = useState("");  
  const [status] = useState("pending");  
  const [showAssessmentModal, setShowAssessmentModal] = useState(false);  
  
  const [isSubmitting, setIsSubmitting] = useState(false);  
  const [selectedSubject, setSelectedSubject] = useState("");  
  const [grade, setGrade] = useState("");  
  const [year, setSelectedYear] = useState("");  
  const [term, setSelectedTerm] = useState("");
```

```
const thumbnailInputRef = useRef(null);
```

```
const fileInputRef = useRef(null);
```

```
const [availableSubjects, setAvailableSubjects] = useState([]);
```

```
const handleSubmit = (e) => {
```

```
  e.preventDefault();
```

```
  if (!projectName.trim() || !description.trim() || !tag) {
```

```
    alert("All fields are required.");
```

```
    return;
```

```
  }
```

```
  setShowAssessmentModal(true);
```

```
};
```

```
const handleFileChange = (e) => {
```

```
  setSelectedFiles([...selectedFiles, ...Array.from(e.target.files)]);
```

```
};
```

```
const handleFileRemove = (fileToRemove) => {
```

```
  setSelectedFiles(selectedFiles.filter((file) => file !== fileToRemove));
```

```
};
```

```
const handleThumbnailChange = (e) => {
```

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

```
if (file) {  
  setThumbnail(file);  
}  
};
```

```
const handleTagSelect = async (e) => {  
  const selectedTag = e.target.value;  
  setTag(selectedTag);  
  
};
```

```
useEffect(() => {  
  if (tag) {  
    fetchSubjectsForTag(tag);  
  }  
}, [tag]);
```

```
const fetchSubjectsForTag = async (selectedTag) => {  
  try {  
    const response = await axiosInstance.get(`/api/getSubjectByTag?tag=${selectedTag}`);  
    console.log("Subjects Fetched:", response.data);  
    if (response.data.success) {  
      setAvailableSubjects(response.data.subjects);  
      setSelectedSubject(response.data.subjects.length === 1 ?  
response.data.subjects[0].subjectCode : "");  
    } else {
```

```
    setAvailableSubjects([]);
    setSelectedSubject("");
  }
} catch (error) {
  console.error("Error fetching subjects:", error);
}
};
```

```
const handleToolSelect = (e) => {
  const selectedTool = e.target.value;
  if (selectedTool && !tools.includes(selectedTool)) {
    setTools([...tools, selectedTool]);
  }
};
```

```
const handleRoleSelect = (e) => {
  const selectedRole = e.target.value;
  if (selectedRole && !roles.includes(selectedRole)) {
    setRoles([...roles, selectedRole]);
  }
};
```

```
const handleRoleRemove = (roleToRemove) => {
  setRoles(roles.filter((role) => role !== roleToRemove));
};
```

```
const handleToolRemove = (toolToRemove) => {  
  setTools(tools.filter((tool) => tool !== toolToRemove));  
};
```

```
const handleAssessmentSubmit = async ({ subjectCode, grade, year, term }) => {  
  try {  
    const response = await axiosInstance.post("/api/saveAssessment", {  
      subject: subjectCode,  
      grade: grade,  
      year: year,  
      term: term,  
    });
```

```
    if (response.data.success) {  
      setSelectedSubject(subjectCode);  
      setGrade(grade);  
      setSelectedYear(year);  
      setSelectedTerm(term);  
      setShowAssessmentModal(false);  
      toast.success("Subject and grade saved successfully!");  
    } else {  
      toast.error("Failed to save subject and grade.");
```

```
    }  
  } catch (error) {  
    console.error("Error saving assessment:", error);  
    toast.error("An error occurred while saving the assessment.");  
  }  
};
```

```
const handleFinalSubmit = async () => {  
  if (!selectedSubject || !grade) {  
    alert("Please select a subject and enter a grade before final submission.");  
    return;  
  }  
}
```

```
setIsSubmitting(true);
```

```
const formData = new FormData();  
formData.append("projectName", projectName);  
formData.append("description", description);  
formData.append("tag", tag);  
tools.forEach((tool) => formData.append("tools", tool));  
roles.forEach((role) => formData.append("roles", role));  
formData.append("projectUrl", projectUrl);  
formData.append("subject", selectedSubject);  
formData.append("grade", grade);
```



```
selectedFiles.forEach((file, index) => {  
    formData.append("selectedFiles", file);  
});
```

```
if (thumbnail) {  
    formData.append("thumbnail", thumbnail);  
}
```

```
// Debugging: Log Form Data
```

```
console.log("Submitting Project Data:");  
console.log("Project Name:", projectName);  
console.log("Description:", description);  
console.log("Tag:", tag);  
console.log("Tools:", tools);  
console.log("Roles:", roles);  
console.log("Project URL:", projectUrl);  
console.log("Selected Subject:", selectedSubject);  
console.log("Grade:", grade);  
console.log("Thumbnail:", thumbnail ? thumbnail.name : "No thumbnail");  
console.log("Selected Files:", selectedFiles.map(file => file.name));
```

```
try {  
    const response = await axiosInstance.post("/api/uploadProject", formData, {  
        headers: { "Content-Type": "multipart/form-data" },  
    });  
};
```

```
if (response.data.success) {  
    toast.success("Project uploaded successfully!");  
  
    // Now update GradesTable  
    updateGradesTable({  
        code: selectedSubject,  
        description: "Subject related to project",  
        grade  
    });  
  
    onClose();  
} else {  
    console.error("Upload failed:", response.data.message);  
}  
} catch (error) {  
    console.error("Error uploading project:", error);  
}  
  
setIsSubmitting(false);  
};  
  
if (!show) return null;  
  
return (  
    <>
```

```
<div className="ProjectUploadModal-overlay" onClick={onClose}>
```

```
<motion.div
```

```
  className="ProjectUploadModal-content"
```

```
  onClick={(e) => e.stopPropagation()}
```

```
  initial={{ opacity: 0, y: -30 }}
```

```
  animate={{ opacity: 1, y: 0 }}
```

```
  exit={{ opacity: 0, y: 20 }}
```

```
>
```

```
<div className="upheader">
```

```
<h3>Upload Your Project</h3>
```

```
<button className="projectup-close-btn" onClick={onClose}>
```

```
  x
```

```
</button>
```

```
</div>
```

```
<form id="projup-form">
```

```
<div className="leftprojup-container">
```

```
<div className="top">
```

```
<label>Project Name:</label>
```

```
<input
```

```
  type="text"
```

```
  value={projectName}
```

```
  onChange={(e) => setProjectName(e.target.value)}
```

```
/>
```

```
</div>
```

```
<div className="mid">
```

```
<label>Description:</label>

<textarea

  value={description}

  onChange={(e) => setDescription(e.target.value)}

></textarea>

</div>
```

```
<div className="bottom">
```

```
<label>Category</label>

<select value={tag} onChange={handleTagSelect}>

  <option value="">Select a Category</option>

  {predefinedTags.map((tag, index) => (

    <option key={index} value={tag}>

      {tag}

    </option>

  ))}

</select>
```

```
<div className="tools-input-container">
```

```
<label>Tools Used:</label>

<select onChange={handleToolSelect}>

  <option value="">Select a Tool</option>

  {predefinedTools.map((tool, index) => (

    <option key={index} value={tool}>
```

```
    {tool}
  </option>
  )})
</select>
```

```
<label>Role:</label>

<select onChange={handleRoleSelect}>
  <option value="">Select a Role</option>
  {predefinedRoles.map((role, index) => (
    <option key={index} value={role}>
      {role}
    </option>
  ))}
</select>
```

```
</div>
```

```
<div className="divlistcat">
  <div className="tools-list">
    {tools.map((tool, index) => (
      <span key={index} className="tool">
        {tool}
      <button
        type="button"
        className="remove-tool-btn"
```

```
      onClick={() => handleToolRemove(tool)}  
    >  
      ×  
    </button>  
  </span>  
  )}]
```

```
{roles.map((role, index) => (  
  <span key={index} className="role">  
    {role}  
    <button  
      type="button"  
      className="remove-tool-btn"  
      onClick={() => handleRoleRemove(role)}  
    >  
      ×  
    </button>  
  </span>  
  )}]  
</div>
```

```
</div>
```

```
</div>
```

```
</div>
```

```
<div className="rightprojup-container">
  <label>Thumbnail:</label>
  <div className="thumbnail-container">
    <input
      type="file"
      accept=".jpg,.jpeg,.png"
      ref={thumbnailInputRef}
      onChange={handleThumbnailChange}
    />
    {thumbnail && (
      <div className="thumbnail-preview">
        <img
          src={URL.createObjectURL(thumbnail)}
          alt="Thumbnail Preview"
          width={60}
          height={60}
        />
      </div>
    )}
  </div>
  <label>Attach Files:</label>
  <input
    type="file"
    multiple
    accept=".zip,.rar,.pdf,.docx,.jpg,.png"
```

```

        ref={fileInputRef}
        onChange={handleFileChange}
      />

      <label>Project URL:</label>

      <input
        className="projecturlinput"
        type="url"
        value={projectUrl}
        onChange={(e) => setProjectUrl(e.target.value)}
      />

    </div>

  </form>


  <div className="submit-btn-container">
    {selectedSubject && grade ? (
      <button type="submit" onClick={handleFinalSubmit} disabled={isSubmitting}>
        {isSubmitting ? "Submitting..." : "Final Submit"}
      </button>
    ) : (
      <button type="submit" onClick={handleSubmit}>
        Submit
      </button>
    )}
  </div>

</motion.div>

</div>

```



```
{showAssessmentModal && (  
  <ProjectAssessmentModal  
    show={showAssessmentModal}  
    onClose={() => setShowAssessmentModal(false)}  
    onAssessmentSubmit={handleAssessmentSubmit}  
    availableSubjects={availableSubjects} //  Pass subjects  
  />  
  )}  
</>  
);  
};
```

```
export default ProjectUploadModal;
```

```
//for push purposes
```

```
//my comment
```

### **ProjectAssessmentModal.jsx:**

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

import axiosInstance from "../../services/axiosInstance";

import "../ProjectAssessmentModal.css";

const ProjectAssessmentModal = ({ show, onClose, onAssessmentSubmit,
availableSubjects }) => {

  const [selectedSubject, setSelectedSubject] = useState("");
  const [grade, setGrade] = useState("");
  const [ratingSlip, setRatingSlip] = useState(null);
  const [selectedYear, setSelectedYear] = useState("");
  const [selectedTerm, setSelectedTerm] = useState("");
  const [existingGrade, setExistingGrade] = useState(null);
  const [isGradeDisabled, setIsGradeDisabled] = useState(false);

  const fetchExistingGrade = async (subjectCode, year, term) => {
    try {
      const response = await axiosInstance.get(
        `/api/checkExistingGrade?subject=${subjectCode}&year=${year}&term=${term}`
      );

      if (response.data.success && response.data.grade) {
        setExistingGrade(response.data.grade);
        setGrade(response.data.grade);
      }
    }
  }
}
```

```
    setIsGradeDisabled(true);
  } else {
    setExistingGrade(null);
    setGrade("");
    setIsGradeDisabled(false);
  }
} catch (error) {
  console.error("Error fetching existing grade:", error);
  setExistingGrade(null);
  setGrade("");
  setIsGradeDisabled(false);
}
};
```

```
const handleTermChange = async (e) => {
  const term = e.target.value;
  setSelectedTerm(term);

  if (selectedSubject && selectedYear && term) {
    await fetchExistingGrade(selectedSubject, selectedYear, term);
  }
};
```

```
const handleYearChange = async (e) => {
  const year = e.target.value;
  setSelectedYear(year);
};
```

```
if (selectedSubject && year && selectedTerm) {  
  await fetchExistingGrade(selectedSubject, year, selectedTerm);  
}  
};
```

```
const handleSubjectChange = async (e) => {  
  const subject = e.target.value;  
  setSelectedSubject(subject);  
  
  if (subject && selectedYear && selectedTerm) {  
    await fetchExistingGrade(subject, selectedYear, selectedTerm);  
  }  
};
```

```
const handleSubmit = async () => {  
  if (!selectedSubject || !grade.trim() || !selectedYear || !selectedTerm) {  
    alert("Please select a subject, grade, year level, and term.");  
    return;  
  }  
}
```

```
onAssessmentSubmit({  
  subjectCode: selectedSubject,  
  grade,  
  year: selectedYear,  
  term: selectedTerm,
```

```

        ratingSlip
    });
    onClose();
};

return (
    <div className="assessment-modal-overlay" onClick={onClose}>
        <div className="assessment-modal-content" onClick={(e) => e.stopPropagation()}>
            <h3>Project Subject Categorization</h3>
            <p>Please select a subject and input your grade:</p>

            <label>Subject:</label>
            {availableSubjects.length > 0 ? (
                <select
                    value={selectedSubject}
                    onChange={handleSubjectChange}
                    required
                >
                    <option value="">Select Subject</option>
                    {availableSubjects.map((subject) => (
                        <option key={subject.subjectCode} value={subject.subjectCode}>
                            {subject.subjectCode} - {subject.subjectName}
                        </option>
                    ))}
                </select>
            ) : (

```

<p>No subjects available for the selected tag.</p>

}}

<label>Year Level:</label>

<select

value={selectedYear}

onChange={handleYearChange}

>

<option value="">Select Year Level</option>

<option value="1st Year">1st Year</option>

<option value="2nd Year">2nd Year</option>

<option value="3rd Year">3rd Year</option>

<option value="4th Year">4th Year</option>

</select>

<label>Term:</label>

<select

value={selectedTerm}

onChange={handleTermChange}

>

<option value="">Select Term</option>

<option value="1st Semester Midterm">1st Semester Midterm</option>

<option value="1st Semester Finals">1st Semester Finals</option>

<option value="2nd Semester Midterm">2nd Semester Midterm</option>

<option value="2nd Semester Finals">2nd Semester Finals</option>

</select>

```

<label htmlFor="finalGrade">Final Grade:</label>

<select
  id="finalGrade"
  value={grade}
  onChange={(e) => setGrade(e.target.value)}
  disabled={isGradeDisabled}
>
  <option value="">Select Grade</option>
  <option value="1.0">1.0</option>
  <option value="1.25">1.25</option>
  <option value="1.5">1.5</option>
  <option value="1.75">1.75</option>
  <option value="2.0">2.0</option>
  <option value="2.25">2.25</option>
  <option value="2.5">2.5</option>
  <option value="2.75">2.75</option>
  <option value="3.0">3.0</option>
  <option value="4.0">4.0 (Conditional Failure)</option>
  <option value="5.0">5.0 (Failure)</option>
  <option value="INC">INC (Incomplete)</option>
  <option value="DRP">DRP (Dropped)</option>
</select>

{isGradeDisabled && (
  <p className="grade-info-message">

```

Grade is already recorded for this subject, year, and term.

</p>

}}

<label>Attach Rating Slip (if available):</label>

<input

type="file"

accept=".zip,.rar,.pdf,.docx,.jpg,.png"

onChange={(e) => setRatingSlip(e.target.files[0])}

/>

<div className="assessment-buttons">

<button onClick={handleSubmit} className="assessment-submit-btn">

Submit

</button>

</div>

</div>

</div>

);

};

export default ProjectAssessmentModal;