```
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 } = require("./models/Tupath_users");
const nodemailer = require("nodemailer");
const crypto = require("crypto");
require('dotenv').config()
const JWT_SECRET = "your-secret-key";
const GOOGLE_CLIENT_ID = "625352349873-
hrob3g09um6f92jscfb672fb87cn4kvv.apps.googleusercontent.com";

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

// Middleware setup
app.use(express.json());
app.use(cors({ origin: 'http://localhost:5173' })); // Updated CORS for specific origin
app.use('/uploads', express.static('uploads'));
app.use("/certificates", express.static(path.join(__dirname, "certificates")));
```

```
// 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+srv://admin123:admin123@cluster0.wfrb9.mongodb.net/tupath_users?retryWrites=true&w=majority",
  {
    useNewUrlParser: true,
  }
)
  .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
 });



// 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"],
    },
  });



  // 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" });
  }
});

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

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

    // Find the comment to delete
    const commentIndex = post.comments.findIndex(
      (comment) => comment._id.toString() === commentId && comment.userId === userId
    );

    if (commentIndex === -1) {
```

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

    // Remove the comment from the comments array
    post.comments.splice(commentIndex, 1);

    // 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 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 },
  votedUsers: [
   {
    userId: String,
    username: String,
    lastName: String,
   },
 ], // Array of users who upvoted
  comments: [
   {
    userId: String,
    profileImg: String,
    username: String,
    comment: String,
    createdAt: Date,
   },
  ],
 });

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

```
// Get all posts
app.get("/api/posts", async (req, res) => {
 try {
  const posts = await Post.find().sort({ timestamp: -1 });
  res.json(posts);
 } 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 { 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" });
  }
});

// Socket.IO events for real-time chat
io.on("connection", (socket) => {
  // console.log(`User connected: ${socket.id}`);

  socket.on("send_message", async (data) => {
    try {
     const message = new Message(data);
     await message.save();
     io.emit("receive_message", data);
    } catch (err) {
     console.error("Error saving message:", err);
    }
  });

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

// Login endpoint
```

```
 app.post("/login", async (req, res) => {
  const { email, password, role } = req.body;
  try {
   const user = role === "student" ? await Tupath_usersModel.findOne({ email }) : await
Employer_usersModel.findOne({ email });

   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
// 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
    const user = await UserModel.findOne({ email });
    if (!user) {
```

```
      return res.status(404).json({ success: false, message: 'User not registered. Please sign up first.' });
    }

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

    const redirectPath = role === 'student' ? '/homepage' : '/homepage';

    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');

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

    // Return profile details tailored to the role
    const profile = {
      email: user.email,
      role: user.role,
```

```
      profileDetails: user.profileDetails,
      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" });
  }
});

// api uploadproject endpoint
 app.post("/api/uploadProject", verifyToken, upload.fields([
  { name: "thumbnail", maxCount: 1 },  // Handle the thumbnail upload field
  { name: "projectFiles", maxCount: 5 }, // Handle other project files (e.g., .zip, .docx, etc.)
 ]), async (req, res) => {
  try {
    const userId = req.user.id;
    const { projectName, description, tags, tools, projectUrl} = req.body;

    // Get the file paths for both thumbnail and project files
    const thumbnailPath = req.files.thumbnail ?
` /uploads/${req.files.thumbnail[0].filename}` : null;
    const filePaths = req.files.projectFiles ? req.files.projectFiles.map(file =>
` /projects/${file.filename}` ) : [];
```

```
// Create the project object
const project = {
  projectName,
  description,
  tags: Array.isArray(tags) ? tags : tags.split(',').map(tag => tag.trim()),
  tools: Array.isArray(tools) ? tools : tools.split(',').map(tool => tool.trim()),
  files: filePaths,  // Array of other project files
  thumbnail: thumbnailPath,  // Path for the thumbnail image
  projectUrl,
  status: 'pending', // Default status
};

// Update the user profile with the new project
const updatedUser = await Tupath_usersModel.findByIdAndUpdate(
  userId,
  {
    $push: {
      "profileDetails.projects": project, // Push the new project to the projects array
    }
  },
  { new: true }
);

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

```
    }

    res.status(200).json({
      success: true,
      message: "Project uploaded successfully",
      project: project,
    });

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



app.get("/api/projects", verifyToken, async (req, res) => {
  try {
    const userId = req.user.id;
    const user = await Tupath_usersModel.findById(userId);
    if (!user) {
      return res.status(404).json({ success: false, message: "User not found" });
    }
    console.log("User projects:", user.profileDetails.projects);
    res.status(200).json({
      success: true,
```

```
      projects: user.profileDetails.projects.map(project => ({
        ...project.toObject(), // Ensure you get a plain object
        status: project.status || 'pending', // Add status if not already present
      })),
    });
  } catch (error) {
    console.error("Error fetching projects:", error);
    res.status(500).json({ success: false, message: "Internal server error" });
  }
});


app.put("/api/projects/:projectId/status", verifyToken, async (req, res) => {
  try {
    const userId = req.user.id;
    const { projectId } = req.params;
    const { status } = req.body; // Expect status (e.g., "pending", "submitted", etc.)

    // Validate status
    const validStatuses = ['pending', 'submitted', 'approved'];
    if (!validStatuses.includes(status)) {
      return res.status(400).json({ success: false, message: "Invalid status" });
    }

    // Find the user and project
    const user = await Tupath_usersModel.findById(userId);
```

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

    const project = user.profileDetails.projects.find(project => project._id.toString() === projectId);
    if (!project) {
      return res.status(404).json({ success: false, message: "Project not found" });
    }

    // Update project status
    project.status = status;
    await user.save();

    res.status(200).json({
      success: true,
      message: "Project status updated successfully",
      project: project,
    });
  } catch (error) {
    console.error("Error updating project status:", 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;
    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 and remove it from the user's profile
    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" });
    }

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

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

```
      message: "Project deleted successfully"
    });
  } 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, filter } = 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
    let results = [];

    if (filter === 'students') {
      const studentResults = await Tupath_usersModel.find({
        $or: [
          { 'profileDetails.firstName': regex },
          { 'profileDetails.middleName': regex },
```

```
        { 'profileDetails.lastName': regex }
      ]
    }).select('profileDetails.firstName profileDetails.middleName profileDetails.lastName profileDetails.profileImg');
    results = [...results, ...studentResults];
  }

  if (filter === 'employers') {
    const employerResults = await Employer_usersModel.find({
      $or: [
        { 'profileDetails.firstName': regex },
        { 'profileDetails.middleName': regex },
        { 'profileDetails.lastName': regex }
      ]
    }).select('profileDetails.firstName profileDetails.middleName profileDetails.lastName profileDetails.profileImg');
    results = [...results, ...employerResults];
  }

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

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

```
  const { id } = req.params;
  try {
    const user = await Tupath_usersModel.findById(id) || await
Employer_usersModel.findById(id);
    if (!user) {
      return res.status(404).json({ success: false, message: 'User not found' });
    }
    res.status(200).json({ success: true, profile: user });
  } catch (err) {
    console.error('Error fetching profile:', 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

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