**Project: Mystelio - Social Media Application**

**Introduction:**

Welcome to the kickoff of our Social Media Application project. This document outlines the key aspects of our project, including the chosen tech stack, modules, and initial tasks for Day 1.

**Tech Stack:**

* **Frontend:** React
* **Backend:** Express, Node.js
* **Database:** MySQL
* **Styling:** Custom CSS with Styled Components

**Modules:**

1. **User Authentication**
2. **Direct Messaging**
3. **User Profiles**
4. **Feed/Posts**
5. **Notifications**

**Database Schema:**

* Users Table
* Messages Table
* Posts Table
* Notifications Table

**Additional Points:**

* Utilize Styled Components for dynamic styling.
* Implement token-based authentication for security.
* Explore real-time communication libraries for instant messaging.

In today's tasks, we configured the database and authentication settings for the Mystelio application. Below are the key configurations made:

**Database Configuration:**

* **Host:** localhost
* **Username:** root
* **Password:** root
* **Database Name:** mystelio\_db

**JWT (JSON Web Token) Secret:**

* A secret key used for generating and verifying JWTs (JSON Web Tokens).
* Example JWT Secret: eyJhbGciOiJIUzI1NiJ9.eyJSb2xlIjoiQWRtaW4iLCJJc3N1ZXIiOiJJc3N1ZXIiLCJVc2VybmFtZSI6IkphdmFJblVzZSIsImV4cCI6MTY5OTc5MzI3OCwiaWF0IjoxNjk5NzkzMjc4fQ.FOiCPTKyS1cEED5\_xQwo6jP1J4xvf9mcaIGUz\_rrHuY

These configurations are crucial for the proper functioning of the Mystelio application, ensuring secure communication with the database and implementing user authentication using JWTs.

**ENV Config:**

DB\_HOST='localhost'

DB\_USER='root'

DB\_PASSWORD='root'

DB\_DATABASE='mystelio\_db'

JWT\_SECRET='eyJhbGciOiJIUzI1NiJ9.eyJSb2xlIjoiQWRtaW4iLCJJc3N1ZXIiOiJJc3N1ZXIiLCJVc2VybmFtZSI6IkphdmFJblVzZSIsImV4cCI6MTY5OTc5MzI3OCwiaWF0IjoxNjk5NzkzMjc4fQ.FOiCPTKyS1cEED5\_xQwo6jP1J4xvf9mcaIGUz\_rrHuY'

**Day 2 Tasks:**

1. Set up the project structure.
2. Initialize React frontend and Express backend.
3. Create basic UI components for login and signup.
4. Establish the MySQL database connection.

**Day 2: Logo Design**

**Logo Concept:**

Our application, Mystelio, needs a unique and representative logo that captures the essence of social interaction and mystery. The chosen color palette sets the tone for a vibrant and engaging visual identity.

**Color Palette:**

:root {

--thistle: #cdb4dbff;

--fairy-tale: #ffc8ddff;

--carnation-pink: #ffafccff;

--uranian-blue: #bde0feff;

--light-sky-blue: #a2d2ffff;

}

**Logo Elements:**



1. **Circle Background:**

.circle {

width: 120px;

height: 120px;

background-color: var(--uranian-blue);

border-radius: 50%;

display: flex;

align-items: center;

justify-content: center;

margin-right: 16px; /\* Adjust as needed \*/

}

**Logo Composition**

<div class="logo-container">

<div class="circle">

<span class="logo">M</span>

</div>

<div class="tagline">Mystelio - Connect in Style</div>

</div>

**Day 2: Frontend Setup and Basic Pages**

**Objective:**

1. Set up the frontend project.
2. Implement a responsive sidebar.
3. Design login and register pages.
4. Apply basic styling using the provided color palette.

**Tasks Completed:**

**1. Frontend Project Setup:**

* Initialized a React project using Create React App.
* Organized project structure with folders for components, assets, and styles.
* Integrated Font Awesome for icons.
* Set up basic CSS reset and color palette.

**2. Responsive Sidebar:**

* Created a responsive sidebar with a toggle button.
* Implemented a smooth transition for the sidebar.

**3. Login and Register Pages:**

* Designed login and register pages with basic form elements.
* Applied styling using the provided color palette.

**4. Styling:**

* Created a global CSS file with color variables and basic styles.
* Styled components using the color palette for a consistent look.

**Next Steps:**

1. **Authentication Flow:**
   * Implement authentication logic on the frontend.
   * Connect the frontend authentication with the backend.
2. **Direct Messaging Module:**
   * Design and implement the direct messaging module.
   * Connect frontend messaging with backend APIs.
3. **Feed Module (Twitter-like):**
   * Plan and design a feed module.
   * Implement basic feed functionalities.
4. **Documentation:**
   * Maintain documentation for code and project structure.
   * Update documentation regularly.

**Challenges Faced:**

* None so far.

**Day-03: Handling User Authentication in a Web Application**

**Introduction**

In today's session, we will focus on implementing user authentication in a web application using React on the frontend and Node.js with Express on the backend. Authentication is a crucial aspect of many applications to ensure that only authorized users can access certain features and data.

**Topics Covered**

1. **Backend Setup**
   * Initialize a Node.js project with Express.
   * Set up routes for user registration and login.
   * Use bcrypt to hash and secure user passwords.
   * Store user data in a database using Sequelize.
2. **Frontend Integration**
   * Create React components for user registration and login forms.
   * Use Axios to send HTTP requests from the frontend to the backend.
   * Implement form validation on the client-side.
3. **Token-Based Authentication**
   * Generate JWT (JSON Web Tokens) for authenticated users.
   * Secure routes on the backend that require authentication.
   * Store and manage JWT on the client-side.
4. **User Profile and Logout**
   * Display user information on the frontend after successful login.
   * Implement a logout feature to terminate the user session.

**Assignment**

Build a simple registration and login system for your web application. Use the concepts covered in today's session to secure user data and implement a smooth authentication flow.

**Storing Image and retrieving**

**SIGNUP.JS**

import React, { useState } from "react";

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

import axios from "./../UrlHelper";

import { toast } from "react-toastify";

*// Countries data in JSON format*

const countriesData = [

  { value: "IND", label: "India" },

  { value: "USA", label: "United States" },

  { value: "UK", label: "United Kingdom" },

  { value: "Germany", label: "Germany" },

  { value: "Japan", label: "Japan" },

];

const Signup = () => {

  const [formData, setFormData] = useState({

    fullName: "",

    phoneNumber: "",

    birthDate: "",

    password: "",

    email: "",

    country: "",

    city: "",

    profileImage: null,

  });

  const handleInputChange = (e) => {

    const { name, value } = e.target;

    setFormData((prevData) => ({ ...prevData, [name]: value }));

  };

  const handleImageChange = (e) => {

    const file = e.target.files[0];

    const reader = new FileReader();

    reader.onloadend = () => {

*// Convert the base64 data to a Uint8Array*

      const uint8Array = new Uint8Array(atob(reader.result.split(",")[1]).split("").map((char) => char.charCodeAt(0)));

*// Set the Uint8Array in the component's state*

      setFormData((prevData) => ({ ...prevData, profileImage: uint8Array }));

    };

    if (file) {

      reader.readAsDataURL(file);

    }

  };

  const handleFormSubmit = async (e) => {

    e.preventDefault();

    console.log(formData)

    try {

      const response = await axios.post("/auth/register", formData);

      toast.success(

        <>

          <i *className*="fa-solid fa-handshake"></i> {"Successfully Registered"}

        </>

      );

      console.log(response.data);

*// Clear form data after successful submission*

      setFormData({

        fullName: "",

        phoneNumber: "",

        birthDate: "",

        password: "",

        email: "",

        country: "",

        city: "",

      });

    } catch (error) {

*// Handle errors, you can console.log them for now*

      console.error("Registration Error:", error.message);

    }

  };

  return (

    <div *className*="login register">

      <div *className*="container">

        <div *className*="heading">

          <h1 *className*="title">

            <i *className*="fa-solid fa-user-plus"></i>&nbsp;Register to Mystelio

          </h1>

        </div>

        <form *className*="form" *onSubmit*={handleFormSubmit}>

          <div *className*="input-box">

            <label>

              <i *className*="fa-solid fa-signature"></i>&nbsp;Full Name

            </label>

            <input

*required*=""

*placeholder*="Enter full name"

*type*="text"

*name*="fullName"

*value*={formData.fullName}

*onChange*={handleInputChange}

            />

          </div>

          <div *className*="column">

            <div *className*="input-box">

              <label>

                <i *className*="fa-solid fa-phone"></i>&nbsp;Phone Number

              </label>

              <input

*required*=""

*placeholder*="Enter phone number"

*type*="telephone"

*name*="phoneNumber"

*value*={formData.phoneNumber}

*onChange*={handleInputChange}

              />

            </div>

            <div *className*="input-box">

              <label>

                <i *className*="fa-solid fa-envelope-open-text"></i>&nbsp;Enter

                Email

              </label>

              <input

*required*=""

*placeholder*="Enter Email address"

*type*="email"

*name*="email"

*value*={formData.email}

*onChange*={handleInputChange}

              />

            </div>

          </div>

          <div *className*="column">

            <div *className*="input-box">

              <label>

                <i *className*="fa-solid fa-cake-candles"></i>&nbsp;Birth Date

              </label>

              <input

*required*=""

*placeholder*="Enter birth date"

*type*="date"

*name*="birthDate"

*value*={formData.birthDate}

*onChange*={handleInputChange}

              />

            </div>

            <div *className*="input-box">

              <label>

                <i *className*="fa-solid fa-key"></i>&nbsp;Enter Password

              </label>

              <input

*required*=""

*placeholder*="Enter password"

*type*="password"

*name*="password"

*value*={formData.password}

*onChange*={handleInputChange}

              />

            </div>

          </div>

          <div *className*="input-box address">

            <label>

              <i *className*="fa-solid fa-folder-open"></i>&nbsp;Additional

              Details

            </label>

            <div *className*="column">

              <div *className*="select-box">

                <select

*name*="country"

*value*={formData.country}

*onChange*={handleInputChange}

                >

                  <option *value*="">Select</option>

                  {countriesData.map((country) => (

                    <option *key*={country.value} *value*={country.value}>

                      {country.label}

                    </option>

                  ))}

                </select>

              </div>

              <input

*required*=""

*placeholder*="Enter your city"

*type*="text"

*name*="city"

*value*={formData.city}

*onChange*={handleInputChange}

              />

            </div>

          </div>

          <div *className*="input-box">

            <label>

              <i *className*="fa-solid fa-file-image"></i>&nbsp;Profile Image

            </label>

            <input *type*="file" *accept*="image/\*" *onChange*={handleImageChange} />

          </div>

          <button *className*="login-button" *type*="submit">

            Submit

          </button>

          <span *className*="agreement">

            <Link *to*="/login">Already a user? Login Here!</Link>

          </span>

        </form>

      </div>

    </div>

  );

};

export default Signup;

**BACKEND AUTHURLS.JS:**

const bcrypt = require("bcrypt");

const Joi = require("joi");

const jwt = require("jsonwebtoken");

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

const multer = require('multer');

const express = require("express");

const router = express.Router();

require('dotenv').config();

*// Joi is a powerful validation library for JavaScript and Node.js. It's commonly used for:*

*// Input Validation: Joi helps you validate and sanitize user input to ensure that it meets the expected criteria. This is crucial for preventing security vulnerabilities such as SQL injection or other forms of injection attacks.*

*// Schema Validation: Joi allows you to define a schema that specifies the expected shape and types of data. This helps ensure that your data adheres to a predefined structure.*

*// Error Handling: Joi provides detailed error messages when validation fails, making it easier to identify and address issues. These error messages can be sent back to clients for informative feedback.*

const userSchema = Joi.object({

  fullName: Joi.string().required(),

  phoneNumber: Joi.string().required(),

  birthDate: Joi.date().required(),

  password: Joi.string().required(),

  email: Joi.string().email().required(),

  country: Joi.string(),

  city: Joi.string(),

  profileImage: Joi.allow(),

});

router.post('/register', async (req, res) => {

  try {

    const { error } = userSchema.validate(req.body);

    if (error) {

      console.log(error)

      return res.status(400).json({ error: error.details[0].message });

    }

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

*// Create a new user in the database*

    const imageBytes = Object.values(req.body.profileImage);

    const imageBuffer = Buffer.from(imageBytes);

    const newUser = await User.create({

      fullName : req.body.fullName,

      phoneNumber: req.body.phoneNumber,

      birthDate: req.body.birthDate,

      password: hashedPassword,

      email: req.body.email,

      country: req.body.country,

      city: req.body.city,

      profileImage: imageBuffer,

    });

    res.status(201).json({ message: 'Success' });

  } catch (error) {

    console.error('Error registering user:', error);

    res.status(500).send('Error registering user');

  }

});

*// Validation schema for login*

const loginSchema = Joi.object({

  email: Joi.string().email().required(),

  password: Joi.string().required(),

});

router.post("/login", async (req, res) => {

  try {

*// Validate input*

    const { error } = loginSchema.validate(req.body);

    if (error) {

      return res.status(400).json({ error: error.details[0].message });

    }

    const { email, password } = req.body;

*// Find user by email*

    const user = await User.findOne({ where: { email } });

    if (!user) {

      return res.status(404).json({ error: "User not found" });

    }

*// Compare passwords*

    const passwordMatch = await bcrypt.compare(password, user.password);

    if (!passwordMatch) {

      return res.status(401).json({ error: "Invalid password" });

    }

*// Generate JWT token*

    const token = jwt.sign(

      { userId: user.id, email: user.email },

      process.env.JWT\_SECRET,

      { expiresIn: "1h" } *// Token expires in 1 hour*

    );

*// Return JWT token and user data (excluding password)*

    res.status(200).json({

      token,

      user: {

        id: user.id,

        fullName: user.fullName,

        phoneNumber: user.phoneNumber,

        birthDate: user.birthDate,

        email: user.email,

        country: user.country,

        city: user.city,

        creted\_at: user.createdAt,

        updated\_at: user.updatedAt,

        profileImage: user.profileImage

      },

    });

  } catch (error) {

    console.error("Error during login:", error);

    res.status(500).send("Error during login");

  }

});

module.exports = router;

**SIGNIN Retrieving img:**

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

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

import axios from "./../UrlHelper";

import { toast } from "react-toastify";

const Login = () => {

  const [user, setUser] = useState(null);

  const [formData, setFormData] = useState({

    email: "",

    password: "",

  });

  const handleInputChange = (e) => {

    const { name, value } = e.target;

    setFormData((prevData) => ({ ...prevData, [name]: value }));

  };

  const handleFormSubmit = async (e) => {

    e.preventDefault();

    try {

      const response = await axios.post("/auth/login", formData);

*// Assume the server sends a JWT token in the response*

      const token = response.data.token;

      const userData = response.data.user;

      setUser(userData)

      handleImageFetch(userData);

*// Handle the token as needed (e.g., store it in local storage or a state variable)*

      toast.success("Login Successful");

      console.log("Login Successful:", response.data);

*// Clear form data after successful submission*

      setFormData({

        email: "",

        password: "",

      });

    } catch (error) {

*// Handle errors, you can console.log them for now*

      console.error("Login Error:", error.message);

      toast.error("Login Failed");

    }

  };

  const handleImageFetch = (userData) => {

    if (userData && userData.profileImage && userData.profileImage.data) {

      const bufferToBase64 = (buffer) => {

        const binary = buffer.reduce(

          (acc, byte) => acc + String.fromCharCode(byte),

          ""

        );

        return btoa(binary);

      };

      const base64Image = bufferToBase64(userData.profileImage.data);

      setUser((prevUser) => ({ ...prevUser, profileImage: base64Image }));

    }

  };

  return (

    <div *className*="login">

      <div *className*="container">

        <div *className*="heading">

          <h1 *className*="title">

            <i *className*="fa-solid fa-right-to-bracket"></i>&nbsp;LogIn to

            Mystelio

          </h1>

        </div>

        <form *className*="form" *onSubmit*={handleFormSubmit}>

          <div *className*="input-box">

            <label>

              <i *className*="fa-solid fa-envelope-open-text"></i>&nbsp;Enter Your

              email

            </label>

            <input

*required*=""

*placeholder*="example@mystelio.com"

*type*="email"

*name*="email"

*value*={formData.email}

*onChange*={handleInputChange}

            />

          </div>

          <div *className*="input-box">

            <label>

              <i *className*="fa-solid fa-unlock-keyhole"></i>&nbsp;Enter Password

            </label>

            <input

*required*=""

*placeholder*="\*\*\*\*\*\*\*\*\*\*"

*type*="password"

*name*="password"

*value*={formData.password}

*onChange*={handleInputChange}

            />

          </div>

          <span *className*="forgot-password">

            <Link *to*="/">Forgot Password ?</Link>

          </span>

          <input *value*="Sign In" *type*="submit" *className*="login-button" />

        </form>

        <span *className*="agreement">

          <Link *to*="/register">New to Mystelio? Signup Here!</Link>

        </span>

      </div>

      {user && user.profileImage && (

        <img *src*={`data:image/png;base64,${user.profileImage}`} *alt*="Profile" />

      )}

      {user && user.fullName}

    </div>

  );

};

export default Login;

**Day-04 Summary:**

1. **File Upload and Image Handling:**
   * Updated the backend to store image URLs instead of the actual image in the database.
   * Implemented image upload functionality for user profiles.
   * Saved images in the server's **uploads/profilepics/** directory.
2. **Frontend Integration:**
   * Modified the frontend registration form to handle file uploads and send the image URL to the backend.
   * Updated the login component to display user information, including the profile image fetched from the URL.
3. **User Authentication:**
   * Enhanced user registration route to store the image URL in the database.
   * Implemented user login functionality with JWT token generation.
4. **Error Handling:**
   * Implemented error handling for various scenarios during registration and login processes.
5. **Bug Fixes:**
   * Resolved issues related to file uploads, undefined file requests, and image display.
6. **Future Considerations:**
   * Discussed considerations for organizing uploads into different folders for various features like user profiles and posts.

**Updated backend:**

**userModel:**

const { DataTypes } = require("sequelize");

const db = require("../config/database");

const User = db.define("User", {

  fullName: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  phoneNumber: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  birthDate: {

    type: DataTypes.DATE,

    allowNull: false,

  },

  password: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  email: {

    type: DataTypes.STRING,

    allowNull: false,

    unique: true,

    validate: {

      isEmail: true,

    },

  },

  country: {

    type: DataTypes.STRING,

  },

  city: {

    type: DataTypes.STRING,

  },

  profileImagePath: {

    type: DataTypes.STRING, *// Store the path to the image file*

  },

});

module.exports = User;

**AuthUrls.js in backend:**

const bcrypt = require("bcrypt");

const Joi = require("joi");

const jwt = require("jsonwebtoken");

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

const multer = require("multer");

const express = require("express");

const router = express.Router();

const path = require("path");

require("dotenv").config();

*// Joi is a powerful validation library for JavaScript and Node.js. It's commonly used for:*

*// Input Validation: Joi helps you validate and sanitize user input to ensure that it meets the expected criteria. This is crucial for preventing security vulnerabilities such as SQL injection or other forms of injection attacks.*

*// Schema Validation: Joi allows you to define a schema that specifies the expected shape and types of data. This helps ensure that your data adheres to a predefined structure.*

*// Error Handling: Joi provides detailed error messages when validation fails, making it easier to identify and address issues. These error messages can be sent back to clients for informative feedback.*

const userSchema = Joi.object({

  fullName: Joi.string().required(),

  phoneNumber: Joi.string().required(),

  birthDate: Joi.date().required(),

  password: Joi.string().required(),

  email: Joi.string().email().required(),

  country: Joi.string(),

  city: Joi.string(),

  profileImage: Joi.allow(),

});

*// Set up storage for multer*

const profilePicsStorage  = multer.diskStorage({

  destination: (req, file, cb) => {

    cb(null, "uploads/profilepics/"); *// Upload files to the 'uploads' folder*

  },

  filename: (req, file, cb) => {

    const uniqueSuffix = Date.now() + "-" + Math.round(Math.random() \* 1e9);

    const ext = path.extname(file.originalname);

    cb(null, file.fieldname + "-" + uniqueSuffix + ext);

  },

});

const profilePicsUpload  = multer({ storage: profilePicsStorage });

router.post("/register", profilePicsUpload.single("profileImage"), async (req, res) => {

  try {

    const { error } = userSchema.validate(req.body);

    if (error) {

      return res.status(400).json({ error: error.details[0].message });

    }

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

*// Create a new user in the database*

    const newUser = await User.create({

      fullName: req.body.fullName,

      phoneNumber: req.body.phoneNumber,

      birthDate: req.body.birthDate,

      password: hashedPassword,

      email: req.body.email,

      country: req.body.country,

      city: req.body.city,

      profileImagePath: req.file ? req.file.path : null, *// Save the image path*

    });

    res.status(201).json({ message: "Success" });

  } catch (error) {

    console.error("Error registering user:", error);

    res.status(500).send("Error registering user");

  }

});

*// Validation schema for login*

const loginSchema = Joi.object({

  email: Joi.string().email().required(),

  password: Joi.string().required(),

});

router.post("/login", async (req, res) => {

  try {

*// Validate input*

    const { error } = loginSchema.validate(req.body);

    if (error) {

      return res.status(400).json({ error: error.details[0].message });

    }

    const { email, password } = req.body;

*// Find user by email*

    const user = await User.findOne({ where: { email } });

    if (!user) {

      return res.status(404).json({ error: "User not found" });

    }

*// Compare passwords*

    const passwordMatch = await bcrypt.compare(password, user.password);

    if (!passwordMatch) {

      return res.status(401).json({ error: "Invalid password" });

    }

*// Generate JWT token*

    const token = jwt.sign(

      { userId: user.id, email: user.email },

      process.env.JWT\_SECRET,

      { expiresIn: "1h" } *// Token expires in 1 hour*

    );

*// Return JWT token and user data (excluding password)*

    res.status(200).json({

      token,

      user: {

        id: user.id,

        fullName: user.fullName,

        phoneNumber: user.phoneNumber,

        birthDate: user.birthDate,

        email: user.email,

        country: user.country,

        city: user.city,

        creted\_at: user.createdAt,

        updated\_at: user.updatedAt,

        profileImage: user.profileImagePath,

      },

    });

  } catch (error) {

    console.error("Error during login:", error);

    res.status(500).send("Error during login");

  }

});

module.exports = router;

**To make uploads folder as static and can be accessible by anyone:**

In server.js:

*// Serve static files from the "uploads" directory*

app.use("/uploads", express.static(path.join(\_\_dirname, "uploads")));

**Frontend to add fetch img:**

  const handleImageFetch = (userData) => {

    if (userData && userData.profileImage) {

*// Assuming userData.profileImage contains the file path*

      const imageUrl = `http://localhost:5000/${userData.profileImage.replace(

        "\\",

        "/"

      )}`;

      console.log(imageUrl);

      setUser((prevUser) => ({ ...prevUser, profileImage: imageUrl }));

    }

  };

**Day-05 Summary:**

**Created AuthContext:**

import React, { useState, createContext, useContext, useEffect } from "react";

import Cookies from "universal-cookie";

const cookies = new Cookies();

const AuthContext = createContext(null);

export const AuthProvider = ({ children }) => {

  const [user, setUser] = useState(null);

  const login = (userData) => {

    cookies.set("user", userData, {

      path: "/",

    });

    setUser(userData);

  };

  const logout = () => {

    cookies.remove("user");

    setUser(null);

  };

  useEffect(() => {

    const storedUser = cookies.get("user");

    if (storedUser) {

      setUser(storedUser);

    }

  }, []);

  return (

    <AuthContext.Provider

*value*={{ user, login, logout }}

    >

      {children}

    </AuthContext.Provider>

  );

};

export const useAuth = () => {

  return useContext(AuthContext);

};

**Stored data in cookies. And now updated login code to make user login and store this**

const Login = () => {

  const auth = useAuth();

//previous code

  const handleUrl = (url) => {

    if (url) {

      const imageUrl = `http://localhost:5000/${url}`;

      return imageUrl.replace("\\", "/");

    }

  };

  const handleFormSubmit = async (e) => {

    e.preventDefault();

    try {

      const response = await axios.post("/auth/login", formData);

      const user = {

        id: response.data.id,

        fullName: response.data.fullName,

        phoneNumber: response.data.phoneNumber,

        birthDate: response.data.birthDate,

        email: response.data.email,

        country: response.data.country,

        city: response.data.city,

        creted\_at: response.data.creted\_at,

        updated\_at: response.data.updated\_at,

        profileImage: handleUrl(response.data.profileImage),

        token: response.data.token,

      };

      auth.login(user);

*// Handle the token as needed (e.g., store it in local storage or a state variable)*

      toast.success("Login Successful");

      setFormData({

        email: "",

        password: "",

      });

    } catch (error) {

*// Handle errors, you can console.log them for now*

      console.error("Login Error:", error.message);

      toast.error("Login Failed");

    }

  };

};

**Don’t forget to wrap index.js with Authcontext to implement:**

import React from "react";

import ReactDOM from "react-dom/client";

import App from "./App";

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

import { AuthProvider } from "./AuthContext";

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(

  <AuthProvider>

    <BrowserRouter>

      <App />

    </BrowserRouter>

  </AuthProvider>

);

**Day-06: Completed Tasks**

1. **Updated Login Route in Backend:**
   * Updated the login route in the backend to set a JWT token as a cookie with a max age of 90 days.
   * Used the **res.cookie** method to achieve this, setting options like **httpOnly**, **expires**, **sameSite**, and **secure**.

res.cookie("authToken", token, {

httpOnly: true,

expires: new Date(Date.now() + 90 \* 24 \* 60 \* 60 \* 1000),

sameSite: "all",

secure: false,

});

1. **User Model Modification:**
   * Modified the **User** model in the backend to include a **posts** array, which contains an array of posts.
   * Created another model named **Post** with necessary fields.

const { DataTypes } = require("sequelize");

const db = require("../config/database");

const User = db.define("User", {

  fullName: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  phoneNumber: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  birthDate: {

    type: DataTypes.DATE,

    allowNull: false,

  },

  password: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  email: {

    type: DataTypes.STRING,

    allowNull: false,

    unique: true,

    validate: {

      isEmail: true,

    },

  },

  country: {

    type: DataTypes.STRING,

  },

  city: {

    type: DataTypes.STRING,

  },

  profileImagePath: {

    type: DataTypes.STRING, *// Store the path to the image file*

  },

});

module.exports = User;

1. **Post Routes Setup:**
   * Set up routes for adding posts and retrieving all posts in a new file named **posturls.js**.
   * Encountered an error regarding a missing **authMiddleware** module, which was resolved by creating the missing module.
2. **Auth Middleware Update:**
   * Modified the **authMiddleware** to access the **authToken** from cookies instead of headers.
3. **Profile Image URL Handling:**
   * Updated the registration route to construct the complete URL for storing the profile image in the database.
   * Handled cases where **req.file** might be **null** to prevent exceptions.

*// Set up storage for multer*

const profilePicsStorage = multer.diskStorage({

  destination: (req, file, cb) => {

    cb(null, "uploads/profilepics/"); *// Upload files to the 'uploads' folder*

  },

  filename: (req, file, cb) => {

    const uniqueSuffix = Date.now() + "-" + Math.round(Math.random() \* 1e9);

    const ext = path.extname(file.originalname);

    cb(null, file.fieldname + "-" + uniqueSuffix + ext);

  },

});

const profilePicsUpload = multer({ storage: profilePicsStorage });

router.post(

  "/register",

  profilePicsUpload.single("profileImage"),

  async (req, res) => {

    try {

      const { error } = userSchema.validate(req.body);

      if (error) {

        return res.status(400).json({ error: error.details[0].message });

      }

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

      let profileImageUrl = null;

*// Check if req.file exists before constructing the complete URL*

      if (req.file) {

        profileImageUrl = `${req.protocol}://${req.get("host")}/${

          req.file.path

        }`;

      }

*// Create a new user in the database*

      const newUser = await User.create({

        fullName: req.body.fullName,

        phoneNumber: req.body.phoneNumber,

        birthDate: req.body.birthDate,

        password: hashedPassword,

        email: req.body.email,

        country: req.body.country,

        city: req.body.city,

        profileImagePath: profileImageUrl, *// Save the image path*

      });

      res.status(201).json({ message: "Success" });

    } catch (error) {

      console.error("Error registering user:", error);

      res.status(500).send("Error registering user");

    }

  }

);

1. **Debugging and Issue Resolution:**
   * Addressed issues related to undefined values and missing modules.
   * Debugged and resolved problems in the login and middleware functionality.
2. **Documentation and Notes:**
   * Provided explanations and documentation for various code snippets and changes.

**Challenges:**

* Encountered and resolved issues related to module dependencies and middleware functions.
* Ensured proper handling of null values when constructing profile image URLs.

**Upcoming Tasks:**

* Continue working on post-related functionalities.
* Implement frontend components for post creation and display.

Overall, progress was made in enhancing authentication, user models, and post-related functionalities. Debugging and issue resolution were crucial parts of today's tasks.

**Day-07:**

1. **Sequelize Associations:**
   * Worked on defining associations between Sequelize models, including one-to-many and many-to-one relationships.
   * Explored the **belongsTo** and **hasMany** associations to establish connections between User and Post models.
2. **Circular Dependency Handling:**
   * Addressed circular dependency issues when models had dependencies on each other.
   * Implemented a modular approach to associate models after all of them are defined to prevent circular dependency conflicts.
   * To overcome write relations in single file:

const { DataTypes } = require("sequelize");

const db = require("../config/database");

const Post = require("./postModel");

const User = db.define("User", {

  fullName: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  phoneNumber: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  birthDate: {

    type: DataTypes.DATE,

    allowNull: false,

  },

  password: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  email: {

    type: DataTypes.STRING,

    allowNull: false,

    unique: true,

    validate: {

      isEmail: true,

    },

  },

  country: {

    type: DataTypes.STRING,

  },

  city: {

    type: DataTypes.STRING,

  },

  profileImagePath: {

    type: DataTypes.STRING, *// Store the path to the image file*

  },

});

*// userModel.js*

User.hasMany(Post, { foreignKey: "userId", as: "posts" });

Post.belongsTo(User, { foreignKey: "userId", as: "user" });

module.exports = User;

**Post model:**

const { DataTypes } = require("sequelize");

const db = require("../config/database");

const User = require("./userModel");

const Post = db.define("Post", {

  title: {

    type: DataTypes.STRING,

    allowNull: false,

  },

  content: {

    type: DataTypes.TEXT,

    allowNull: false,

  },

  postImagePath: {

    type: DataTypes.STRING, *// Store the path to the image file*

  },

});

module.exports = Post;

**NOTE: when I wrote associations in two files I got circular dependency error so I wrote associations in one single file…**

1. **User and Post Models:**
   * Revised and optimized the User and Post model definitions to ensure proper associations.
   * Resolved errors related to Sequelize associations and model instantiation.
2. **API Routes:**
   * Developed API routes for creating and retrieving posts.
   * Integrated authentication middleware to handle user authentication before allowing post-related actions.
3. **Error Handling:**
   * Implemented error handling mechanisms for better responses in case of validation errors, database queries, and association issues.
4. **Postman Testing:**
   * Conducted testing using Postman to verify the functionality of user login, post creation, and post retrieval.
   * Debugged and resolved issues related to cookies, JWT tokens, and data retrieval.
5. **Multer Configuration:**
   * Configured Multer for handling file uploads, specifically for user profile pictures.
   * Defined storage options and filename generation for saving uploaded profile pictures.
6. **Cookie Configuration:**
   * Configured cookies for authentication tokens with specific options such as HTTP-only, max age, and secure settings.
7. **Next Steps:**
   * Plan for implementing user retrieval with associated posts.
   * Enhance error handling and validation for various API routes.
   * Consider additional features such as post updating and deletion.

**If user deleted then his posts get deleted use onDelete**

User.hasMany(Post, { foreignKey: "userId", as: "posts"});

*// onDelete cascade makes if user deletes his posts also gets deleted!*

*// User.hasMany(Post, { foreignKey: "userId", as: "posts", onDelete: "CASCADE"});*

Post.belongsTo(User, { foreignKey: "userId", as: "user" });

**LIKES MODULE:**

It seems like you’re trying to save the likes array to the database, but it’s not being saved. This could be due to a few reasons:

1. **Database Schema**: Ensure that your Post model schema in your database is set up to accept and store an array of user IDs in the likes field.
2. **Sequelize Array Handling**: If you’re using Sequelize (which it looks like you might be, given the findByPk method), handling arrays can be tricky. Sequelize doesn’t directly support arrays for all databases. If your database does support arrays and it’s not working, you might need to handle the addition operation differently. Instead of post.likes.push(userId), you might need to use something like post.likes = [...post.likes, userId].
3. **Database Syncing**: Ensure that after making changes to your database schema, you’ve synced the changes correctly.

Here’s how you might modify your code:

**JavaScript**AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

exports.likePost = async (req, res) => {

try {

const postId = req.params.postId;

const userId = req.user.id;

const post = await Post.findByPk(postId);

if (!post) {

return res.status(400).json({ message: "Post not found" });

}

// Check if the user has already liked the post

const alreadyLiked = post.likes.includes(userId);

if (alreadyLiked) {

return res.status(400).json({ message: "Post already liked" });

}

// Add the user ID to the likes array

post.likes = [...post.likes, userId]; // Change this line

// Save the updated post

try {

await post.save();

console.log("Post saved successfully");

res.status(200).json({ message: "Post liked successfully", post: post });

} catch (error) {

console.error("Error saving post:", error);

res

.status(500)

.json({ message: "Error saving post", error: error.message });

}

} catch (error) {

console.error("Error liking post:", error);

res

.status(500)

.json({ message: "Error liking post", error: error.message });

}

};

1. **Associations in Sequelize:**
   * Defined associations between **User** and **Post** models.
   * Ensured proper setup of **hasMany** and **belongsTo** associations.
   * Resolved issues related to the association setup.
2. **Cascade Deletion:**
   * Implemented cascade deletion of posts when a user is deleted.
   * Used **onDelete: "CASCADE"** in the association definition.
3. **Like Functionality:**
   * Added a **likes** field to the **Post** model to store user IDs who liked the post.
   * Implemented a route and controller to handle post likes.
   * Handled scenarios such as checking if a user has already liked a post.
4. **Postman Testing:**
   * Tested the like functionality using Postman.
   * Observed and resolved issues related to saving likes to the database.
5. **Sequelize Model Definitions:**
   * Made adjustments to Sequelize model definitions for **Post** and **User**.
   * Ensured correct usage of **Model** and associations.

**Unlike Post:**

exports.unlikePost = async (req, res) => {

  try {

    const postId = req.params.postId;

    const userId = req.user.id;

    const post = await Post.findByPk(postId);

    if (!post) {

      return res.status(400).json({ message: "Post not found" });

    }

*// Check if the user has already liked the post*

    const likedIndex = post.likes.indexOf(userId);

    if (likedIndex === -1) {

      return res.status(400).json({ message: "Post not liked yet" });

    }

*// Remove the user ID from the likes array*

    post.likes = post.likes.filter(id => id !== userId); *// Change this line*

*// Save the updated post*

    try {

      await post.save();

      console.log("Post saved successfully");

      res.status(200).json({ message: "Post unliked successfully", post: post });

    } catch (error) {

      console.error("Error saving post:", error);

      res

        .status(500)

        .json({ message: "Error saving post", error: error.message });

    }

  } catch (error) {

    console.error("Error unliking post:", error);

    res

      .status(500)

      .json({ message: "Error unliking post", error: error.message });

  }

};

**Fetching posts with post details, created\_user and the details of user who liked post:**

exports.getPosts = async (req, res) => {

  try {

    const allPosts = await Post.findAll({

      attributes: ["id", "title", "content", "createdAt", "likes"],

      include: [

        {

          model: User,

          as: "user",

          attributes: ["id", "fullName", "email"],

        },

      ],

    });

    const postsWithLikes = await Promise.all(allPosts.map(async (post) => {

      const likesWithUserInfo = await Promise.all((post.likes || []).map(async (userId) => {

        const likedUser = await User.findByPk(userId, {

          attributes: ["id", "fullName", "email"],

        });

        return likedUser;

      }));

      return {

        id: post.id,

        title: post.title,

        content: post.content,

        createdAt: post.createdAt,

        likes: likesWithUserInfo,

        created\_user: post.user

      };

    }));

    res.status(200).json({ posts: postsWithLikes });

  } catch (error) {

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

    res.status(500).json({ message: "Error fetching posts", error: error.message });

  }

};

**Day 08: Summary - Followers and Related Tasks**

**1. Database Model Update:**

* Added **followers** and **following** fields of type JSON to the **User** model.

**2. Follow and Unfollow Routes:**

* Implemented routes for following and unfollowing users.
* Connected these routes to corresponding controller functions.

// Routes

router.post("/follow/:userId", authMiddleware, follow);

router.post("/unfollow/:userId", authMiddleware, unFollow);

// Controller Functions (Incomplete)

exports.follow = async (req, res) => { /\* Logic for following \*/ }

exports.unFollow = async (req, res) => { /\* Logic for unfollowing \*/ }

**3. Get Following and Followers Lists:**

* Initiated routes to fetch the list of users a person is following and their followers.

Refer follow.js in controller

// Routes (Incomplete)

router.get("/following/:userId", authMiddleware, getFollowingList);

router.get("/followers/:userId", authMiddleware, getFollowersList);

// Controller Functions (Incomplete)

exports.getFollowingList = async (req, res) => { /\* Logic to fetch following list \*/ }

exports.getFollowersList = async (req, res) => { /\* Logic to fetch followers list \*/ }

**4. Friend's Posts Update:**

* Modified the route to get posts from friends and the logged-in user, now considering the following list.

// Updated Route

router.get("/getFriendsPosts", authMiddleware, getFriendsPosts);

// Updated Controller Function (Incomplete)

exports.getFriendsPosts = async (req, res) => { /\* Updated logic for fetching friend's posts \*/ }

**Comment and replies:**

const { DataTypes } = require("sequelize");

const db = require("../config/database");

const Comment = db.define("Comment", {

  comment: {

    type: DataTypes.TEXT,

    allowNull: false,

  },

  replies: {

    type: DataTypes.JSON,

    defaultValue: [],

  },

});

module.exports = Comment;

exports.addComment = async (req, res) => {

  try {

    const { comment, replies } = req.body;

    const userId = req.user.id;

    const postId = req.params.postId;

*// Create a new comment*

    const newComment = await Comment.create({

      comment,

      replies,

      userId,

      postId,

    });

    res

      .status(201)

      .json({ message: "Comment added successfully", comment: newComment });

  } catch (error) {

    console.error("Error adding comment:", error);

    res

      .status(500)

      .json({ message: "Error adding comment", error: error.message });

  }

};

*// Add a reply to a comment*

exports.addReply = async (req, res) => {

  try {

    const commentId = req.params.commentId;

    const userId = req.user.id;

    const { reply } = req.body;

*// Find the comment*

    const comment = await Comment.findByPk(commentId);

    if (!comment) {

      return res.status(404).json({ message: "Comment not found" });

    }

*// Fetch user information for the reply*

    const user = await User.findByPk(userId, {

      attributes: ["id", "fullName", "profileImagePath"],

    });

    const newReply = {

      user,

      reply,

      createdAt: new Date(),

    };

*// Add the reply*

    comment.replies = [...(comment.replies || []), newReply];

*// Save the updated comment*

    await comment.save();

    res

      .status(201)

      .json({ message: "Reply added successfully", reply: newReply });

  } catch (error) {

    console.error("Error adding reply:", error);

    res

      .status(500)

      .json({ message: "Error adding reply", error: error.message });

  }

};

**Added comments and all updated the posts routes to fetch comments and replies too:**

**Example:**

const fetchPostsWithInfo = async (posts) => {

  return Promise.all(

    posts.map(async (post) => {

      const likesWithUserInfo = await fetchLikesWithUserInfo(post.likes);

      return {

        id: post.id,

        title: post.title,

        content: post.content,

        createdAt: post.createdAt,

        likes: likesWithUserInfo,

        created\_user: post.user,

        comments: post.comments,

      };

    })

  );

};

*// Function to fetch likes with user information*

const fetchLikesWithUserInfo = async (likes) => {

  return Promise.all(

    (likes || []).map(async (userId) => {

      const likedUser = await User.findByPk(userId, {

        attributes: ["id", "fullName", "email", "profileImagePath"],

      });

      return likedUser;

    })

  );

};

*// Fetch all posts with user and likes information*

exports.getPosts = async (req, res) => {

  try {

    const allPosts = await Post.findAll({

      include: [

        {

          model: User,

          as: "user",

          attributes: ["id", "fullName", "email", "profileImagePath"],

        },

        {

          model: Comment,

          as: "comments",

          attributes: ["id", "comment", "replies", "userId"],

          include: [

            {

              model: User,

              as: "user", *// Match the alias used in the association*

              attributes: ["id", "fullName", "profileImagePath"],

            },

          ],

        },

      ],

    });

    const postsWithLikes = await fetchPostsWithInfo(allPosts);

    res.status(200).json({ posts: postsWithLikes.reverse() });

  } catch (error) {

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

    res

      .status(500)

      .json({ message: "Error fetching posts", error: error.message });

  }

};