Final Project: Social Media Dashboard

Estimated time needed: 3 hours

In this hands-on learning experience, you will delve into the world of web development and build a robust Social Media Dashboard using Node.js, Express, and MongoDB. This project will guide you through essential concepts such as user authentication, API development, middleware usage, and error handling. This will provide you with a practical understanding of building modern web applications.

Learning Objectives

After completing this lab you will be able to:

1. Create Users and Posts collections in MongoDB.
2. Secure JWT and middleware function for authentication.
3. Implement API endpoints for user management and social media posts.
4. Develop the frontend component for a social media dashboard.
5. Deploy the Social Media Dashboard app on Docker.
6. Work the Social Media Dashboard app.

Note: *Please make sure to copy and save the output of all the `curl` commands used in Exercise 3, and download the `app.js` file once the project is complete. These will be needed to answer the questions in the assignment following the project*

About Skills Network Cloud IDE

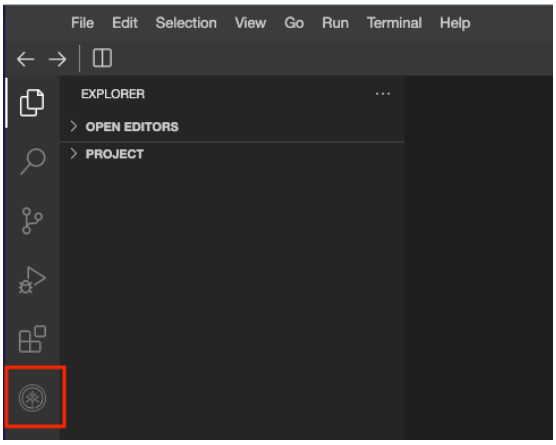
Skills Network Cloud IDE (based on Theia and Docker) provides an environment for hands on labs for course and project related labs. Theia is an open source IDE (Integrated Development Environment).

Important Notice about this Lab Environment

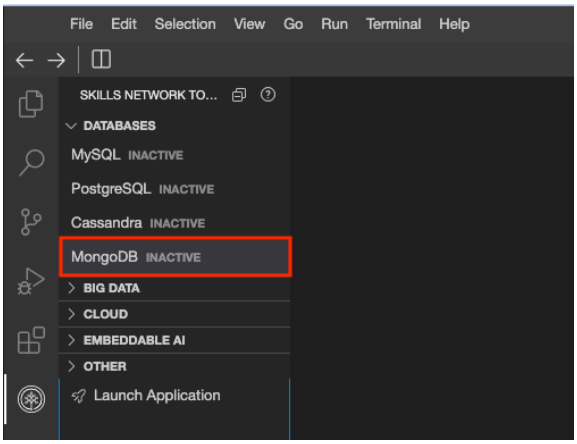
Please be aware that sessions for this lab environment are not persisted. Every time you connect to this lab, a new environment is created for you. Any data you may have saved in the earlier session would get lost. Plan to complete these labs in a single session, to avoid losing your data.

Exercise 1 - Start MongoDB Server

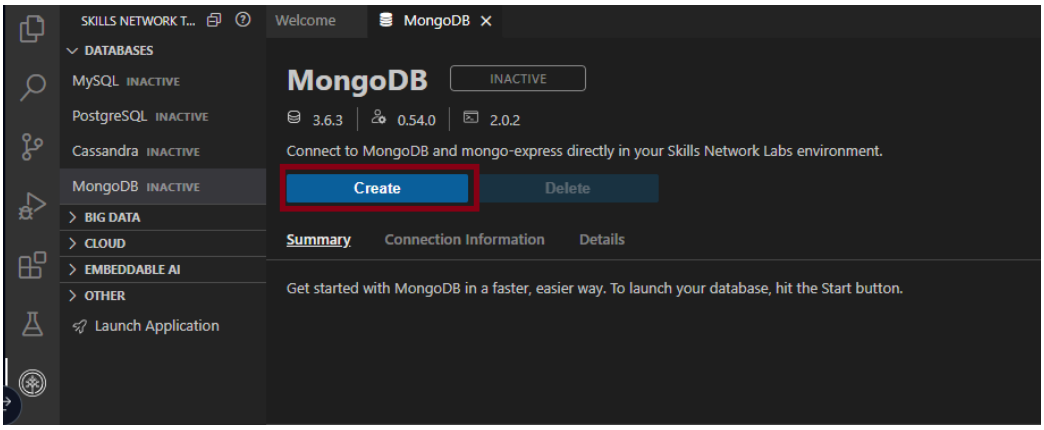
1. Click on the SkillsNetwork Toolbox Icon.



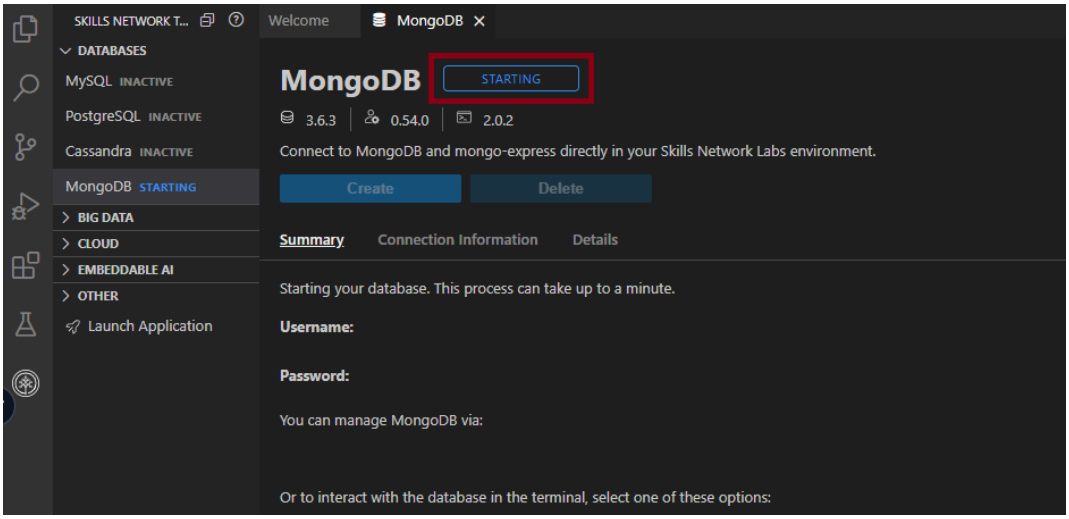
2. You will notice MongoDB listed, under `DATABASES`, but inactive. Which means the database is not available to use.



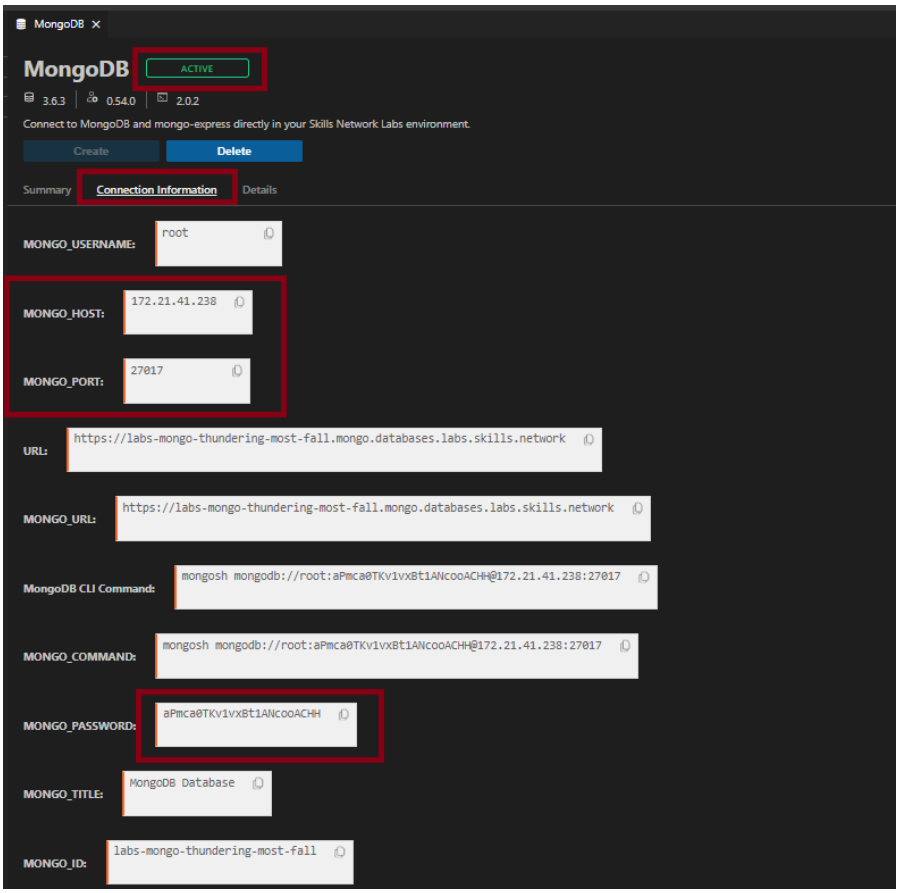
3. Once you click on it, you will see more details and a button to start.



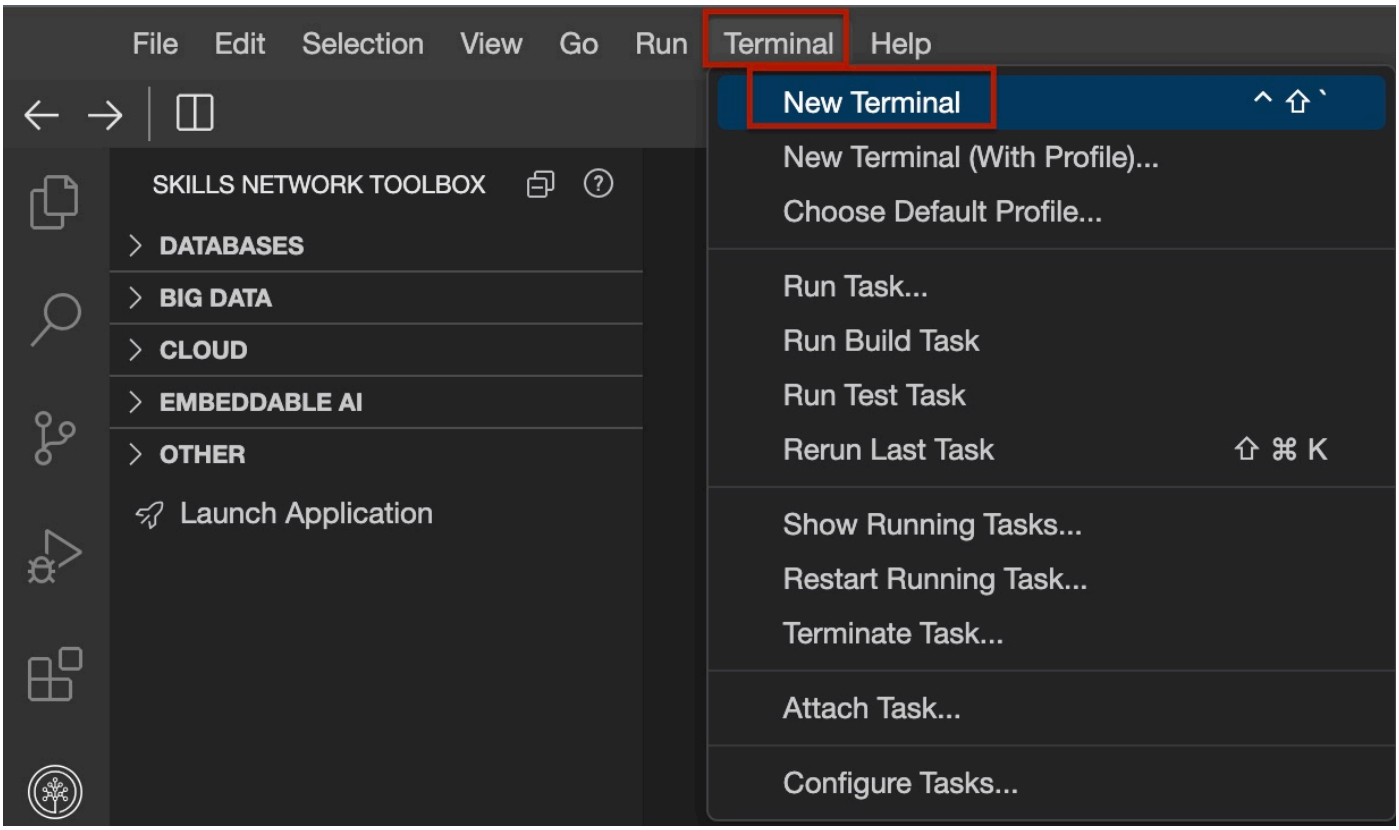
4. Clicking on the start button will run a background process to configure and start your MongoDB server.



5. Shortly after that, your server is ready for use. This deployment has access control enabled and MongoDB enforces authentication. So, Copy the `MONGO_HOST`, `MONGO_PORT`, `MONGO_PASSWORD` which is generated under `connection information` tab. You will need this information in the next step of this lab. Since the lab environment is temporary, the password and Host Id might be different the next time you start MongoDB database service.



Exercise 2 - Secure JWT and Middleware Function for Authentication



1. In the terminal, clone the git repository which has the starter code.

```
git clone https://github.com/ibm-developer-skills-network/rofgy-backend-nodejs-finalproject.git
```

2. Change to the new directory that you just cloned.

```
cd rofgy-backend-nodejs-finalproject
```

3. To set up the necessary Node.js packages, execute the following command in the terminal. This will install the required packages specified in the `packages.json` file, including `mongoose` for connecting to MongoDB server, `express` for creating API endpoints for user registration and login, and `body-parser` for parsing JSON input sent to the API endpoints.

```
npm install
```

4. Launch your editor and examine the contents of the `app.js` file. Replace the `MongoDB password` & `MongoDB host` with the password you copied in the previous exercise.

5. Paste the below code for `authenticateJWT Function` to authenticate users based on a JSON Web Token (JWT) stored in the session. You will write code create this token when the user logs in.

```
function authenticateJWT(req, res, next) {
  // Get token from session
  const token = req.session.token;
  // If no token, return 401 Unauthorized
  if (!token) return res.status(401).json({ message: 'Unauthorized' });
  try {
    // Verify token
    const decoded = jwt.verify(token, SECRET_KEY);

    // Attach user data to request
    req.user = decoded;

    // Continue to the next middleware
    next();
  } catch (error) {
```

```
    // If invalid token, return 401
    return res.status(401).json({ message: 'Invalid token' });
  }
}
```

This middleware checks for a JWT in the session of an incoming request.

- If the token is present, it is verified using a secret key (`SECRET_KEY`).
- If verification is successful, the decoded user information is attached to `req.user`, and control is passed to the next middleware.
- If there's no token or the verification fails, it sends a `401 Unauthorized` response.

6. Paste the below code for `requireAuth Function` to check whether a user is authenticated. If not, it redirects them to the login page.

```
function requireAuth(req, res, next) {
  const token = req.session.token;  // Retrieve token from session
  if (!token) return res.redirect('/login');  // If no token, redirect to login page
  try {
    const decoded = jwt.verify(token, SECRET_KEY);  // Verify the token using the secret key
    req.user = decoded;  // Attach decoded user data to the request
    next();  // Pass control to the next middleware/route
  } catch (error) {
    return res.redirect('/login');  // If token is invalid, redirect to login page
  }
}
```

This middleware checks for a JWT in the session of an incoming request:

- If the token is present, it is verified using a secret key (`SECRET_KEY`).
- If verification succeeds, the decoded user information is attached to `req.user`, and control is passed to the next middleware.
- If there's no token or the verification fails, it redirects the user to the `/login` page.

Exercise 3 - Implement API Endpoints for User Management and Social Media Posts

1. Insert the following code into the designated placeholder in the app.js file for `routing HTML` files.

```
app.get('/', (req, res) => res.sendFile(path.join(__dirname, 'public', 'index.html')));
app.get('/register', (req, res) => res.sendFile(path.join(__dirname, 'public', 'register.html')));
app.get('/login', (req, res) => res.sendFile(path.join(__dirname, 'public', 'login.html')));
app.get('/post', requireAuth, (req, res) => res.sendFile(path.join(__dirname, 'public', 'post.html')));
app.get('/index', requireAuth, (req, res) => res.sendFile(path.join(__dirname, 'public', 'index.html'), { username: req.user.username }));
```

Routing HTML Files

- **GET `/`**: Sends `index.html`.
- **GET `/register`**: Sends `register.html`.
- **GET `/login`**: Sends `login.html`.
- **GET `/post`**: Sends `post.html` (if authenticated).
- **GET `/index`**: Sends `index.html` with username (if authenticated).

These routes serve the corresponding HTML files based on the URL and user authentication.

2. Place the provided code into the assigned location in the app.js file to enable `user registration` with applied JWT authentication.

```
app.post('/register', async (req, res) => {
  const { username, email, password } = req.body;
  try {
    // Check if the user already exists
    const existingUser = await User.findOne({ $or: [{ username }, { email }] });
    if (existingUser) return res.status(400).json({ message: 'User already exists' });
    // Create and save the new user
    const newUser = new User({ username, email, password });
    await newUser.save();
    // Generate JWT token and store in session
    const token = jwt.sign({ userId: newUser._id, username: newUser.username }, SECRET_KEY, { expiresIn: '1h' });
    req.session.token = token;
    // Respond with success message
    res.send({"message":`The user ${username} has been added`});
  } catch (error) {
    console.error(error);
    // Handle server errors
    res.status(500).json({ message: 'Internal Server Error' });
  }
});
```

User Registration Implementation

- **POST `/register` Endpoint:**
  - Handles **user registration** by accepting `username`, `email`, and `password` from the request body.
  - **Checks for existing users** in the database by matching either the `username` or `email`. If a user already exists, it responds with a **400 error** and the message: "User already exists".
  - If the user does not exist:
    - **Creates a new user** in the database with the provided details (`username`, `email`, and `password`).
    - **Generates a JWT token** using `jwt.sign()`, which includes the `userId` and `username` with an expiration time of 1 hour.
    - **Stores the token** in the session (`req.session.token`).
    - Responds with a **success message** indicating the user has been added.
- **Error Handling:**
  - If any error occurs during the process, it responds with a **500 error** and the message: "Internal Server Error".

3. Insert the given code into the specified area in the app.js file to implement `user login` with applied JWT authentication.

```
app.post('/login', async (req, res) => {
  const { username, password } = req.body;
  try {
    // Check if the user exists with the provided credentials
    const user = await User.findOne({ username, password });
    if (!user) return res.status(401).json({ message: 'Invalid credentials' });
    // Generate JWT token and store in session
    const token = jwt.sign({ userId: user._id, username: user.username }, SECRET_KEY, { expiresIn: '1h' });
    req.session.token = token;
    // Respond with a success message
    res.send({"message":`${user.username} has logged in`});
  } catch (error) {
    console.error(error);
    // Handle server errors
    res.status(500).json({ message: 'Internal Server Error' });
  }
});
```

User Login Implementation

- **POST `/login` Endpoint:**
  - Handles **login** requests by accepting the `username` and `password` from the request body.
  - **Validates** the credentials by checking if a user with the provided `username` and `password` exists in the database.
  - If no matching user is found, it responds with a **401 error** and the message: "Invalid credentials".
  - If credentials are valid:
    - **Generates** a **JWT token** using `jwt.sign()`, which includes the `userId` and `username` with an expiration time of 1 hour.
    - **Stores** the token in the session (`req.session.token`).
    - Responds with a **success message** indicating the user has logged in.
- **Error Handling:**
  - If any error occurs during the process, it responds with a **500 error** and a message: "Internal Server Error".

4. Integrate the supplied code into the specified section of the app.js file to implement post creation and retrieves posts from the `Post` model in the database with JWT authentication.

```
app.post('/post', authenticateJWT, async (req, res) => {
    const { text } = req.body;
    // Validate post content
    if (!text || typeof text !== 'string') {
        return res.status(400).json({ message: 'Please provide valid post content' });
    }
    try {
        // Create and save new post with userId
        const newPost = new Post({ userId: req.user.userId, text });
        await newPost.save();
        res.status(201).json({ message: 'Post created successfully', post: newPost });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
    }
});
// Get all posts for the authenticated user
app.get('/posts', authenticateJWT, async (req, res) => {
    try {
        // Fetch posts for the logged-in user
        const posts = await Post.find({ userId: req.user.userId });
        res.json({ posts });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
    }
});
```

Post Creation and Retrieval Explanation

- **POST `/posts` Endpoint:**
  - Requires **authentication** via JWT (`authenticateJWT` middleware).
  - Validates the `text` content of the post (ensuring it's a valid string).
  - If validation fails, responds with a **400 error** and message: "Please provide valid post content".
  - If valid, creates a new post using the `Post` model, associating it with the authenticated user's `userId`.
  - Saves the new post to the database and responds with **201 success** and the created post.
- **GET `/posts` Endpoint:**

- Requires **authentication** via JWT (`authenticateJWT` middleware).
- Retrieves all posts associated with the authenticated user (`userId`).
- Responds with a JSON object containing all posts for the user.
- If there's an error during retrieval, responds with **500 error** and message: "Internal Server Error".

5. Insert the provided code into the assigned section of the app.js file to enable `post updation` with implemented JWT authentication.

```
app.put('/posts/:postId', authenticateJWT, async (req, res) => {
    const postId = req.params.postId;
    const { text } = req.body;
    try {
        // Find and update the post, ensuring it's owned by the authenticated user
        const post = await Post.findOneAndUpdate(
            { _id: postId, userId: req.user.userId },
            { text },
            { new: true } // Return updated post
        );
        // Return error if post not found
        if (!post) return res.status(404).json({ message: 'Post not found' });
        res.json({ message: 'Post updated successfully', updatedPost: post });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
    }
});
```

Update Post Explanation

- Handles **PUT** requests to the `/posts/:postId` endpoint, requiring authentication (`authenticateJWT` middleware).
- Retrieves the `postId` from the request parameters (`req.params.postId`) and the new `text` content from the request body (`req.body.text`).
- Finds and updates the post in the database based on `postId` and `userId` (authenticated user).
- Returns a success message along with the updated post if found and updated.
- If the post is not found, it returns a **404 error** with the message: "Post not found".
- In case of an internal error, it returns a **500 error** with the message: "Internal Server Error".

6. Place the provided code into the designated section of the app.js file to implement `post deletion` with applied JWT authentication.

```
app.delete('/posts/:postId', authenticateJWT, async (req, res) => {
    const postId = req.params.postId;
    try {
        // Find and delete the post, ensuring it's owned by the authenticated user
        const post = await Post.findOneAndDelete({ _id: postId, userId: req.user.userId });
        // Return error if post not found
        if (!post) return res.status(404).json({ message: 'Post not found' });
        res.json({ message: 'Post deleted successfully', deletedPost: post });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
    }
});
```

Delete Post Explanation

- Handles **DELETE** requests to the `/posts/:postId` endpoint, requiring authentication (`authenticateJWT` middleware).
- Retrieves the `postId` from the request parameters (`req.params.postId`).
- Finds and deletes the post from the database based on `postId` and `userId` (authenticated user).
- Returns a success message along with the deleted post if found.
- If the post is not found, it returns a **404 error** with a message: "Post not found".
- In case of an internal error, it returns a **500 error** with a message: "Internal Server Error".

7. Insert the provided code into the specified portion of the app.js file to implement `user logout` with applied JWT authentication.

```
app.get('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) console.error(err); // Log any session destruction errors
    res.redirect('/login'); // Redirect to login page after logout
  });
});
```

Logout Explanation

- Handles **GET** requests to the `/logout` endpoint.
- Destroys the user session with `req.session.destroy()`.
- If successful, redirects the user to the `/login` page.

Result:

- User is logged out and redirected to the login page.

Complete Solution of app.js

▼ Click here for the full solution

```
const express = require('express');
const jwt = require('jsonwebtoken');
const session = require('express-session');
const path = require('path');
const mongoose = require('mongoose');
const app = express();
const PORT = process.env.PORT || 3000;
const SECRET_KEY = 'your_secret_key';
mongoose.set('strictQuery', false);
const uri =  "mongodb://root:<replace passwaord>@<replace mongo host>:27017";
mongoose.connect(uri,{'dbName':'SocialDB'});
const User = mongoose.model('User', { username: String, email: String, password: String });
const Post = mongoose.model('Post', { userId: mongoose.Schema.Types.ObjectId, text: String });
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(session({ secret: SECRET_KEY, resave: false, saveUninitialized: true, cookie: { secure: false } }));
// Insert your authenticateJWT Function code here.
function authenticateJWT(req, res, next) {
    // Get token from session
    const token = req.session.token;

    // If no token, return 401 Unauthorized
    if (!token) return res.status(401).json({ message: 'Unauthorized' });

    try {
      // Verify token
      const decoded = jwt.verify(token, SECRET_KEY);

      // Attach user data to request
      req.user = decoded;

      // Continue to the next middleware
      next();
    } catch (error) {
      // If invalid token, return 401
      return res.status(401).json({ message: 'Invalid token' });
    }
  }

// Insert your requireAuth Function code here.
function requireAuth(req, res, next) {
    const token = req.session.token;  // Retrieve token from session

    if (!token) return res.redirect('/login');  // If no token, redirect to login page

    try {
      const decoded = jwt.verify(token, SECRET_KEY);  // Verify the token using the secret key
      req.user = decoded;  // Attach decoded user data to the request
      next();  // Pass control to the next middleware/route
    } catch (error) {
      return res.redirect('/login');  // If token is invalid, redirect to login page
    }
  }
// Insert your routing HTML code here.
app.get('/', (req, res) => res.sendFile(path.join(__dirname, 'public', 'index.html')));
app.get('/register', (req, res) => res.sendFile(path.join(__dirname, 'public', 'register.html')));
app.get('/login', (req, res) => res.sendFile(path.join(__dirname, 'public', 'login.html')));
app.get('/post', requireAuth, (req, res) => res.sendFile(path.join(__dirname, 'public', 'post.html')));
app.get('/index', requireAuth, (req, res) => res.sendFile(path.join(__dirname, 'public', 'index.html'), { username: req.user.username }));
// Insert your user registration code here.
app.post('/register', async (req, res) => {
    const { username, email, password } = req.body;

    try {
      // Check if the user already exists
      const existingUser = await User.findOne({ $or: [{ username }, { email }] });

      if (existingUser) return res.status(400).json({ message: 'User already exists' });

      // Create and save the new user
      const newUser = new User({ username, email, password });
      await newUser.save();

      // Generate JWT token and store in session
      const token = jwt.sign({ userId: newUser._id, username: newUser.username }, SECRET_KEY, { expiresIn: '1h' });
      req.session.token = token;

      // Respond with success message
      res.send({"message":`The user ${username} has been added`});
    } catch (error) {
      console.error(error);
      // Handle server errors
      res.status(500).json({ message: 'Internal Server Error' });
    }
  });

// Insert your user login code here.
app.post('/login', async (req, res) => {
    const { username, password } = req.body;

    try {
      // Check if the user exists with the provided credentials
      const user = await User.findOne({ username, password });

      if (!user) return res.status(401).json({ message: 'Invalid credentials' });

      // Generate JWT token and store in session
      const token = jwt.sign({ userId: user._id, username: user.username }, SECRET_KEY, { expiresIn: '1h' });
      req.session.token = token;

      // Respond with a success message
      res.send({"message":`${user.username} has logged in`});
    } catch (error) {
      console.error(error);
      // Handle server errors
      res.status(500).json({ message: 'Internal Server Error' });
    }
  });
// Insert your post creation code here.
app.post('/post', authenticateJWT, async (req, res) => {
    const { text } = req.body;
    // Validate post content
    if (!text || typeof text !== 'string') {
        return res.status(400).json({ message: 'Please provide valid post content' });
    }
    try {
        // Create and save new post with userId
        const newPost = new Post({ userId: req.user.userId, text });
        await newPost.save();
        res.status(201).json({ message: 'Post created successfully', post: newPost });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
```

```
    }
});
// Get all posts for the authenticated user
app.get('/posts', authenticateJWT, async (req, res) => {
    try {
        // Fetch posts for the logged-in user
        const posts = await Post.find({ userId: req.user.userId });
        res.json({ posts });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
    }
});
// Insert your post updation code here.
app.put('/posts/:postId', authenticateJWT, async (req, res) => {
    const postId = req.params.postId;
    const { text } = req.body;
    try {
        // Find and update the post, ensuring it's owned by the authenticated user
        const post = await Post.findOneAndUpdate(
            { _id: postId, userId: req.user.userId },
            { text },
            { new: true } // Return updated post
        );
        // Return error if post not found
        if (!post) return res.status(404).json({ message: 'Post not found' });
        res.json({ message: 'Post updated successfully', updatedPost: post });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
    }
});
// Insert your post deletion code here.
app.delete('/posts/:postId', authenticateJWT, async (req, res) => {
    const postId = req.params.postId;
    try {
        // Find and delete the post, ensuring it's owned by the authenticated user
        const post = await Post.findOneAndDelete({ _id: postId, userId: req.user.userId });
        // Return error if post not found
        if (!post) return res.status(404).json({ message: 'Post not found' });
        res.json({ message: 'Post deleted successfully', deletedPost: post });
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: 'Internal Server Error' });
    }
});
// Insert your user logout code here.
app.get('/logout', (req, res) => {
    req.session.destroy((err) => {
      if (err) console.error(err); // Log any session destruction errors
      res.redirect('/login'); // Redirect to login page after logout
    });
  });
app.listen(PORT, () => console.log(`Server is running on port ${PORT}`));
```

8. Testing `/register` and `/login` Endpoints with cURL

   Note: Ensure the server is running before testing the endpoints.

Register a New User

```
curl -X POST -H "Content-Type: application/json" -d '{
  "username": "coopercoote",
  "email": "ccoote@gmail.com",
  "password": "cg123"
}' -k http://localhost:3000/register
```

- This will register a new user with the provided username, email, and password.
- Record the response message in Notepad or any other editor.

Register with Existing User Details

- Run the same command again to register with the same username, email, and password.
- You should receive an appropriate error message indicating the user already exists.
- Record the response message.

Login with username, password you just registered

```
curl  -X POST  -H 'Content-Type: application/json'  -d '{
    "username":"coopercoote",
    "password":"cg123"}' -k 'http://localhost:3000/login'
```

- This should successfully log in the user.
- Record the response message.

Login with Incorrect Password

- You should receive an error message indicating invalid login credentials.
- Record the response message.

  Note: Stop the server once testing is complete.

Exercise 4 - Developing the frontend component for a Social Media Dashboard

1. Paste the below code in `public/index.html`.

This HTML code is the main page of a Social Media Dashboard App.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Social Media Dashboard App</title>
  <style>
    body {
      font-family: 'Roboto', sans-serif;
      background-color: #f5f5f5;
      text-align: center;
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }
    header {
      background-color: #0077cc;
      color: #fff;
      padding: 20px;
      text-align: right;
    }
    h1 {
      color: #333;
      font-size: 2.5em;
      margin-bottom: 20px;
      text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.3);
    }
    p {
      color: #555;
      font-size: 1.2em;
    }
    .app-names {
      display: flex;
      justify-content: center;
      margin-top: 30px;
    }
    .app-names div {
      padding: 10px 20px;
      margin: 0 10px;
      background-color: #0099ff;
      color: #fff;
      border-radius: 5px;
      font-weight: bold;
      text-transform: uppercase;
    }
    .button-link {
      margin-top: 30px;
      padding: 15px 30px;
      font-size: 1.2em;
      border: 2px solid #0099ff;
      border-radius: 8px;
      background-color: #fff;
      color: #0099ff;
      cursor: pointer;
      text-decoration: none;
      display: inline-block;
    }
  </style>
</head>
<body>
  <header>
    <button id="registerBtn" class="button-link" style="display:none" onclick="window.location.href='/register'">Register</button>
    <button id="loginBtn" class="button-link" style="display:none" onclick="window.location.href='/login'">Login</button>
    <button id="logoutBtn" class="button-link" style="display:none" onclick="logout()">Logout</button>
  </header>
  <h1>Welcome to the Social Media Dashboard App!</h1>
  <p id="welcomeMessage">Explore and manage your social media accounts in one place.</p>
  <div class="app-names">
    <div>FaceSnap</div>
    <div>InstaBook</div>
    <div>TweetChat</div>
  </div>
  <button id="createPostBtn" class="button-link" style="display:none" onclick="redirectToPostCreation()">Create Post</button>
  <script>
    function getUsernameFromURL() {
      return new URLSearchParams(window.location.search).get('username');
    }
    function handleAuthentication() {
      const username = getUsernameFromURL();
      const registerBtn = document.getElementById('registerBtn');
      const loginBtn = document.getElementById('loginBtn');
      const logoutBtn = document.getElementById('logoutBtn');
      const createPostBtn = document.getElementById('createPostBtn');
      if (username) {
        registerBtn.style.display = loginBtn.style.display = 'none';
        logoutBtn.style.display = createPostBtn.style.display = 'inline-block';
        welcomeMessage.innerText = `Welcome, ${username}!`;
      } else {
        registerBtn.style.display = loginBtn.style.display = 'inline-block';
        logoutBtn.style.display = createPostBtn.style.display = 'none';
      }
    }
    function logout() {
      window.location.href = '/logout';
    }
    function redirectToPostCreation() {
      window.location.href = '/post';
    }
    window.onload = function () {
      getUsernameFromURL();
      handleAuthentication();
    };
  </script>
</body>
</html>
```

- **Social Media Dashboard App Functionality**
  - **Header Navigation:**
    - Displays **Register**, **Login**, and **Logout** buttons based on user authentication status.
  - **Dynamic Welcome Message:**
    - Greets users by name when logged in using the **`username`** from the URL.
  - **App Highlights Section:**
    - Showcases featured apps (**FaceSnap**, **InstaBook**, **TweetChat**) with stylized labels.
  - **Create Post Button:**
    - Visible only to authenticated users, allowing them to create new posts.
  - **JavaScript Logic:**
    - **`getUsernameFromURL()`** extracts the **`username`** parameter from the URL.
    - **`handleAuthentication()`** manages button visibility and updates the welcome message.
    - **`logout()`** redirects users to **`/logout`** for session termination.
    - **`redirectToPostCreation()`** navigates authenticated users to the post creation page (**`/post`**).
  - **Responsive UI:**
    - Uses CSS for a clean, modern, and responsive layout with hover effects and interactive buttons.

2. Paste the below code in `public/login.html`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login</title>
    <style>
        body{font-family:Arial,sans-serif;background-color:#f8f8f8;margin:0}
        .container{max-width:400px;margin:50px auto;padding:20px;background-color:#fff;box-shadow:0 0 10px rgba(0,0,0,.1);border-radius:8px}
        h1{color:#333;font-size:2em;margin-bottom:20px}
        .auth-form label{display:block;margin-bottom:8px;font-weight:bold}
        .auth-form input{width:100%;padding:10px;margin-bottom:15px;border:1px solid #ccc;border-radius:5px;box-sizing:border-box}
        .auth-form button{background-color:#0077cc;color:#fff;padding:10px 20px;border:none;border-radius:5px;cursor:pointer}
        .auth-form button:hover{background-color:#005faa}
        p{margin-top:20px;font-size:1.2em}
        p a{color:#0077cc;font-weight:bold;text-decoration:none}
        p a:hover{text-decoration:underline}
    </style>
</head>
<body>
    <div class="container">
        <h1>Login</h1>
        <form action="/login" method="post" class="auth-form">
            <label for="username">Username:</label>
            <input type="text" id="username" name="username" required>
            <br>
            <label for="password">Password:</label>
            <input type="password" id="password" name="password" required>
            <br>
            <button type="submit">Login</button>
        </form>
        <p>Don't have an account? <a href="/register">Register here</a>.</p>
    </div>
</body>
</html>
```

- **Login Page Functionality**
  - Provides a login form for the Social Media Dashboard app.
  - Allows users to input their **username** and **password**.
  - Submits a **POST** request to the **`/login`** endpoint to handle authentication.
  - Includes basic form validation using the **`required`** attribute on input fields.
  - Displays a **"Register here"** link for users without an account, redirecting them to the **`/register`** page.
  - Uses CSS for a clean, user-friendly, and responsive layout.
  - Adds hover effects on buttons and links for better interactivity.
  - Ensures mobile responsiveness through the **viewport** meta tag.

3. Paste the below code in `public/register.html`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Register</title>
    <style>
        body {
            font-family: 'Arial', sans-serif;
            background-color: #f8f8f8;
            margin: 0;
        }
        .container {
            max-width: 400px;
            margin: 50px auto;
            padding: 20px;
            background-color: #fff;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            border-radius: 8px;
        }
        h1 {
            color: #333;
            font-size: 2em;
            margin-bottom: 20px;
        }
        .auth-form label {
            display: block;
            margin-bottom: 8px;
            font-weight: bold;
        }
```

```
        .auth-form input {
            width: 100%;
            padding: 10px;
            margin-bottom: 15px;
            border: 1px solid #ccc;
            border-radius: 5px;
            box-sizing: border-box;
        }
        .auth-form button {
            background-color: #0077cc;
            color: #fff;
            padding: 10px 20px;
            border: none;
            border-radius: 5px;
            cursor: pointer;
        }
        .auth-form button:hover {
            background-color: #005faa;
        }
        p {
            margin-top: 20px;
            font-size: 1.2em;
        }
        p a {
            color: #0077cc;
            font-weight: bold;
            text-decoration: none;
        }
        p a:hover {
            text-decoration: underline;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Register</h1>
        <form action="/register" method="post" class="auth-form">
            <label for="username">Username:</label>
            <input type="text" id="username" name="username" required>
            <br>
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" required>
            <br>
            <label for="password">Password:</label>
            <input type="password" id="password" name="password" required>
            <br>
            <button type="submit">Register</button>
        </form>
        <p>Already have an account? <a href="/login">Login here</a>.</p>
    </div>
</body>
</html>
```

- **Registration Page Functionality**
  - Provides a registration form for the Social Media Dashboard app.
  - Allows users to input their **username**, **email**, and **password**.
  - Submits a **POST** request to the **`/register`** endpoint to handle user registration.
  - Includes basic form validation using the **`required`** attribute on input fields.
  - Displays a **"Login here"** link directing existing users to the **`/login`** page.
  - Uses CSS for a clean, user-friendly, and responsive layout.
  - Implements hover effects on buttons and links for improved interactivity.
  - Ensures mobile responsiveness through the **viewport** meta tag.

4. Paste the below code in `public/post.html`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Create and Manage Posts</title>
  <style>
    body {
      font-family: 'Roboto', sans-serif;
      background-color: #f5f5f5;
      text-align: center;
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }
    header {
      background-color: #0077cc;
      color: #fff;
      padding: 20px;
      text-align: right;
    }
    h1 {
      color: #333;
      font-size: 2.5em;
      margin-bottom: 20px;
      text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.3);
    }
    .post-form, .post-list {
      width: 60%;
      margin: 50px auto;
      padding: 20px;
      background-color: #fff;
      border-radius: 8px;
      box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    }
    label, textarea {
      display: block;
      margin-bottom: 10px;
      font-size: 1.2em;
      color: #333;
    }
    textarea {
      width: 100%;
      padding: 10px;
      margin-bottom: 20px;
```

```
      border: 1px solid #ccc;
      border-radius: 4px;
      box-sizing: border-box;
    }
    button {
      background-color: #0099ff;
      color: #fff;
      padding: 10px 20px;
      font-size: 1.2em;
      border: none;
      border-radius: 8px;
      cursor: pointer;
    }
    .post {
      border-bottom: 1px solid #ccc;
      padding: 10px;
      margin-bottom: 10px;
      display: flex;
      justify-content: space-between;
      align-items: center;
    }
    .post-text {
      flex-grow: 1;
      margin-right: 10px;
      font-size: 1.1em;
      color: #333;
    }
    .edit-post-btn, .delete-post-btn {
      padding: 8px;
      margin-right: 5px;
      cursor: pointer;
      background-color: #0099ff;
      color: #fff;
      border: none;
      border-radius: 4px;
    }
    .pagination {
      margin-top: 20px;
    }
    .pagination button {
      background-color: #0099ff;
      color: #fff;
      border: none;
      padding: 10px;
      margin: 0 5px;
      cursor: pointer;
      border-radius: 4px;
      transition: background-color 0.3s;
    }
    .pagination button:hover {
      background-color: #0077cc;
    }
  </style>
</head>
<body>
  <header>
    <button id="logoutBtn" onclick="logout()">Logout</button>
  </header>
  <h1>Create and Manage Posts</h1>
  <div class="post-form">
    <label for="text">Create a new post:</label>
    <textarea id="text" name="text" rows="4" required></textarea>
    <button onclick="createPost()">Create Post</button>
  </div>
  <div class="post-list" id="postList"></div>
  <div class="pagination" id="pagination"></div>
  <script>
    const postsPerPage = 3;
    let currentPage = 1;
    // On page load, fetch the posts
    document.addEventListener('DOMContentLoaded', () => {
      getPosts();
    });
    // Create a new post
    async function createPost() {
      const text = document.getElementById('text').value;
      if (!text) return alert('Post text cannot be empty');
      try {
        const response = await fetch('/post', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ text })
        });
        const data = await response.json();
        if (response.ok) {
          document.getElementById('text').value = '';
          currentPage = 1; // Reset to first page after new post
          getPosts(); // Refresh list
        } else {
          alert(data.message);
        }
      } catch (error) {
        console.error('Error:', error);
        alert('Error creating post');
      }
    }
    // Fetch posts from the server
    async function getPosts() {
      try {
        const response = await fetch('/posts', { method: 'GET' });
        const data = await response.json();
        if (response.ok) {
          renderPosts(data.posts);
        } else {
          console.error(data.message);
        }
      } catch (error) {
        console.error('Error:', error);
      }
    }
    // Render the posts into the DOM with pagination
    function renderPosts(posts) {
      const postList = document.getElementById('postList');
      const pagination = document.getElementById('pagination');
      postList.innerHTML = '';
      if (posts.length === 0) {
        postList.innerHTML = '<p>No posts available.</p>';
        pagination.innerHTML = '';
        return;
      }
      // Calculate slice indices for the current page
      const startIndex = (currentPage - 1) * postsPerPage;
      const endIndex = startIndex + postsPerPage;
```

```
      const paginatedPosts = posts.slice(startIndex, endIndex);
      // Render posts for the current page
      paginatedPosts.forEach(post => {
        const postDiv = document.createElement('div');
        postDiv.className = 'post';
        const postText = document.createElement('div');
        postText.className = 'post-text';
        postText.textContent = post.text;
        // Edit button
        const editButton = document.createElement('button');
        editButton.textContent = 'Edit';
        editButton.className = 'edit-post-btn';
        editButton.onclick = () => editPost(post);
        // Delete button
        const deleteButton = document.createElement('button');
        deleteButton.textContent = 'Delete';
        deleteButton.className = 'delete-post-btn';
        deleteButton.onclick = () => deletePost(post._id);
        postDiv.appendChild(postText);
        postDiv.appendChild(editButton);
        postDiv.appendChild(deleteButton);
        postList.appendChild(postDiv);
      });
      // Create pagination buttons
      const totalPages = Math.ceil(posts.length / postsPerPage);
      pagination.innerHTML = '';
      for (let i = 1; i <= totalPages; i++) {
        const pageButton = document.createElement('button');
        pageButton.textContent = i;
        pageButton.onclick = () => {
          currentPage = i;
          renderPosts(posts);
        };
        // Highlight the active page button
        if (i === currentPage) {
          pageButton.style.fontWeight = 'bold';
        }
        pagination.appendChild(pageButton);
      }
    }
    // Prompt for editing and then update the post
    function editPost(post) {
      const newText = prompt('Edit your post:', post.text);
      if (newText !== null && newText !== post.text) {
        updatePost(post._id, newText);
      }
    }
    // Update a post via the API
    async function updatePost(postId, text) {
      try {
        const response = await fetch(`/posts/${postId}`, {
          method: 'PUT',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ text })
        });
        const data = await response.json();
        if (response.ok) {
          getPosts(); // Refresh list
        } else {
          alert(data.message);
        }
      } catch (error) {
        console.error('Error:', error);
        alert('Error updating post');
      }
    }
    // Delete a post via the API
    async function deletePost(postId) {
      try {
        const response = await fetch(`/posts/${postId}`, { method: 'DELETE' });
        const data = await response.json();
        if (response.ok) {
          getPosts();
        } else {
          alert(data.message);
        }
      } catch (error) {
        console.error('Error:', error);
        alert('Error deleting post');
      }
    }
    // Logout by redirecting to the logout endpoint
    function logout() {
      window.location.href = '/logout';
    }
  </script>
</body>
</html>
```

- **Creating a Post (POST /post)**
  - The `createPost()` function sends a POST request to `/post` with the post content.
  - The backend handles authentication and saves the post in MongoDB.
  - After a successful post, the frontend clears the input, resets to the first page, and refreshes the post list using `getPosts()`.
- **Fetching Posts (GET /posts)**
  - `getPosts()` fetches all posts belonging to the logged-in user and calls `renderPosts()` to display them.
- **Updating a Post (PUT /posts/:postId)**
  - `editPost()` prompts the user for new content and, if changed, sends an update request using `updatePost()`.
  - The backend updates the post in MongoDB and returns a success message, after which the post list is refreshed.
- **Deleting a Post (DELETE /posts/:postId)**
  - `deletePost()` sends a DELETE request to `/posts/:postId`.
  - The backend deletes the post and returns a success message, and the frontend refreshes the list of posts.

- **Pagination**
  - Uses `postsPerPage` (set to 3) and `currentPage` to manage post display.
  - `renderPosts()` slices posts based on the current page and generates pagination buttons.
  - Clicking a pagination button updates `currentPage` and re-renders posts.
- **Logout (/logout)**
  - The "Logout" button triggers the `logout()` function, redirecting to `/logout` and clearing the session.

5. In `app.js` replace the JSON response on successful registration, with the following.

```
res.redirect(`/index?username=${newUser.username}`);
```

This will redirect the response to `/index` with the username parameter set.

6. In `app.js` replace the JSON response on successful login, with the following.

```
res.redirect(`/index?username=${user.username}`);
```

This will redirect the response to `/index` with the username parameter set.

Exercise 5 - Deploy the Social Media Dashboard app on Docker

1. Replace the mongo URL and connect in `app.js` with the following.

```
const uri =  "mongodb://mongodb:27017";
mongoose.connect(uri,{'dbName':'SocialDB'});
```

Note: This URI is used because the Docker Compose setup defines a MongoDB service named `mongodb`, and `27017` is the default MongoDB port. Unlike Exercise 1, no credentials are required here, as the MongoDB container is configured without authentication.

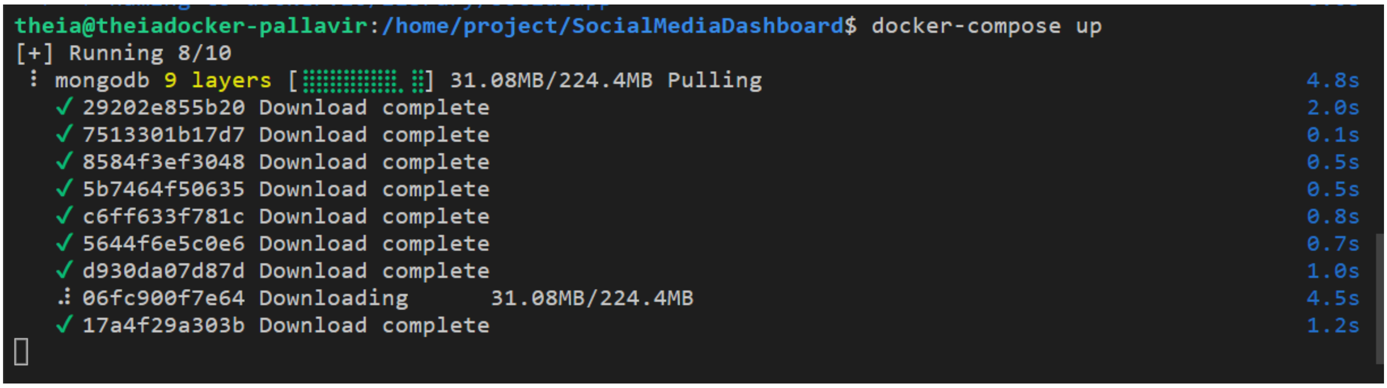
2. Run the following command to build the Docker app

```
docker build . -t socialapp
```

```
theia@theiadocker-pallavir:/home/project/abltc-backend-nodejs-customerportal$ d
ocker build . -t customerapp
[+] Building 20.5s (11/11) FINISHED                                                docker:default
 => [internal] load build definition from Dockerfile                                         0.0s
 => => transferring dockerfile: 436B                                                         0.0s
 => [internal] load .dockerignore                                                            0.0s
 => => transferring context: 2B                                                              0.0s
 => [internal] load metadata for docker.io/library/node:18.12.1-bullseye-slim                0.5s
 => [auth] library/node:pull token for registry-1.docker.io                                  0.0s
 => [internal] load build context                                                            1.1s
 => => transferring context: 15.90MB                                                         1.1s
 => [1/5] FROM docker.io/library/node:18.12.1-bullseye-slim@sha256:70bf84739156657c85440e6a55a3d77a7  13.9s
 => => resolve docker.io/library/node:18.12.1-bullseye-slim@sha256:70bf84739156657c85440e6a55a3d77a7c  0.0s
 => => sha256:3f4ca61aafcd4fc07267a105067db35c0f0ac630e1970f3cd0c7bf552780e985 31.40MB / 31.40MB  3.3s
 => => sha256:00fde01815c92cc90586fcf531723ab210577a0f1cb1600f08d9f8e12c18f108 4.18kB / 4.18kB  0.1s
 => => sha256:723367e6ef2d175cd363a4ca96a65ac6fa3f388506de7c0489318e8d8ed59dd7 45.15MB / 45.15MB  5.0s
 => => sha256:70bf84739156657c85440e6a55a3d77a7cac668f9c4c3c44005bc29bdc529db7 1.21kB / 1.21kB  0.0s
 => => sha256:0c3ea57b6c560f83120801e222691d9bd187c605605185810752a19225b5e4d9 1.37kB / 1.37kB  0.0s
```

3. The docker-compose.yml has been created to run two containers, one for Mongo and the other for the Node app. Run the following command to run the server:

```
docker-compose up
```

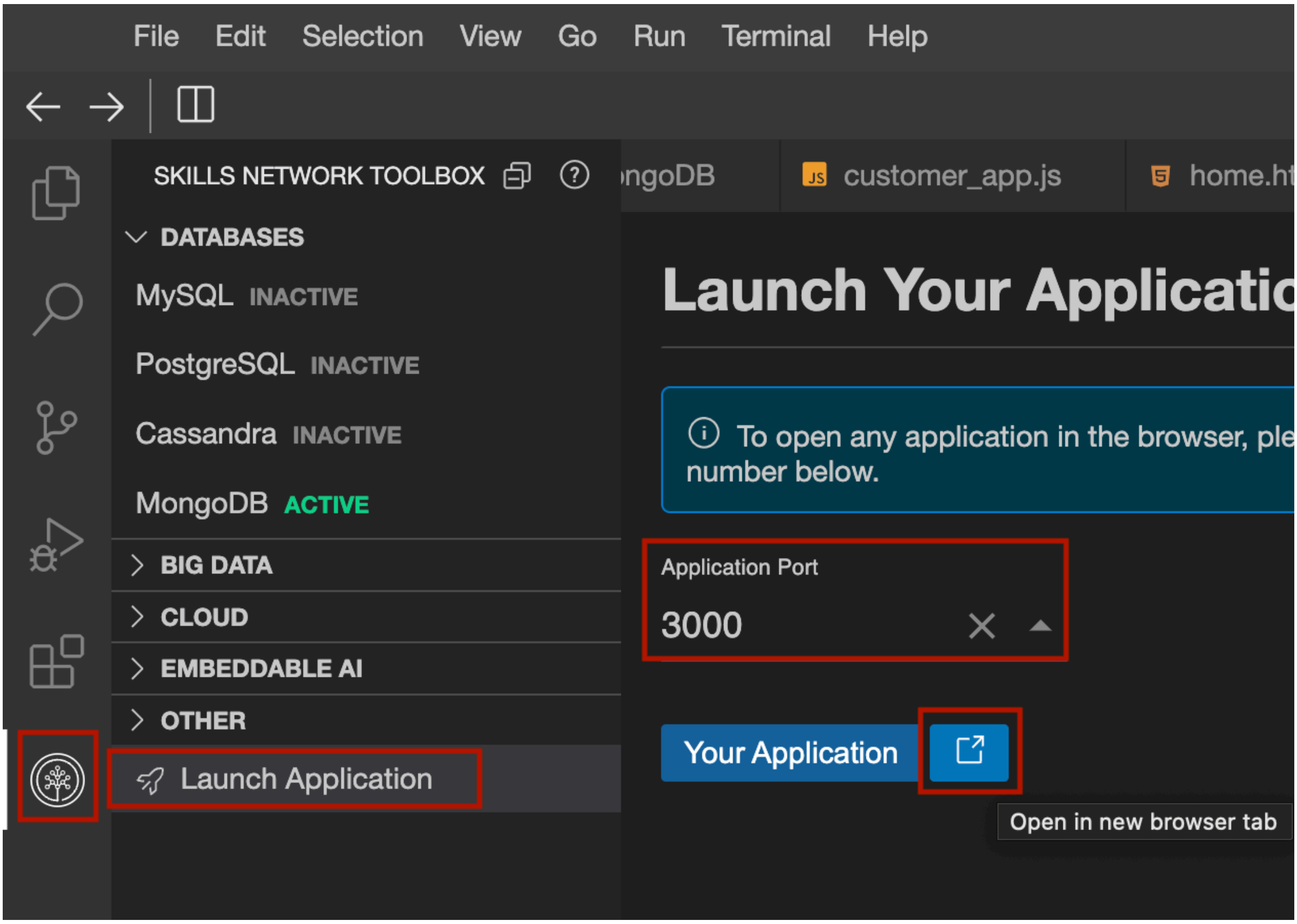


4. Click on the button below to launch the application.

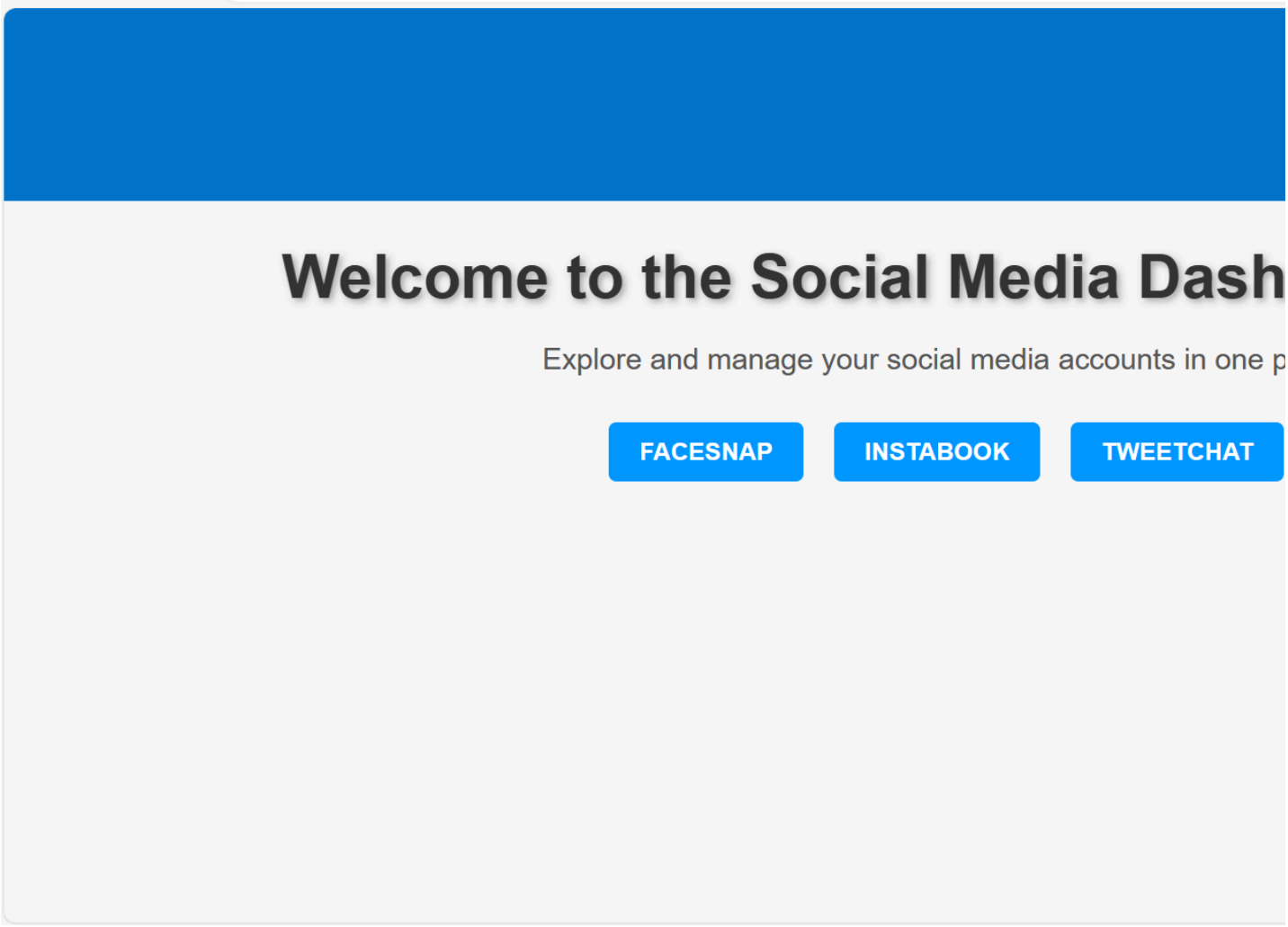
Social Media Dashboard

▼ Click here if the button above doesn't work

1. Click on the `Skills Network` icon in the left tool bar.
2. Choose `Launch Application`.
3. Enter the port number `3000` on which the application is running.
4. Click on the icon, as shown in the image, to open the customer portal in a new tab.



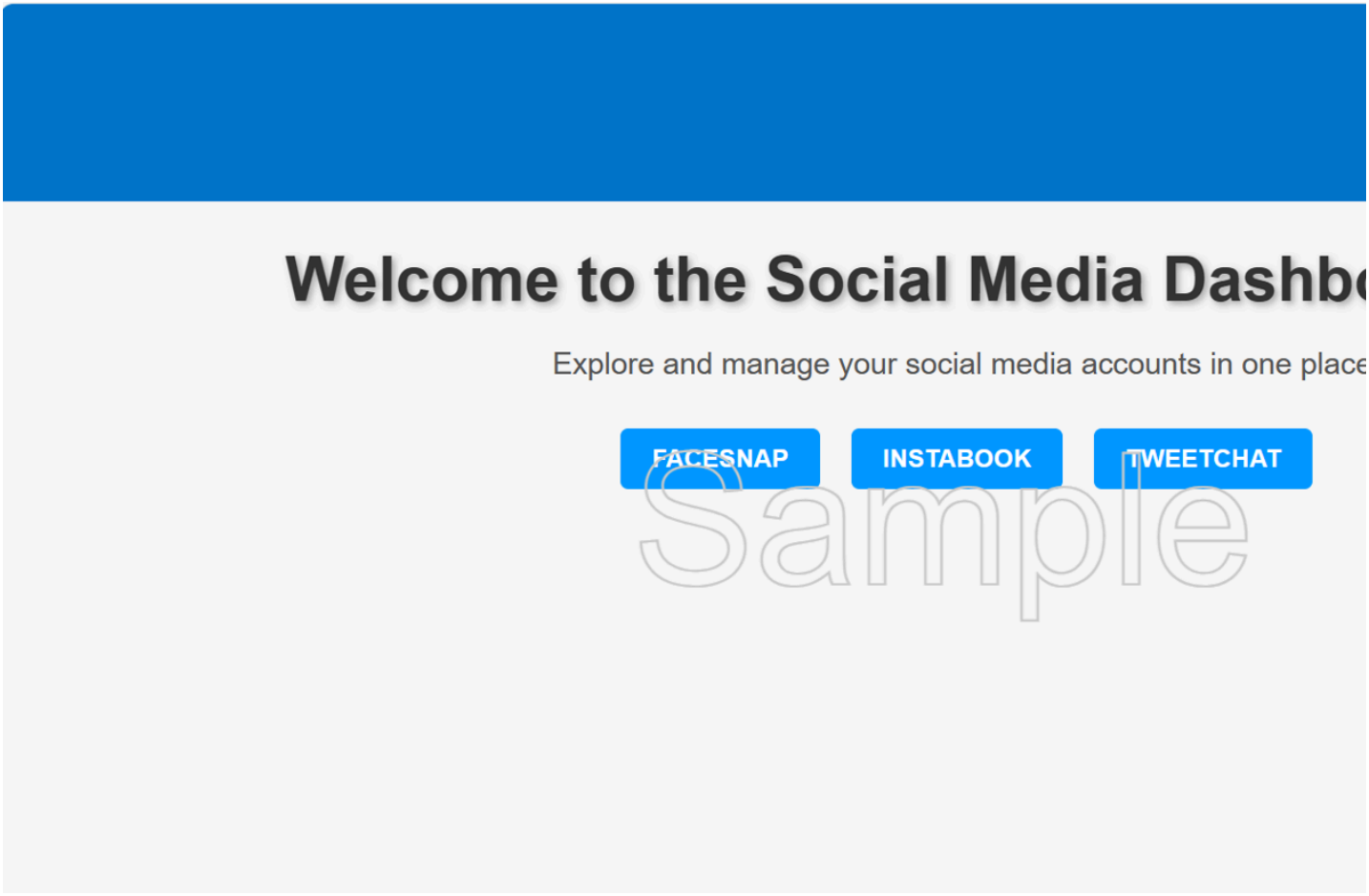
4. Verify the dashboard:



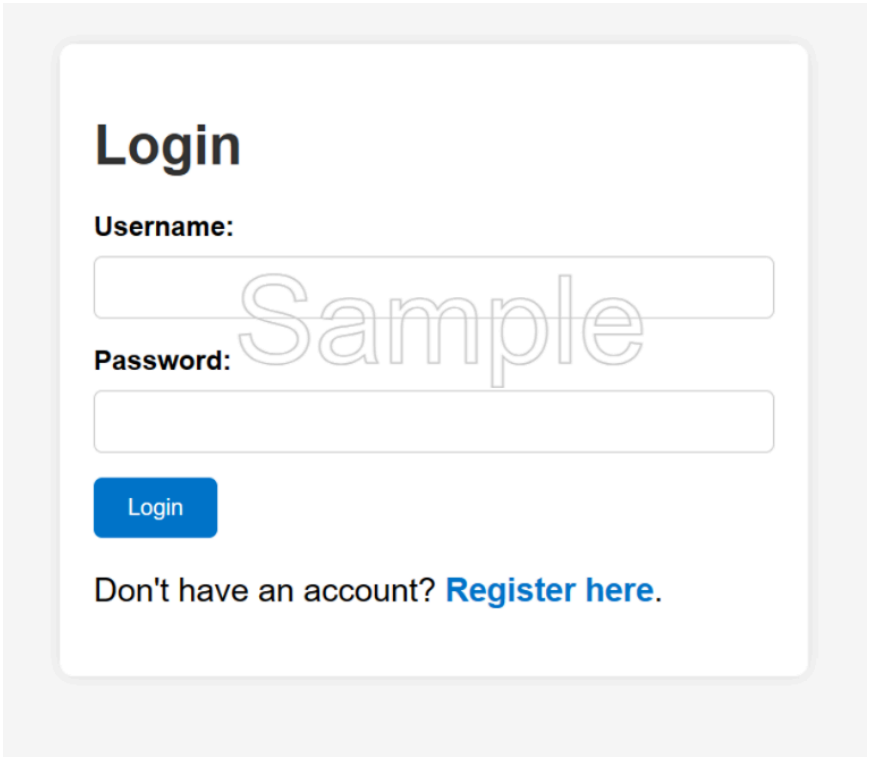
Note: Download the `app.js` file.This file will be referenced later for final assignment.

Exercise 6 - Working of Social Media Dashboard app

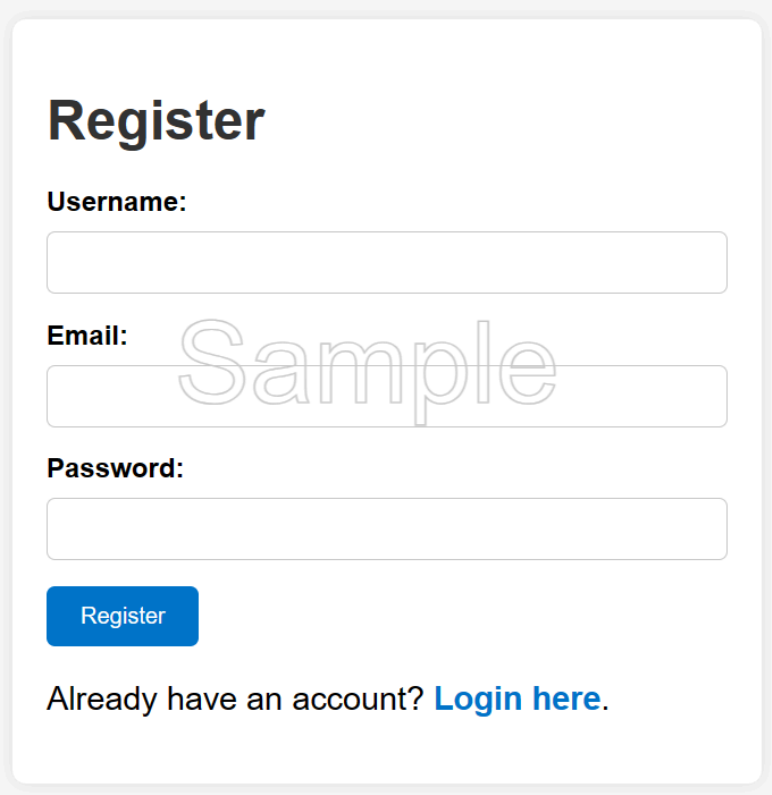
1. Launch the application to verify the successful creation of the social media dashboard.



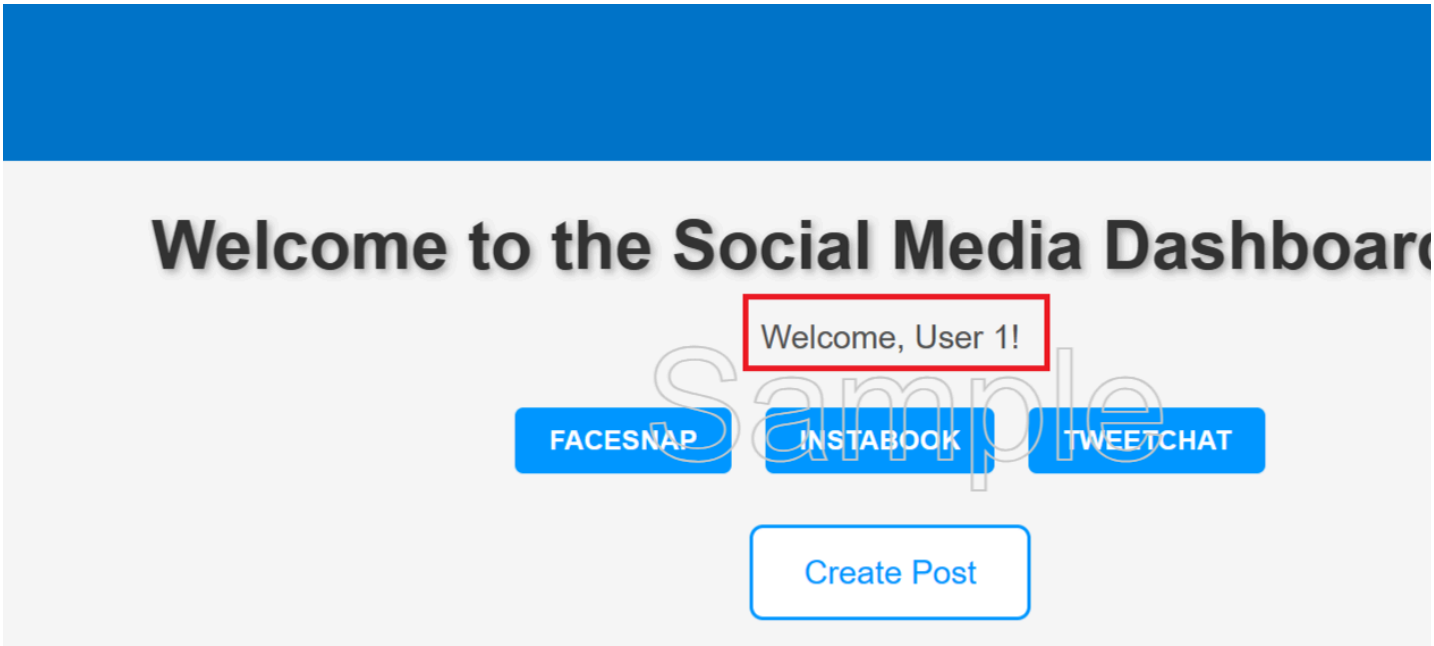
2. Append "/post" to the end of the URL and verify that you are redirected to the login page, indicating that posting is restricted without authentication.



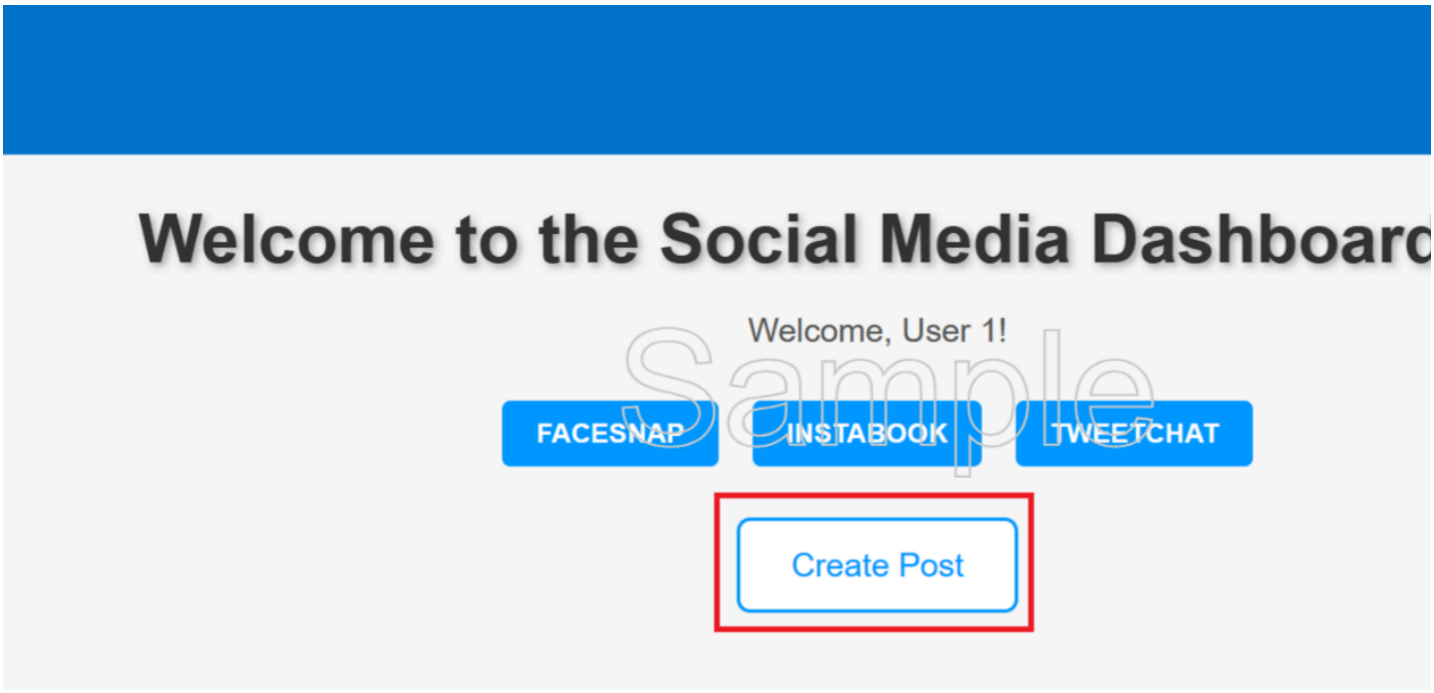
3. Click on the register button, provide the required details, and complete the registration process to gain access to the application.



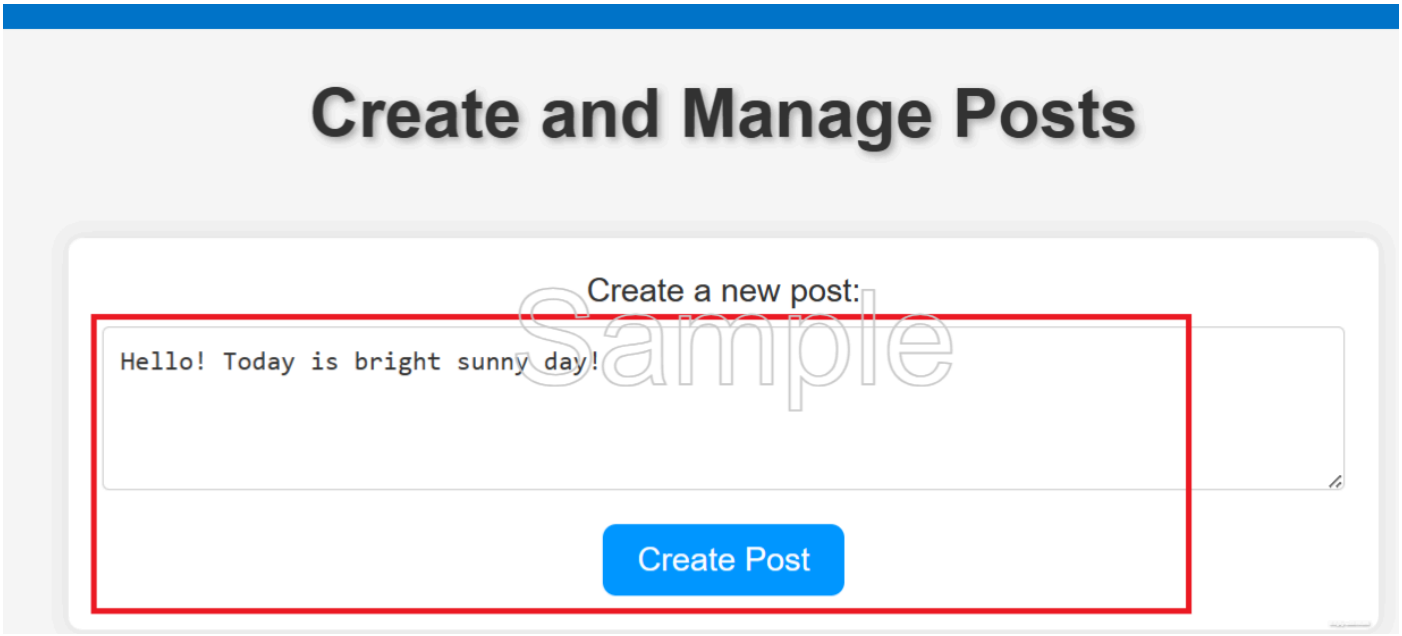
4. Upon successful registration, observe the welcome message containing your username on the app interface.



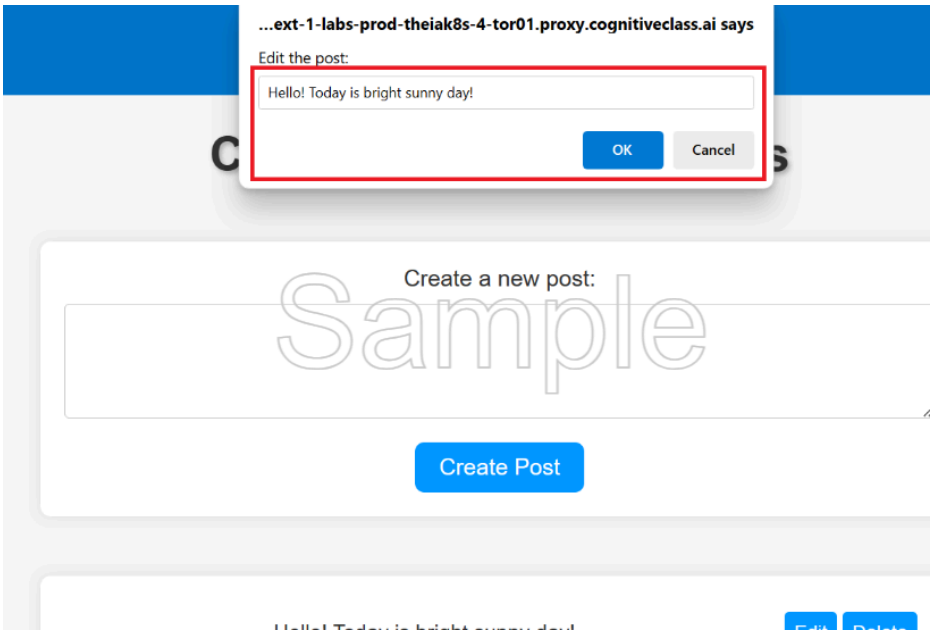
5. Confirm the visibility of the "Create post" button on the application page post-registration.



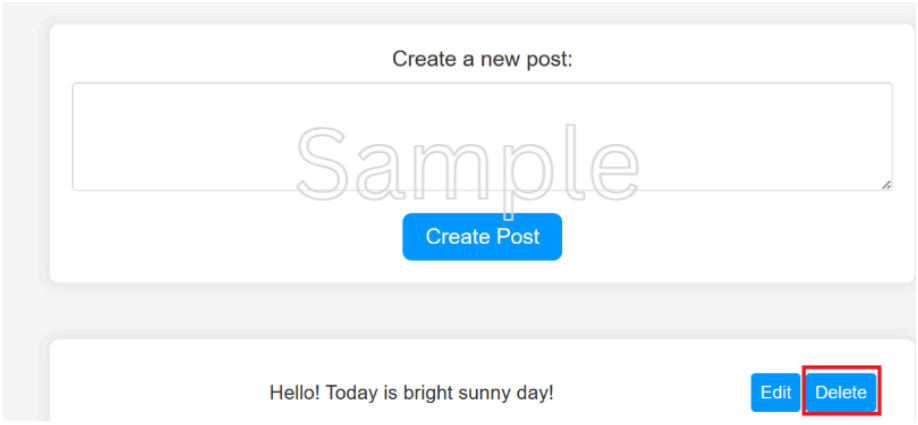
6. Verify your ability to successfully create a post, ensuring that the content is accurately reflected.



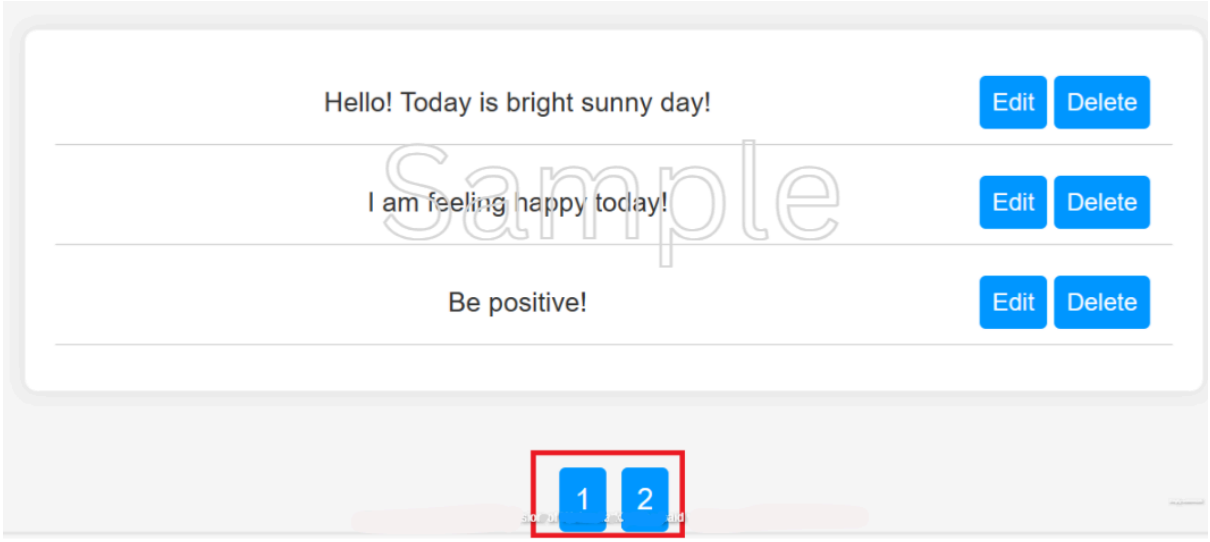
7. Confirm your capability to edit the posts you've created without encountering issues.



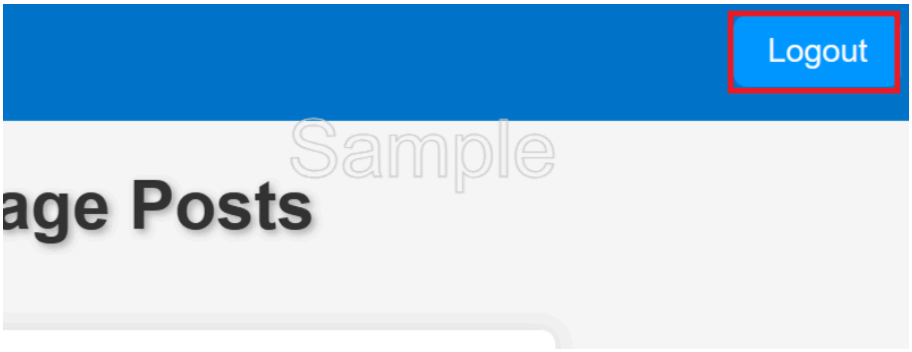
8. Check the ability to delete the posts you've created with the app, ensuring the process works correctly.



9. Observe the pagination feature by adding three posts and confirming that the fourth post is displayed on a new page.



10. Click on the logout button and ensure successful logout from the application.



Note: Ensure that you save the output of all the curl commands used in Exercise 3, and download the app.js file once the project is complete. These will be needed to answer the questions in the assignment following the project.

Conclusion

Congratulations! You have successfully finished this Final Project lab and acquired the skills to develop a Social Media dashboard application.

In this lab, you learned how to:

- Create a Social Media Dashboard web app with user authentication, JWT-based security, and post management.
- Develop a web application with Node.js and Express on the backend, and HTML/CSS/JS on the frontend. Include features like user registration, login, post creation, and pagination.

Authors

Pallavi Rai

Other Contributors

Nikesh Kumar