# **טכנולוגיות אינטרנט מתקדמות - 61776 (WEB)**

**הגשת פרויקט**

**להגשה עד 29.8.24 בשעה 23:59**

**networking-facilitator B15 group27**

|  |  |
| --- | --- |
| **שם חבר.ת הצוות** | **תז** |
| מוחמד אבו אחמד | 314934613 |
| שאדי מנסור | 211732946 |
| איאס ריזק | 208262733 |
| מגד סלאמה | 211864954 |
| יעקוב סדראן | 314892498 |

1. עליכם להמשיך את בניית האתר לפי האלמנטים המתקדמים שלמדתם
2. יש למנות מהנדס מערכת בכל צוות, אשר יהיה אחראי על הגדרת והקצאת המשימות בתרגיל זה.  
   נא לרשום את שם הסטודנט בתרגיל זה. על מהנדס המערכת לכתוב כיצד נעשתה חלוקת העבודה מול הצוות, מה היו המשימות של כל חבר צוות, האם היה ממשק בין חברי הצוות, והאם המשימות מולאו:

A white sheet with black text

Description automatically generated

**מימוש-שמות הטכנולוגיות המרכזיות בכל אחד מהחלקים:**

Styling: CSS+ tailwind + React

Db: MongoDB

Backend: VS Code using Node.js and Express.js

Frontend: VS Code using React, JavaScript, HTML, Axios, React Router

קישור לתיקיית גיט : https://github.com/majdsalameh1211/web\_final\_M.M..git

קישור ל Git Pages:

קישור ל -MTW: https://www.morethanwallet.com/app/891

https://web-final-m-m.vercel.app/: vercel קישור

2. הציגו רשימת דרישות פונקציונליות ולא פונקציונליות (בנפרד, יש לסווג דרישות לא פונקציונליות לפי wikipedia NFR).

**דרישות פונקציונליות:**

**User Registration**: Users can sign up on the website and create a personal profile.

**Post Management**: Users can create posts.

**Comments and Likes**: Users can comment on and like posts from other users.

**Friend Search**: Users can search for friends and send follow requests.

**Profile Update**: Users can edit their profile, including personal details.

**reset password :**user who forgot his password can retrive using the 2 recovery qustions .

**login :**the system allows users to get into website by login .

**דרישות לא פונקציונליות:**

**High Availability**: The website should be available at least 99.9% of the time, with minimal downtime.

**Performance**: The average page load time should not exceed 5 seconds.

**Data Security**: The website must protect user information, including password encryption and SSL usage.

**Scalability**: The system should be scalable to support an increasing number of users.

הציגו ארכיטקטורה מעודכנת של האתר (תרשים הכולל את האלמנטים המרכזיים).

3. הציגו דיאגרמת use case המתארת את השימוש באתר.

A diagram of a company

Description automatically generated

A diagram of a server

Description automatically generated with medium confidence

networking-main/

├── backend/ # Backend directory

│ ├── entities/ # Entities (data models) for backend

│ ├── server/ # Server-related files

│ │ ├── node\_modules/ # Node modules for server

│ │ ├── database.js # Database connection or configuration file

│ │ ├── package-lock.json # Lock file for server dependencies

│ │ ├── package.json # Package manifest for server

│ │ ├── server.js # Main server file

│ │ └── vercel.json # Vercel configuration for server deployment

│ ├── websocket/ # WebSocket-related files

│ │ ├── Procfile # Heroku or similar service process file

│ │ ├── websocketServer.js # WebSocket server implementation

│ │ ├── package.json # Package manifest for WebSocket server

│ │ └── vercel.json # Vercel configuration for WebSocket deployment

│ ├── .gitignore # Git ignore file for backend

│ ├── mongoDBserverBackup.js # Backup script or utility for MongoDB

│ ├── package-lock.json # Lock file for backend dependencies

│ ├── package.json # Package manifest for backend

│ └── websocketbackup.js # Backup WebSocket server implementation or script

├── frontend/ # Frontend directory

│ ├── build/ # Production build files for frontend

│ ├── node\_modules/ # Node modules for frontend

│ ├── public/ # Public assets directory for frontend

│ ├── src/ # Source code for frontend

│ │ ├── api/ # API-related files for frontend

│ │ │ └── axios.jsx # Axios instance or configuration file

│ │ ├── components/ # Reusable components for frontend

│ │ │ ├── AddPost.js # Component for adding posts

│ │ │ ├── EditPost.js # Component for editing posts

│ │ │ └── Layout.js # Layout component

│ │ ├── entities/ # Frontend data models or entities

│ │ ├── pages/ # Page components for frontend

│ │ │ ├── CommentSidebar/ # Component for comment sidebar

│ │ │ ├── ForgetPasswordPage/ # Component for password recovery

│ │ │ ├── FriendsPage/ # Component for friends management

│ │ │ ├── HomePage/ # Component for homepage

│ │ │ ├── icons/ # Icons used in the frontend

│ │ │ ├── LoginPage/ # Component for login page

│ │ │ ├── NotificationsPage/ # Component for notifications

│ │ │ ├── ProfilePage/ # Component for user profile

│ │ │ ├── RegisterPage/ # Component for registration page

│ │ │ └── SettingsPage/ # Component for user settings

│ │ ├── Sidebar/ # Sidebar components

│ │ ├── DarkModeToggle.js # Component for dark mode toggle

│ │ ├── MessagingPage.js # Component for messaging page

│ │ ├── UserContext.jsx # Context API for user management

│ │ ├── App.css # Global CSS for app

│ │ ├── App.js # Main App component

│ │ ├── App.test.js # Test file for App component

│ │ ├── index.css # Global CSS for index

│ │ ├── index.js # Entry point for the React application

│ │ ├── logo.svg # Logo for the application

│ │ ├── reportWebVitals.js # Performance measuring for web vitals

│ │ ├── setupTests.js # Setup for testing environment

│ ├── .gitignore # Git ignore file for frontend

│ ├── package-lock.json # Lock file for frontend dependencies

│ ├── package.json # Package manifest for frontend

│ ├── postcss.config.js # PostCSS configuration for Tailwind CSS

│ ├── README.md # Documentation for frontend

│ └── tailwind.config.js # Tailwind CSS configuration file

4 יש להציג מבנה סופי של האתר שלכם:

https://networking-lotw.vercel.app/

**נדרשת בכל פרויקט פריסה מלאה (deploment) של הפרויקט!   
לא יתקבלו הגשות של קבצים או אתר ב -localhost .**

**טכנולוגיות**:    להלן המרכיבים הטכנולוגיים המומלצים לשימוש בפרויקט:

·           - front-end: React or Preact with Tailwind

    - back-end:

    - option 1 - remote services/APIs from morethanwallet.com (https://www.morethanwallet.com/appStore/apidoc)

    - option 2 - node js and express deployed on remote web server

א. האתר ימומש ב -react/preact, וכן שימוש ב Tailwind - נא להציג דיאגרמה המתארת את התיקיות והקבצים השונים. יש לפרט את הקומפוננטות השונות.

ב. יש לפרט את פריטי המידע - יש להשתמש במידע אמיתי ורלוונטי לפרויקט שלכם (בשליפה ממסד נתונים חיצוני או מ - API). יש להראות דיאגרמת מבנה DB.

Backend & Frontend   
DB :  
A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a recovery questions

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a social media post

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A diagram of a user flow

Description automatically generated

Loginpage:

A screenshot of a login form

Description automatically generated

The code defines a user verification function and a login endpoint to handle user authentication. The \*\*verifyUser function\*\* checks a user's credentials by searching for the user's email in the database. If the user is not found or the password does not match, it returns an error message indicating the credentials are incorrect. If the credentials are correct, it constructs a user object with all relevant details and passes it back using a callback function.

The \*\*login endpoint\*\* (`/login`) handles login requests by taking the email and password from the request body. It calls the `verifyUser` function to check the credentials. If there is an error or the credentials are incorrect, it sends an appropriate error message back to the client. If the credentials are valid, it responds with a success message and the user’s details. This setup ensures secure and efficient user authentication for the application.

Server side:

// Verify user function for login

const verifyUser = async (email, password, callback) => {

    try {

        const user = await db.collection('users').findOne({ email });

        if (!user) {

            console.log('No user found with this email:', email);

            return callback(null, { message: 'Wrong email or password.' });

        }

        if (password !== user.password) {

            return callback(null, { message: 'Wrong email or password.' });

        } else {

            const userObject = {

                user\_id: user.\_id,

                user\_name: user.user\_name,

                first\_name: user.first\_name,

                last\_name: user.last\_name,

                gender: user.gender,

                email: user.email,

                password: user.password,

                phone\_number: user.phone\_number,

                education: user.education,

                photo: user.photo

            };

            callback(null, userObject);

        }

    } catch (err) {

        console.error('Database query error:', err);

        return callback(err);

    }

};

// Add login endpoint

app.post('/login', (req, res) => {

    const { email, password } = req.body;

    console.log("server side :");

    console.log(email, password);

    verifyUser(email, password, (err, result) => {

        console.log("err:", err);

        console.log(result);

        if (err) {

            return res.status(500).send('An error occurred. Please try again.');

        }

        if (result.message) {

            return res.status(401).send(result.message);

        }

        res.json({ message: 'Login successful', user: result });

    });

});

Client side:

const LoginPage = ({ setAuth }) => {

  const [formData, setFormData] = useState({

    email: '',

    password: ''

  });

  const [error, setError] = useState('');

  const navigate = useNavigate();

  const { setCurrentUser } = useUser();

  const handleChange = (e) => {

    setFormData({ ...formData, [e.target.name]: e.target.value });

  };

  const handleSubmit = async (e) => {

    e.preventDefault();

    try {

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

      console.log('frontend:', response);

      if (response.data.message === 'Login successful') {

        setAuth(true);

        setCurrentUser(response.data.user);

        navigate('/home');

      } else {

        setError('Invalid credentials');

      }

    } catch (error) {

      setError('Invalid credentials');

    }

  };

A screenshot of a computer screen

Description automatically generated

regesterpage:

A screenshot of a computer

Description automatically generated

A screenshot of a registration form

Description automatically generated

The code provides two functionalities for handling user registration and login verification in an application. The \*\*user registration endpoint\*\* (`/register`) processes user registration requests by extracting the necessary details from the request body, such as username, email, password, and other personal information. It creates a new user object and inserts it into the `users` collection in the database. Upon successful insertion, it stores additional user data like skills and recovery questions in their respective collections, linking them to the new user ID. If all operations are successful, it returns a success message and the new user's details, while handling errors gracefully by logging them and sending appropriate error messages back to the client.

The \*\*verify user function\*\* (`verifyUser`) handles user login verification by taking an email and password as input and searching for a matching user in the `users` collection. It checks if the user exists and whether the provided password matches the stored one. If the credentials are incorrect, it returns an error message indicating that the email or password is wrong. If the credentials are valid, it constructs a user object containing all relevant details and returns it via a callback. The function also includes error handling to manage any database query failures, ensuring errors are logged and appropriately reported back to the caller.

Serverside:

// User registration endpoint

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

    console.log('Request received at /register');

    console.log('Request body:', req.body);

    const { username, first\_name, last\_name, gender, email, password, phone\_number, education, photo, skills, recovery\_q1, recovery\_q2 } = req.body;

    try {

        // Create user object

        const newUser = {

            user\_name: username,

            first\_name,

            last\_name,

            gender,

            email,

            password,

            phone\_number,

            education,

            photo

        };

        // Insert new user into users collection

        const result = await db.collection('users').insertOne(newUser);

        if (result.acknowledged && result.insertedId) {

            const userId = result.insertedId;

            // Insert skills into skills collection

            const skillsData = skills.map(skill => ({ user\_id: userId, skill }));

            await db.collection('skills').insertMany(skillsData);

            // Insert recovery questions into recovery\_questions collection

            const recoveryData = {

                user\_id: userId,

                question1: recovery\_q1.question,

                answer1: recovery\_q1.answer,

                question2: recovery\_q2.question,

                answer2: recovery\_q2.answer

            };

            await db.collection('recovery\_questions').insertOne(recoveryData);

            // Return the new user object with just the user fields

            res.json({ message: 'Registration successful', user: newUser });

        } else {

            console.error('Failed to insert user into the database');

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

        }

    } catch (error) {

        console.error('Error during registration:', error);

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

    }

});

// Verify user function for login

const verifyUser = async (email, password, callback) => {

    try {

        const user = await db.collection('users').findOne({ email });

        if (!user) {

            console.log('No user found with this email:', email);

            return callback(null, { message: 'Wrong email or password.' });

        }

        if (password !== user.password) {

            return callback(null, { message: 'Wrong email or password.' });

        } else {

            const userObject = {

                user\_id: user.\_id,

                user\_name: user.user\_name,

                first\_name: user.first\_name,

                last\_name: user.last\_name,

                gender: user.gender,

                email: user.email,

                password: user.password,

                phone\_number: user.phone\_number,

                education: user.education,

                photo: user.photo

            };

            callback(null, userObject);

        }

    } catch (err) {

        console.error('Database query error:', err);

        return callback(err);

    }

};

Clientside:

 const handleRegister = async () => {

    if (formData.recovery\_q1.question && formData.recovery\_q1.answer && formData.recovery\_q2.question && formData.recovery\_q2.answer) {

      try {

        //console.log('Frontend data:', formData);

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

        //console.log('Response from backend:', response.data);

        setSuccess('Registration successful! You can now log in.');

        setError('');

        setCurrentUser(response.data.user);

        navigate('/home');

      } catch (error) {

        console.error('Error from backend:', error.response ? error.response.data : error.message);

        setError('Error registering user');

        setSuccess('');

      }

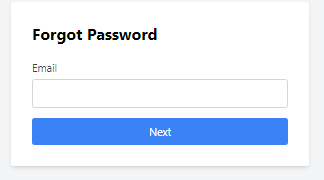
    } else {

      setError('Please fill out all required fields');

    }

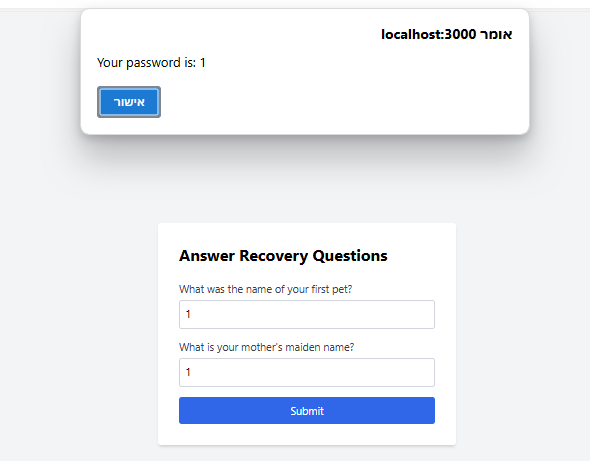
  };

Forget my password page:



A screenshot of a questionnaire

Description automatically generated



The provided code defines two Express.js endpoints to handle a "forgot password" feature. The first endpoint (`/forgot-password`) receives a user's email and checks if the email exists in the database. If found, it retrieves the corresponding recovery questions from the "recovery\_questions" collection and returns them to the client. If the user or recovery questions are not found, an appropriate error message is sent back .

The second endpoint (`/verify-answers`) verifies the user's answers to the recovery questions. It takes the user's email and submitted answers as input, retrieves the user's data from the database, and checks if the answers match the stored answers in the "recovery\_questions" collection. If the answers match, it responds with a success message and includes the user's password; otherwise, it indicates that the answers were incorrect. Both endpoints handle any potential errors by logging them and sending a generic error message to the client.

Server side:

// Endpoint to handle forgot password

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

const { email } = req.body;

try {

const user = await db.collection('users').findOne({ email });

if (!user) {

return res.status(404).send('Email not found');

}

const recoveryData = await db.collection('recovery\_questions').findOne({ user\_id: user.\_id });

if (!recoveryData) {

return res.status(404).send('Recovery questions not found');

}

res.json({ questions: { question1: recoveryData.question1, question2: recoveryData.question2 } });

} catch (err) {

console.error('Error fetching user or recovery questions:', err);

return res.status(500).send('Error fetching user or recovery questions');

}

});

// Endpoint to verify recovery answers

app.post('/verify-answers', async (req, res) => {

const { email, answers } = req.body;

try {

const user = await db.collection('users').findOne({ email });

if (!user) {

return res.status(404).send('Email not found');

}

const password = user.password;

const recoveryData = await db.collection('recovery\_questions').findOne({ user\_id: user.\_id });

if (!recoveryData) {

return res.status(404).send('Recovery answers not found');

}

const { answer1, answer2 } = recoveryData;

if (answers.answer1 === answer1 && answers.answer2 === answer2) {

res.json({ correct: true, password });

} else {

res.json({ correct: false });

}

} catch (err) {

console.error('Error fetching user or recovery answers:', err);

return res.status(500).send('Error fetching user or recovery answers');

}

});

Client side:

const handleSubmitEmail = async (e) => {

    e.preventDefault();

    try {

      const response = await axios.post('/forgot-password', { email });

      if (response.data.questions) {

        setQuestions(response.data.questions);

        setStep(2);

      } else {

        setError('Email not found.');

      }

    } catch (error) {

      setError('Error retrieving recovery questions.');

    }

  };

const handleSubmitAnswers = async (e) => {

    e.preventDefault();

    try {

      const response = await axios.post('/verify-answers', { email, answers });

      if (response.data.correct) {

        alert(`Your password is: ${response.data.password}`);

        setMessage('Password has been sent to your email.');

        setStep(3);

      } else {

        setError('Incorrect answers. Please try again.');

      }

    } catch (error) {

      setError('Error verifying answers.');

    }

  };

Fetch Data (Home Page)

The code provides an endpoint (`/fetch-data`) to retrieve a user's friends' posts, along with their associated comments, likes, and user details. It begins by extracting the `userId`, `limit`, and `skip` parameters from the query string to manage which user’s data to fetch and control pagination. The `userId` is validated to ensure it is correctly formatted.

Next, it fetches the list of the user's friends' IDs from the `friends` collection. If the user has no friends, it returns an empty response with no data. If friends are found, it retrieves the posts made by these friends from the `posts` collection, sorts them by the date in descending order, and applies pagination using the `skip` and `limit` parameters.

After fetching the posts, the code retrieves the associated comments and likes for these posts from their respective collections. It also gathers the user details related to the posts to provide complete information. The comments are then formatted to rename the `\_id` field to `comment\_id` for clarity.

Finally, the endpoint responds with a JSON object containing the posts, formatted comments, likes, and user details. If an error occurs at any point, it logs the error and returns a message indicating the failure to fetch data.

app.get('/fetch-data', async (req, res) => {

try {

const { userId, limit = 10, skip = 0 } = req.query; *// Get userId, limit, and skip from query parameters*

*// Ensure the userId is valid*

if (!userId || !ObjectId.isValid(userId)) {

return res.status(400).send({ message: 'Invalid user ID' });

}

*// Fetch the user's friends' IDs as strings*

const friends = await db.collection('friends').find({ user\_id: new ObjectId(userId) }).toArray();

const friendIds = friends.map(friend => friend.friend\_id.toString()); *// Convert ObjectId to string*

if (friendIds.length === 0) {

*// No friends, return empty data*

return res.json({ posts: [], comments: [], likes: [], users: [] });

}

*// Fetch posts by friends, sorted by date, and limited by pagination*

const posts = await db.collection('posts')

.find({ user\_id: { $in: friendIds } }) *// Compare with friend IDs*

.sort({ post\_date: -1 })

.skip(parseInt(skip)) *// Skip a number of documents (for pagination)*

.limit(parseInt(limit)) *// Limit the number of documents to fetch*

.toArray();

const postIds = posts.map(post => new ObjectId(post.\_id)); *// Convert to ObjectId for matching*

*// Fetch comments and likes only for the posts that are being returned*

const comments = await db.collection('comments').find({ post\_id: { $in: postIds } }).toArray();

const likes = await db.collection('likes').find({ post\_id: { $in: postIds } }).toArray();

*// Fetch the users associated with these posts*

const userIds = [...new Set(posts.map(post => post.user\_id))]; *// Unique user IDs from posts*

const users = await db.collection('users').find({ \_id: { $in: userIds.map(id => new ObjectId(id)) } }).toArray();

*// Map through comments to rename \_id to comment\_id*

const formattedComments = comments.map(comment => ({

...comment,

comment\_id: comment.\_id.toString(), *// Assign \_id to comment\_id and convert to string*

\_id: undefined, *// Optionally remove \_id to avoid confusion*

}));

res.json({ posts, comments: formattedComments, likes, users });

} catch (err) {

console.error('Error fetching data:', err);

res.status(500).send({ message: 'Failed to fetch data' });

}

});

Likes :

The code provides functionality for managing likes and notifications in a social media application. It includes functions to add and remove like notifications (`addLikeNotification` and `removeLikeNotification`), which create notifications in the `notifications` collection when a user likes or unlikes a post. The `/add-like` endpoint allows users to like a post if they haven't already done so, updates the like count in the `posts` collection, and triggers a notification for the post owner. Similarly, the `/remove-like` endpoint removes a like from a post if the user had previously liked it, adjusts the like count, and adds an unlike notification. The `checkIfLiked` function, along with the `/check-if-liked` endpoint, checks if a user has already liked a particular post and returns the result. Overall, the code handles these interactions and notifications efficiently, providing clear feedback and managing errors effectively.

async function addLikeNotification(postId, currentUser) {

try {

const post = await db.collection('posts').findOne({ \_id: new ObjectId(postId) });

if (!post) {

throw new Error('Post not found');

}

const notification = {

user\_id: post.user\_id,

notification\_content: `${currentUser.user\_name} has liked your post`,

seen: false,

date: new Date()

};

await db.collection('notifications').insertOne(notification);

console.log('Like notification added successfully');

} catch (err) {

console.error('Error adding like notification:', err);

}

}

*// Function to add unlike notification ++++++++++++++++++*

async function removeLikeNotification(postId, currentUser) {

try {

const post = await db.collection('posts').findOne({ \_id: new ObjectId(postId) });

if (!post) {

throw new Error('Post not found');

}

const notification = {

user\_id: post.user\_id,

notification\_content: `${currentUser.user\_name} has removed the like from your post`,

seen: false,

date: new Date()

};

await db.collection('notifications').insertOne(notification);

console.log('Unlike notification added successfully');

} catch (err) {

console.error('Error adding unlike notification:', err);

}

}

*// Endpoint to add like to a post*

app.post('/add-like', async (req, res) => {

const { post\_id, user } = req.body;

const user\_id = user.user\_id;

try {

const existingLike = await db.collection('likes').findOne({ post\_id: new ObjectId(post\_id), user\_id: new ObjectId(user\_id) });

if (existingLike) {

res.status(400).send({ message: 'User has already liked this post' });

} else {

*// User has not liked the post, add like*

await db.collection('likes').insertOne({ post\_id: new ObjectId(post\_id), user\_id: new ObjectId(user\_id) });

await db.collection('posts').updateOne({ \_id: new ObjectId(post\_id) }, { $inc: { likes\_num: 1 } });

*// Add like notification*

await addLikeNotification(post\_id, user);

res.json({ message: 'Like added successfully' });

}

} catch (err) {

console.error('Error adding like:', err);

res.status(500).send({ message: 'Failed to add like' });

}

});

*// Function to check if the current user liked a post*

async function checkIfLiked(post\_id, user\_id) {

try {

const existingLike = await db.collection('likes').findOne({

post\_id: new ObjectId(post\_id),

user\_id: new ObjectId(user\_id)

});

return !!existingLike; *// Return true if a like exists, false otherwise*

} catch (err) {

console.error('Error checking if liked:', err);

throw err;

}

}

*// Endpoint to check if the current user liked a post*

app.post('/check-if-liked', async (req, res) => {

const { post\_id, user } = req.body;

const user\_id = user.user\_id;

try {

const hasLiked = await checkIfLiked(post\_id, user\_id);

res.json({ liked: hasLiked });

} catch (err) {

res.status(500).send({ message: 'Failed to check if liked' });

}

});

*// Endpoint to remove like from a post*

app.post('/remove-like', async (req, res) => {

const { post\_id, user } = req.body;

const user\_id = user.user\_id;

try {

const existingLike = await db.collection('likes').findOne({ post\_id: new ObjectId(post\_id), user\_id: new ObjectId(user\_id) });

if (!existingLike) {

res.status(400).send({ message: 'User has not liked this post' });

} else {

*// User already liked the post, remove like*

await db.collection('likes').deleteOne({ post\_id: new ObjectId(post\_id), user\_id: new ObjectId(user\_id) });

await db.collection('posts').updateOne({ \_id: new ObjectId(post\_id) }, { $inc: { likes\_num: -1 } });

*// Add unlike notification*

await removeLikeNotification(post\_id, user);

res.json({ message: 'Like removed successfully' });

}

} catch (err) {

console.error('Error removing like:', err);

res.status(500).send({ message: 'Failed to remove like' });

}

});

Comments :  
  
  
The code provides functionality for managing comments and related notifications in a social media application. It includes functions to create notifications when a user adds or deletes a comment on a post. The `addCommentNotification` function generates a notification when a new comment is added, while the `addDeleteCommentNotification` function does the same when a comment is deleted. The `/add-comment` endpoint handles adding a new comment by inserting it into the database, updating the post's comment count, and notifying the post owner. Similarly, the `/delete-comment` endpoint allows users to delete a comment by verifying its existence, removing it from the database, updating the post's comment count, and notifying the post owner of the deletion. This approach ensures efficient comment management and user engagement by keeping them informed about actions on their posts.  
  
  
  
async function addCommentNotification(postId, currentUser, commentContent) {

try {

const post = await db.collection('posts').findOne({ \_id: new ObjectId(postId) });

if (!post) {

throw new Error('Post not found');

}

const notification = {

user\_id: post.user\_id,

notification\_content: `${currentUser.user\_name} added a comment to your post: ${commentContent}`,

seen: false,

date: new Date()

};

await db.collection('notifications').insertOne(notification);

console.log('Comment notification added successfully');

} catch (err) {

console.error('Error adding comment notification:', err);

}

}

*// Endpoint to add a new comment and notification*

app.post('/add-comment', async (req, res) => {

const { post\_id, user\_id, comment\_content, comment\_date, user } = req.body;

try {

await db.collection('comments').insertOne({

post\_id: new ObjectId(post\_id),

user\_id: new ObjectId(user\_id),

comment\_content,

comment\_date

});

await db.collection('posts').updateOne(

{ \_id: new ObjectId(post\_id) },

{ $inc: { comments\_num: 1 } }

);

*// Add the notification*

await addCommentNotification(post\_id, user, comment\_content);

res.json({ message: 'Comment added successfully' });

} catch (err) {

console.error('Error adding comment:', err);

res.status(500).send({ message: 'Failed to add comment' });

}

});

*// Function to add a notification for deleting a comment*

async function addDeleteCommentNotification(postId, currentUser, commentContent) {

try {

const post = await db.collection('posts').findOne({ \_id: new ObjectId(postId) });

if (!post) {

throw new Error('Post not found');

}

const notification = {

user\_id: post.user\_id, *// Notification goes to the user who created the post*

notification\_content: `${currentUser.user\_name} has deleted the comment: "${commentContent}" from your post`,

seen: false,

date: new Date()

};

await db.collection('notifications').insertOne(notification);

console.log('Delete comment notification added successfully');

} catch (err) {

console.error('Error adding delete comment notification:', err);

}

}

*// Helper function to check if a string is a valid ObjectId*

function isValidObjectId(id) {

return ObjectId.isValid(id) && new ObjectId(id).toString() === id;

}

app.post('/delete-comment', async (req, res) => {

const { post\_id, comment\_id } = req.body;

try {

console.log('Request received to delete comment:', { post\_id, comment\_id });

*/\**

*if (!ObjectId.isValid(post\_id) || !ObjectId.isValid(comment\_id)) {*

*console.error('Invalid post ID or comment ID');*

*return res.status(400).send('Invalid post ID or comment ID');*

*}*

*\*/*

*//const postObjectId = new ObjectId(post\_id);*

const commentObjectId = new ObjectId(comment\_id);

const comment = await db.collection('comments').findOne({ \_id: commentObjectId });

if (!comment) {

console.error('Comment not found:', commentObjectId);

return res.status(404).send('Comment not found');

}

await db.collection('comments').deleteOne({ \_id: commentObjectId });

console.log('Comment deleted:', commentObjectId);

console.log('post\_id: '+new ObjectId(post\_id));

const postUpdateResult = await db.collection('posts').findOneAndUpdate(

{ \_id: new ObjectId(post\_id) },

{ $inc: { comments\_num: -1 } },

{ returnDocument: 'after' }

);

console.log(postUpdateResult.value);

if (!postUpdateResult.value) {

console.error('Post not found:', new ObjectId(post\_id));

return res.status(404).send('Post not found');

}

console.log('Post updated:', new ObjectId(post\_id));

try {

await addDeleteCommentNotification(post\_id, { user\_name: req.body.current\_user\_name }, comment.comment\_content);

console.log('Delete comment notification added');

} catch (error) {

console.error('Failed to add delete comment notification, but comment was deleted:', error);

}

res.json({ message: 'Comment deleted successfully', post: postUpdateResult.value });

} catch (error) {

console.error('Error deleting comment:', error);

res.status(500).send('Failed to delete comment');

}

});

async function addDeleteCommentNotification(postId, currentUser, commentContent) {

try {

const post = await db.collection('posts').findOne({ \_id: new ObjectId(postId) });

if (!post) throw new Error('Post not found');

const notification = {

user\_id: post.user\_id,

notification\_content: `${currentUser.user\_name} has deleted the comment: "${commentContent}" from your post`,

seen: false,

date: new Date(),

};

await db.collection('notifications').insertOne(notification);

} catch (error) {

console.error('Error adding delete comment notification:', error);

}

}

Followers :

The code manages follow and unfollow actions between users in a social media application, along with the corresponding notifications. The `addFollowNotification` function creates a notification when a user starts following another user. The `/follow` endpoint allows a user to follow another by creating a relationship in the `friends` collection, fetching the friend's information, and generating a follow notification. Similarly, the `/remove-follow` endpoint handles unfollowing by deleting the relationship from the `friends` collection and creating an "unfollow" notification if the deletion is successful. Additionally, the `/remove-follower` endpoint manages removing a follower by deleting the follower relationship and returning a success message or error if the follower is not found. These functionalities ensure proper handling of follow/unfollow actions and keep users informed through notifications.  
*// Function to add follow notification*

async function addFollowNotification(userId, currentUser) {

try {

const notification = {

user\_id: new ObjectId(userId),

notification\_content: `${currentUser.user\_name} started to follow you`,

seen: false,

date: new Date()

};

await db.collection('notifications').insertOne(notification);

console.log('Follow notification added successfully');

} catch (err) {

console.error('Error adding follow notification:', err);

throw err;

}

}

*// Endpoint to add a friend and create notification*

app.post('/follow', async (req, res) => {

const { user\_id, friend\_id, currentUser } = req.body;

try {

*// Add friend relationship*

await db.collection('friends').insertOne({ user\_id: new ObjectId(user\_id), friend\_id: new ObjectId(friend\_id) });

*// Fetch friend's information*

const friend = await db.collection('users').findOne({ \_id: new ObjectId(friend\_id) }, { projection: { user\_id: 1, user\_name: 1 } });

if (!friend) {

return res.status(404).send('Friend not found');

}

*// Add follow notification*

await addFollowNotification(friend\_id, currentUser);

res.json({ message: 'Friend added successfully', friend: { user\_id: friend.\_id, user\_name: friend.user\_name } });

} catch (err) {

console.error('Error adding friend:', err);

res.status(500).send('Error adding friend');

}

});

app.post('/remove-follow', async (req, res) => {

const { user\_id, friend\_id, currentUser } = req.body;

try {

*// Remove friend relationship*

const result = await db.collection('friends').deleteOne({ user\_id: new ObjectId(user\_id), friend\_id: new ObjectId(friend\_id) });

if (result.deletedCount === 1) {

await addUnfollowNotification(friend\_id, currentUser);

res.json({ message: 'Friend removed successfully' });

} else {

res.status(404).send('Friend not found');

}

} catch (err) {

console.error('Error removing friend:', err);

res.status(500).send('Error removing friend');

}

});

app.post('/remove-follower', async (req, res) => {

const { user\_id, follower\_id } = req.body;

console.log('Request body:', req.body); *// Log the entire request body*

console.log(`Attempting to remove follower: follower\_id=${follower\_id} from user\_id=${user\_id}`);

try {

*// Convert IDs to ObjectId if they are not already*

const userIdObj = new ObjectId(user\_id);

const followerIdObj = new ObjectId(follower\_id);

*// Remove the follower relationship*

const result = await db.collection('friends').deleteOne({ user\_id: followerIdObj, friend\_id: userIdObj });

if (result.deletedCount === 1) {

console.log(`Successfully removed follower: follower\_id=${follower\_id} from user\_id=${user\_id}`);

res.json({ message: 'Follower removed successfully' });

} else {

console.log(`Follower not found: follower\_id=${follower\_id} from user\_id=${user\_id}`);

res.status(404).send('Follower not found');

}

} catch (err) {

console.error(`Error removing follower: follower\_id=${follower\_id} from user\_id=${user\_id}`, err);

res.status(500).send('Error removing follower');

}

});

websocket code :  
  
  
The code sets up a WebSocket server that listens for incoming connections from clients, allowing real-time communication between them. It uses the WebSocket library and runs on a port specified by an environment variable or defaults to port 50246. When a new client connects, a message is logged to the console. The server listens for messages from any connected client and broadcasts each received message to all other connected clients, excluding the sender. It checks that each client is still open before sending the message. Additionally, when a client disconnects, a message is logged indicating that the client has disconnected. This setup facilitates a simple chat or real-time communication system where multiple clients can send and receive messages.

const WebSocket = require('ws');

*// Use an environment variable for the port or default to 8080*

const port = process.env.PORT || 50246;

const wss = new WebSocket.Server({ port });

wss.on('connection', (ws, req) => {

console.log('New WebSocket client connected');

ws.on('message', (message) => {

console.log(`Received message => ${message}`);

*// Broadcast the message to all clients except the sender*

wss.clients.forEach((client) => {

if (client !== ws && client.readyState === WebSocket.OPEN) {

client.send(message);

console.log(`Sent message to client: ${client.\_socket.remoteAddress}`);

}

});

});

ws.on('close', () => {

console.log('WebSocket client disconnected');

});

});

console.log(`WebSocket server running on port ${port}`);

### 1. **Login Page**

### The Login page allows users to access their accounts by entering their credentials. It typically consists of input fields for the user’s **email** and **password** and a **"Login"** button to submit these credentials. Additional elements may include a **"Forgot Password?"** link that redirects to the Reset Password page if the user has trouble remembering their password, and a **"Register"** link to direct new users to the registration page. The frontend handles user input, provides real-time feedback (like validation errors), and sends a request to the backend to verify the user's credentials. 2. **Register Page**

The Register page enables new users to create an account. It generally includes fields for the user's **username**, **first name**, **last name**, **email**, **password**, and **confirm password**. Additional fields might include **gender**, **phone number**, **profile photo**, **education**, and **skills**. The form may also prompt the user to set up **security questions** for future password recovery. The frontend ensures all input fields are properly validated (e.g., matching passwords, valid email format) and provides feedback to the user in real-time. Upon form submission, the data is sent to the backend to create a new user account, and the user may receive a success or error message based on the server response.

### 3. **Reset Password Page**

### The Reset Password page assists users in recovering their accounts when they forget their password. Initially, it may prompt the user to enter their **registered email** to receive a recovery link or code. If security questions are set up, the page might present these questions for the user to answer. Once the email or security questions are verified, the page typically transitions to allow the user to **enter a new password** and **confirm** it. The frontend handles validating the new password requirements (like minimum length and complexity) and ensures both password entries match. After submission, it sends the information to the backend to update the user's password and displays a success or error message depending on the server's response. 1. **Homepage Layout**

* The homepage features a welcoming message ("Welcome to Your Professional Network") prominently displayed at the top.
* Below the welcome message, there is a text input area labeled "What's on your mind?" allowing users to post their thoughts or updates. There is a "Post" button to submit the content.
* A "Load More" button at the bottom suggests the ability to fetch more posts dynamically, indicating pagination or infinite scrolling functionality.

### 2. **Navigation Sidebar**

* A left-aligned sidebar provides navigation links to various sections such as **Home**, **Profile**, **Settings**, **Notifications**, **Messages**, **Friends**, and **Logout**.
* A **Dark Mode** toggle button is present, allowing users to switch between light and dark themes for better accessibility and personal preference.

### 3. **Post Interaction Features**

* The feed displays posts from different users. Each post shows the user’s name, their post content, and interaction options like **Likes** and **Comments**.
* Users can add a comment by typing in the provided input field and clicking the **Comment** button. There is also a **Like** button for liking posts.
* The number of likes and comments is displayed below each post, providing users with a quick overview of engagement.

### 4. **Comments and Likes Pop-ups**

* Clicking on the **Comments** or **Likes** links opens a side panel displaying the details. The **Comments** panel shows user comments along with their usernames, while the **Likes** panel lists all users who have liked a particular post.
* Both panels include a **Close** button to exit the view, enhancing the user experience by keeping interactions smooth and non-intrusive.

### 5. **Dark Mode View**

* The platform supports a **Dark Mode** view, which can be toggled from the sidebar. In the dark mode, the background changes to a dark color, and the text and buttons adapt accordingly to maintain readability and comfort for users in low-light environments.

A screen shot of a login screen

Description automatically generated

זה העמוד הראשון שלנו ממנו את יכול לכנס לחשבון שלך אם יש לך אחד או לייצר אחד דרך לחיצה על כפתור יצירת חשבון וגם אם שכחתה את הסיסמה אתה יכול להכניס את המייל שלך ותענה על השאלות ואתה תקבל אותה

A screenshot of a phone

Description automatically generatedA screenshot of a phone number

Description automatically generated

זה דף יצירת חשבון שלנו את נכנס ממלה את הפרטים שלך ואם אתה רוצה יכול גם להעלאת תמונת פרופיל או לקחת אותה ישירות רק תלחץ על כפתור לפתיחת מצלמה וכפתור הבא הוא עובד גם לעבור לדף הבא וגם לקחת תמונה

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated  
  
  
  
  
A screenshot of a computer

Description automatically generated

בא זה הדף הראשי שלנו אתה יכול לראות את הפוסטים של חברים שלך וגם לתת להם ליקס ו קומינת על הפוסטים שלהם ויש לך גם את המיניו שממנו את יכול לעבור לשאר הדפים  
בדף ברופיל אתה יכול לראות את הפרטים שלך בדף ההגדרות אתה ישול לשנות אותם גם במניו יש לך דף לחיפוש חברים חדשים ולעשות להם עוקב ולדבר איתם וגם במנניו אתה יכול לשנות ממצב שחור ל אור

אנחנו נכנסים למניו הזה דרך לחיצה על כפתור ליק או קומינת איפה שיש מספר כדי לראות מי עשה לכם לייקים ו קומיתז

בו זה דף ההתראות אם מישהוא עשה ליק או משהוא אחר רתה יכול לראות את זה בו