SECURITE SYSTEME EXPLOITATION SMARTPHONES

Cybersecurity Analysis Project Report

Filière: 2ème année cyber security

Semestre: 3

Module: SECURITE SYSTEME EXPLOITATION SMARTPHONES

Supervised by:
Pr. TMIMI Mehdi

Prepared by:
EL AMRANI Yassine

SUMMARY

Introduction :

Web applications are an integral part of modern computing, enabling businesses and organizations to offer services and engage with users over the internet. However, as the reliance on web applications grows, so does the threat landscape. Cyberattacks targeting web applications have become increasingly sophisticated, with attackers exploiting vulnerabilities to gain unauthorized access, steal sensitive information, or disrupt operations.

This project focuses on analyzing the security of a Node.js-based task management web application. The application is intentionally built with several security flaws to serve as a learning platform for understanding common vulnerabilities and applying mitigation techniques. The goal of the analysis is not only to identify security issues but also to demonstrate the importance of adopting secure coding practices to safeguard web applications against potential threats.

The application includes basic features such as user registration, login functionality, and task management. While these features are commonplace in many applications, they are also often targeted by attackers. For example, weak password storage, improper session management, and lack of input validation can lead to serious security breaches. This project offers a comprehensive security audit of these components, with the intention of educating developers on how to recognize and address common vulnerabilities.

The focus of this report is to evaluate the application's codebase, identify vulnerabilities, and suggest appropriate remedies to improve both the security and reliability of the system. By addressing these weaknesses, the application will be better protected against a range of attacks, including SQL injection, session fixation, brute-force attacks, and data tampering.
Ultimately, the findings and recommendations from this project aim to promote best practices in web application security. With an increasing amount of sensitive data being stored and transmitted online, ensuring the security of web applications is crucial to maintaining trust and protecting users from malicious activities.

Partie Théorique

2.1 Overview of Web Application Security

Web application security is the practice of safeguarding websites and web applications from various forms of cyberattacks. As more businesses move their operations online, web applications have become a prime target for cybercriminals. Attacks on web applications can result in severe consequences, including data breaches, financial loss, damage to reputation, and regulatory fines.
Web applications interact with databases, servers, and external services, making them inherently vulnerable to different types of attacks. Thus, it is essential to follow secure development practices to prevent security flaws from being introduced during development.

2.2 Common Web Application Vulnerabilities

Web applications can be susceptible to a wide range of vulnerabilities, many of which are well-documented by organizations such as the Open Web Application Security Project (OWASP). Some of the most common vulnerabilities include:

2.2.1 Injection Attacks

Injection attacks, such as SQL injection and NoSQL injection, occur when an attacker is able to insert malicious code into a query or command executed by the application. These attacks can give attackers access to the application's underlying database, allowing them to retrieve, modify, or delete sensitive data.
Mitigation:

- *Use parameterized queries or prepared statements to prevent malicious data from being executed as code.*
- *Implement input validation and sanitization.*

2.2.2 Authentication and Session Management Vulnerabilities

Improper handling of authentication and session management can lead to various attacks, including session fixation, session hijacking, and unauthorized access. Insecure password storage, such as storing passwords in plaintext, is also a significant vulnerability.
Mitigation:

- *Use strong hashing algorithms (e.g., bcrypt, Argon2) for password storage.*
- *Implement session regeneration upon login to prevent session fixation.*
- *Ensure proper session expiration and use secure cookies for session management.*

Partie Théorique

2.2 Common Web Application Vulnerabilities

2.2.3 Cross-Site Request Forgery (CSRF)

CSRF attacks force authenticated users to perform unintended actions on a website without their knowledge. This occurs when an attacker tricks the user into making an unwanted request, such as changing their email address or initiating a money transfer.
Mitigation:

- *Use anti-CSRF tokens that validate the authenticity of requests.*
- *Implement same-site cookie attributes to prevent the browser from sending cookies with cross-site requests.*

2.2.4 Cross-Site Scripting (XSS)

XSS occurs when an attacker injects malicious scripts into a website's content, which is then executed by the browser of an unsuspecting user. This can lead to cookie theft, session hijacking, and redirecting users to malicious websites.
Mitigation:

- *Sanitize user input by escaping characters that could be interpreted as code.*
- *Use Content Security Policy (CSP) headers to limit the execution of malicious scripts.*

2.2.5 Insecure Direct Object References (IDOR)

IDOR vulnerabilities occur when an attacker is able to access or modify resources that they are not authorized to, such as accessing someone else's files or tasks.
Mitigation:

- *Perform access control checks to ensure users can only access their own resources.*
- *Avoid exposing internal resource IDs in URLs or responses.*

Partie Théorique

2.3 Secure Coding Practices

To mitigate the risks associated with these vulnerabilities, developers should follow secure coding practices. These practices aim to prevent the introduction of vulnerabilities during the development phase and reduce the overall attack surface of an application.

2.3.1 Input Validation and Sanitization

Input validation ensures that only correctly formatted data is accepted by the application. All inputs, whether from forms, APIs, or URL parameters, should be validated to prevent injection attacks and ensure that they conform to the expected type, length, and format.
Best Practices:

- *Use whitelisting (allow only known good inputs) rather than blacklisting.*
- *Sanitize inputs by removing any potentially dangerous characters (e.g., <, >, ;).*

2.3.2 Authentication and Authorization

Strong authentication mechanisms should be used to ensure that users are who they claim to be, while authorization mechanisms should control access to specific resources based on user roles.
Best Practices:

- *Use multi-factor authentication (MFA) for critical applications.*
- *Implement role-based access control (RBAC) to ensure users only have access to resources they are authorized to.*

2.3.3 Secure Session Management

Sessions should be managed securely to prevent unauthorized users from gaining access. Sessions should be invalidated after logout or a period of inactivity, and session cookies should be configured with security flags to prevent hijacking.
Best Practices:

- *Use secure, httpOnly, and sameSite cookie attributes.*
- *Implement session timeout and regeneration mechanisms after login.*

2.3.4 Error Handling and Logging

Error messages should not reveal sensitive information, such as stack traces or database details. Proper error handling ensures that an application can gracefully recover from unexpected situations without exposing vulnerabilities.
Best Practices:

- *Log errors for internal monitoring but avoid exposing stack traces in production environments.*
- *Implement custom error messages that provide helpful feedback without compromising security.*

Partie Théorique

2.4 Security Testing and Vulnerability Assessment

Once the application has been developed, it is essential to conduct security testing to identify potential vulnerabilities. Some common methods of security testing include:

2.4.1 Static Application Security Testing (SAST)

SAST tools analyze the source code of an application to identify potential vulnerabilities before the code is executed. This type of testing is typically done early in the development process.

2.4.2 Dynamic Application Security Testing (DAST)

DAST tools test the application in a running state to simulate real-world attacks, such as attempting to exploit an XSS or SQL injection vulnerability.

2.4.3 Penetration Testing

Penetration testing involves ethical hackers attempting to exploit the application in the same way a malicious attacker would. This type of testing helps identify both known and unknown vulnerabilities.

Partie Pratique

Task 1: Code Analysis – Security Vulnerabilities and Error Handling Issues

To assess the security of the provided Node.js application, we reviewed its codebase and identified multiple vulnerabilities and areas where error handling is insufficient. Below is a structured analysis of these issues along with recommended solutions.

1. Password Storage Vulnerability

Issue:

- The application stores passwords in plain text inside database.json.
- If this file is compromised, all user credentials can be easily accessed.

Solution:

- Use bcrypt to hash passwords before storing them in the database.

Example Fix (Registration Process)

```
const bcrypt = require('bcrypt');
const saltRounds = 10;

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

- This ensures that even if the database is leaked, passwords remain unreadable.

2. Lack of Input Validation on User Registration and Login

Issue:

- The application does not validate user inputs, making it vulnerable to SQL Injection, NoSQL Injection, and other attacks.

Solution:

- Use express-validator to enforce validation rules.
- Require strong passwords (minimum length, uppercase, special characters).

Example Fix:

```
const { check, validationResult } = require('express-validator');

app.post('/register', [
    check('username').notEmpty().withMessage('Username is required'),
    check('password').isLength({ min: 8 }).withMessage('Password must be at least 8 characters long')
Tabnine | Edit | Test | Explain | Document
], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    // Proceed with user creation
});
```

Partie Pratique

Task 1: Code Analysis – Security Vulnerabilities and Error Handling Issues

4. No Brute Force Protection

Issue:

- There is no rate limiting, allowing attackers to attempt multiple login attempts.

Solution:

- Implement rate limiting using express-rate-limit.

Example Fix:

```
const rateLimit = require('express-rate-limit');
const loginLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 5, // Limit each IP to 5 login attempts per window
    message: "Too many login attempts, please try again later."
});
app.post('/login', loginLimiter, loginHandler);
```

5. Unsecured Static File Access

Issue:

- Files inside /protected are accessible directly via URL.

Solution:

- Restrict access by enforcing authentication.

Example Fix:

```
app.use('/protected', (req, res, next) => {
    if (!req.session.user) {
        return res.status(403).json({ msg: "Access Denied" });
    }
    next();
});
```

6. No CSRF Protection

Issue:

- The application is vulnerable to Cross-Site Request Forgery (CSRF), allowing attackers to trick users into executing unwanted actions.

Solution:

- Implement CSRF tokens using the csurf package.

Example Fix:

```
const csrf = require('csurf');
const csrfProtection = csrf({ cookie: true });
app.use(csrfProtection);
```

Partie Pratique

Task 1: Code Analysis – Security Vulnerabilities and Error Handling Issues

7. Missing Authorization Check for Task Deletion

Issue:

- Any authenticated user can delete any task, regardless of ownership.

Solution:

- Ensure only the task owner can delete their own tasks.

Example Fix:

```
if (task.user !== req.session.user.username) {
   return res.status(403).send('Unauthorized to delete this task');
}
```

8. Improper Error Handling

Issue:

- The application crashes if database.json is missing or malformed.

Solution:

- Implement try/catch error handling and validate JSON structure before accessing it.

9. No Secure Cookie Configuration

Issue:

- The session cookie is not protected with httpOnly, secure, and sameSite attributes.

Solution:

- Update session configuration to enforce security.

Example Fix:

```
app.use(session({
    secret: 'mysecretkey',
    resave: false,
    saveUninitialized: true,
    cookie: {
        httpOnly: true,
        secure: true,  // Enable only if using HTTPS
        sameSite: 'strict'
    }
}));
```

Partie Pratique

Task 1: Security Vulnerabilities in database.json

The file database.json also contains security risks, primarily due to improper user storage and missing validation.

1. Plaintext Password Storage

Issue:
- User passwords are stored in plain text.

Solution:
- Store hashed passwords instead.

Example Fix:

2. Lack of Unique User Identification

Issue:
- Users are stored by username only, making it easy for username collisions to occur.

Solution:
- Assign a unique userID to each user.

Example Fix:

3. Task Ownership is Not Enforced Securely

Issue:
- Tasks are linked to users by username instead of a unique ID.

Solution:
- Use userID instead of username.

Example Fix:

Partie Pratique

Task 1: Security Vulnerabilities in database.json

4. No Role-Based Access Control (RBAC)

Issue:

- All users have the same access level.

Solution:

- Introduce user roles (e.g., admin, user).

Example Fix:

```
{
  "users": [
    {
      "id": "a1b2c3d4",
      "login": "admin",
      "hash": "$2b$10$...",
      "role": "admin"
    },
    {
      "id": "b2c3d4e5",
      "login": "user1",
      "hash": "$2b$10$...",
      "role": "user"
    }
  ]
}
```

5. No Data Integrity Verification

Issue:

- The application does not check if database.json is valid before using it.

Solution:

- Implement JSON schema validation before reading/writing data.

Practical Part

Task 1: Security Vulnerabilities in database.json

app.js

```
const fs = require('fs');
const bcrypt = require('bcrypt');
const express = require('express');
const { v4: uuidv4 } = require('uuid');

const app = express();
const PORT = 3000;

app.use(express.json());

// Load database.json safely
Tabnine | Edit | Explain
const loadDatabase = () => {
    try {
        const data = fs.readFileSync('database.json', 'utf8');
        return JSON.parse(data);
    } catch (error) {
        console.error('Error loading database:', error);
        return { users: [], tasks: [] };
    }
};

// Save database.json safely
Tabnine | Edit | Explain
const saveDatabase = (db) => {
    try {
        fs.writeFileSync('database.json', JSON.stringify(db, null, 4));
    } catch (error) {
        console.error('Error saving database:', error);
    }
};

// Register user with hashed password
Tabnine | Edit | Test | Explain | Document
app.post('/register', async (req, res) => {
    let { login, password, role } = req.body;

    if (!login || !password) {
        return res.status(400).json({ error: 'Username and password are required' });
    }

    if (role !== 'user' && role !== 'admin') {
        role = 'user'; // Default to user if invalid role
    }

    const db = loadDatabase();
    const existingUser = db.users.find(user => user.login === login);

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

    const hash = await bcrypt.hash(password, 10);
    const newUser = { id: uuidv4(), login, hash, role };

    db.users.push(newUser);
    saveDatabase(db);

    res.status(201).json({ message: 'User registered successfully' });
});

// Login user and verify password
Tabnine | Edit | Test | Explain | Document
app.post('/login', async (req, res) => {
    const { login, password } = req.body;

    const db = loadDatabase();
    const user = db.users.find(user => user.login === login);

    if (!user) {
        return res.status(401).json({ error: 'Invalid username or password' });
    }

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

    if (!isMatch) {
        return res.status(401).json({ error: 'Invalid username or password' });
    }

    res.json({ message: 'Login successful', userID: user.id, role: user.role });
});
```

```
// Add new task with user authentication
Tabnine | Edit | Test | Explain | Document
app.post('/tasks', (req, res) => {
    const { task, userID } = req.body;

    if (!task || !userID) {
        return res.status(400).json({ error: 'Task and user ID are required' });
    }

    const db = loadDatabase();
    const user = db.users.find(user => user.id === userID);

    if (!user) {
        return res.status(403).json({ error: 'Invalid user ID' });
    }

    const newTask = { id: Date.now(), task, userID };
    db.tasks.push(newTask);
    saveDatabase(db);

    res.status(201).json({ message: 'Task added successfully' });
});

// Get all tasks for a specific user
Tabnine | Edit | Test | Explain | Document
app.get('/tasks/:userID', (req, res) => {
    const { userID } = req.params;

    const db = loadDatabase();
    const user = db.users.find(user => user.id === userID);

    if (!user) {
        return res.status(403).json({ error: 'Invalid user ID' });
    }

    const userTasks = db.tasks.filter(task => task.userID === userID);
    res.json(userTasks);
});

// Delete task with ownership validation
Tabnine | Edit | Test | Explain | Document
app.delete('/tasks/:taskID/:userID', (req, res) => {
    const { taskID, userID } = req.params;

    const db = loadDatabase();
    const user = db.users.find(user => user.id === userID);

    if (!user) {
        return res.status(403).json({ error: 'Invalid user ID' });
    }

    const taskIndex = db.tasks.findIndex(task => task.id == taskID && task.userID === userID);

    if (taskIndex === -1) {
        return res.status(404).json({ error: 'Task not found or unauthorized' });
    }

    db.tasks.splice(taskIndex, 1);
    saveDatabase(db);

    res.json({ message: 'Task deleted successfully' });
});

// Admin can delete any task
Tabnine | Edit | Test | Explain | Document
app.delete('/admin/tasks/:taskID/:adminID', (req, res) => {
    const { taskID, adminID } = req.params;

    const db = loadDatabase();
    const admin = db.users.find(user => user.id === adminID && user.role === 'admin');

    if (!admin) {
        return res.status(403).json({ error: 'Unauthorized' });
    }

    const taskIndex = db.tasks.findIndex(task => task.id == taskID);

    if (taskIndex === -1) {
        return res.status(404).json({ error: 'Task not found' });
    }

    db.tasks.splice(taskIndex, 1);
    saveDatabase(db);

    res.json({ message: 'Admin deleted the task successfully' });
});

// Start the server
Tabnine | Edit | Test | Explain | Document
app.listen(PORT, () => {
    console.log(`Server running on http://localhost:${PORT}`);
});
```

Partie Pratique

Task 1: Security Vulnerabilities in database.json

user.js


```json
[
    {
        "id": "unique-user-id",
        "login": "username",
        "hash": "hashed-password",
        "role": "user"
    },
    {
        "id": "fa631af7-38b9-4553-883e-148765bdbff6",
        "login": "ibrahim_3",
        "hash": "$2b$10$jPnSvnmXonzdoVb2ZWQZIOqXeGD4Asg2iKCMPSCuNIG0kL/wv0TMa",
        "role": "user"
    },
    {
        "id": "7177837b-006c-4edf-8a42-d0ec5442d879",
        "login": "Yassine-EL-amrani",
        "hash": "$2b$10$a.VN9g4XeAXtDtIMs1kc5u/LiCj8.f9Dppc/y2c44r3veDQti0jKe",
        "role": "user"
    }
]
```


task.js


```json
[
    {
        "id": 1,
        "task": "Example task",
        "userID": "unique-user-id"
    },
    {
        "id": "04020171-ef60-4a34-ab00-2159806db9a9",
        "task": "hi",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "3705d782-20b2-4db1-b380-eb470bd0fa8f",
        "task": "dn",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "cf8e4eda-0554-4c39-80ca-7f284ca59b2a",
        "task": "hello",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "f841730e-ddc4-4972-9f09-146dd536ae4f",
        "task": "dkk",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "b4cf7a1d-43f6-462c-a37d-f892fde4e456",
        "task": "cyber-security",
        "userID": "7177837b-006c-4edf-8a42-d0ec5442d879"
    },
    {
        "id": "ac6d70ef-e0f5-4e7f-8127-dce6a169e592",
        "task": "AI ing",
        "userID": "7177837b-006c-4edf-8a42-d0ec5442d879"
    }
]
```

Partie Pratique

Task 1: Security Vulnerabilities in database.json

user.js


```json
[
    {
        "id": "unique-user-id",
        "login": "username",
        "hash": "hashed-password",
        "role": "user"
    },
    {
        "id": "fa631af7-38b9-4553-883e-148765bdbff6",
        "login": "ibrahim_3",
        "hash": "$2b$10$jPnSvnmXonzdoVb2ZWQZIOqXeGD4Asg2iKCMPSCuNIG0kL/wv0TMa",
        "role": "user"
    },
    {
        "id": "7177837b-006c-4edf-8a42-d0ec5442d879",
        "login": "Yassine-EL-amrani",
        "hash": "$2b$10$a.VN9g4XeAXtDtIMs1kc5u/LiCj8.f9Dppc/y2c44r3veDQti0jKe",
        "role": "user"
    }
]
```


task.js

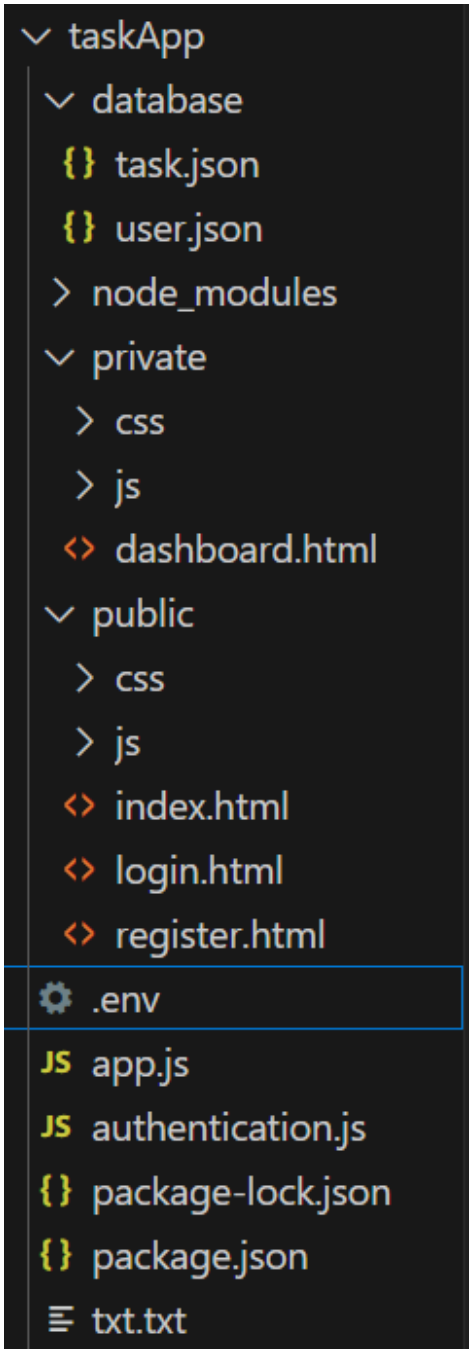

```json
[
    {
        "id": 1,
        "task": "Example task",
        "userID": "unique-user-id"
    },
    {
        "id": "04020171-ef60-4a34-ab00-2159806db9a9",
        "task": "hi",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "3705d782-20b2-4db1-b380-eb470bd0fa8f",
        "task": "dn",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "cf8e4eda-0554-4c39-80ca-7f284ca59b2a",
        "task": "hello",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "f841730e-ddc4-4972-9f09-146dd536ae4f",
        "task": "dkk",
        "userID": "fa631af7-38b9-4553-883e-148765bdbff6"
    },
    {
        "id": "b4cf7a1d-43f6-462c-a37d-f892fde4e456",
        "task": "cyber-security",
        "userID": "7177837b-006c-4edf-8a42-d0ec5442d879"
    },
    {
        "id": "ac6d70ef-e0f5-4e7f-8127-dce6a169e592",
        "task": "AI ing",
        "userID": "7177837b-006c-4edf-8a42-d0ec5442d879"
    }
]
```

Practical Part

Task 2: Development of Static Files

This task involves setting up the frontend structure of the task management system. The application is structured into public and private folders to differentiate between pages accessible to all users and those restricted to authenticated users.

Project Structure Explanation
The project follows a well-organized file structure to separate concerns efficiently:



Practical Part

Task 2: Development of Static Files

1. Public Folder (public/)

The public/ folder contains pages accessible to all users, including login, registration, and the initial index page.

🔹 1.1 login.html – User Login Page

Purpose:

- Provides a simple login form with fields for username and password.
- Uses main.js to send login requests to the backend via the Fetch API.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Main Dashboard</title>
    <link rel="stylesheet" href="css/main.css">
</head>
<body>
    <header>
        <h1>Welcome to Your Dashboard</h1>
    </header>
    <main>
        <section>
            <h2>Welcome, User!</h2>
            <p>Manage your tasks effectively with our platform.</p>
            <button onclick="window.location.href='login.html'">Login</button>
            <br><br>
            <button onclick="window.location.href='register.html'">Register</button>
        </section>
    </main>
</body>
</html>
```

🔹 1.2 register.html – User Registration Page

Purpose:

- Allows new users to create an account with username and password fields.
- Ensures password validation before sending data to the backend.

```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>
    <link rel="stylesheet" href="css/main.css">
</head>
<body>
    <header>
        <h1>Register</h1>
    </header>
    <main>
        <section>
            <h2>Register</h2>
            <form>
                <input id="loginInput" type="text" placeholder="Username" required>
                <input id="pwdInput" type="password" placeholder="Password" required>
                <button id="registerBtn">Register</button>
            </form>
        </section>
    </main>
    <script src="js/main.js"></script>
</body>
</html>
```

Practical Part

Task 2: Development of Static Files

🔹 1.3 index.html – Main Page (Redirects to Dashboard or Login)

Purpose:

- Acts as a landing page.
- Redirects authenticated users to dashboard.html, and unauthenticated users to login.html.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Main Dashboard</title>
    <link rel="stylesheet" href="css/main.css">
</head>
<body>
    <header>
        <h1>Welcome to Your Dashboard</h1>
    </header>
    <main>
        <section>
            <h2>Welcome, User!</h2>
            <p>Manage your tasks effectively with our platform.</p>
            <button onclick="window.location.href='login.html'">Login</button>
            <br><br>
            <button onclick="window.location.href='register.html'">Register</button>
        </section>
    </main>
</body>
</html>

```

🔹 **14. public/js/main.js – Frontend Logic for Authentication**

Purpose:

- Handles user login and registration using the Fetch API.
- Listens for form submissions and sends data to the backend.

Key Functionalities:

✓ Handles form submissions
✓ Sends login/registration requests using Fetch API
✓ Displays error messages if login/registration fails

```js
document.addEventListener('DOMContentLoaded', () => {
    // Registration functionality
    const registerBtn = document.getElementById('registerBtn');
    if (registerBtn) {
        registerBtn.addEventListener('click', async () => {
            const login = document.getElementById('loginInput').v
            const password = document.getElementById('pwdInput').

            if (!login || !password) {
                alert('Please fill in all fields');
                return;
            }

            try {
                const response = await fetch('/register', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json',
                    },
                    body: JSON.stringify({
                        login,
                        password,
                        role: 'user'
                    })
                });

                const data = await response.json();

                if (response.ok) {
                    alert('Registration successful!');
                    window.location.href = '/login.html';
                } else {
                    alert(data.error || 'Registration failed');
                }
            } catch (error) {
                console.error('Error:', error);
                alert('Registration failed. Please try again.');
            }
        });
    }
```

```js
    // Login functionality
    const loginForm = document.getElementById('loginForm');
    if (loginForm) {
        loginForm.addEventListener('submit', async (e) => {
            e.preventDefault(); // Prevent form from submitting normally

            const login = document.getElementById('login').value;
            const password = document.getElementById('pwd').value;

            try {
                const response = await fetch('/login', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json',
                    },
                    body: JSON.stringify({
                        login,
                        password
                    })
                });

                const data = await response.json();

                if (response.ok) {
                    // Store user info if needed
                    localStorage.setItem('userID', data.userID);
                    localStorage.setItem('userRole', data.role);

                    alert('Login successful!');
                    // Redirect to dashboard
                    window.location.href = '/app/dashboard.html';
                } else {
                    alert(data.error || 'Login failed');
                }
            } catch (error) {
                console.error('Error:', error);
                alert('Login failed. Please try again.');
            }
        });
    }
});
```

Practical Part

Task 2: Development of Static Files

🔹 1.3 index.html – Main Page (Redirects to Dashboard or Login)

Purpose:

- Acts as a landing page.
- Redirects authenticated users to dashboard.html, and unauthenticated users to login.html.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Main Dashboard</title>
    <link rel="stylesheet" href="css/main.css">
</head>
<body>
    <header>
        <h1>Welcome to Your Dashboard</h1>
    </header>
    <main>
        <section>
            <h2>Welcome, User!</h2>
            <p>Manage your tasks effectively with our platform.</p>
            <button onclick="window.location.href='login.html'">Login</button>
            <br><br>
            <button onclick="window.location.href='register.html'">Register</button>
        </section>
    </main>
</body>
</html>

```

🔹 **14. public/js/main.js – Frontend Logic for Authentication**

Purpose:

- Handles user login and registration using the Fetch API.
- Listens for form submissions and sends data to the backend.

Key Functionalities:

✓ Handles form submissions
✓ Sends login/registration requests using Fetch API
✓ Displays error messages if login/registration fails

```js
document.addEventListener('DOMContentLoaded', () => {
    // Registration functionality
    const registerBtn = document.getElementById('registerBtn');
    if (registerBtn) {
        registerBtn.addEventListener('click', async () => {
            const login = document.getElementById('loginInput').v
            const password = document.getElementById('pwdInput').

            if (!login || !password) {
                alert('Please fill in all fields');
                return;
            }

            try {
                const response = await fetch('/register', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json',
                    },
                    body: JSON.stringify({
                        login,
                        password,
                        role: 'user'
                    })
                });

                const data = await response.json();

                if (response.ok) {
                    alert('Registration successful!');
                    window.location.href = '/login.html';
                } else {
                    alert(data.error || 'Registration failed');
                }
            } catch (error) {
                console.error('Error:', error);
                alert('Registration failed. Please try again.');
            }
        });
    }
```

```js
    // Login functionality
    const loginForm = document.getElementById('loginForm');
    if (loginForm) {
        loginForm.addEventListener('submit', async (e) => {
            e.preventDefault(); // Prevent form from submitting normally

            const login = document.getElementById('login').value;
            const password = document.getElementById('pwd').value;

            try {
                const response = await fetch('/login', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json',
                    },
                    body: JSON.stringify({
                        login,
                        password
                    })
                });

                const data = await response.json();

                if (response.ok) {
                    // Store user info if needed
                    localStorage.setItem('userID', data.userID);
                    localStorage.setItem('userRole', data.role);

                    alert('Login successful!');
                    // Redirect to dashboard
                    window.location.href = '/app/dashboard.html';
                } else {
                    alert(data.error || 'Login failed');
                }
            } catch (error) {
                console.error('Error:', error);
                alert('Login failed. Please try again.');
            }
        });
    }
});
```

Practical Part

Task 2: Development of Static Files

2. Private Folder (private/)

The private/ folder contains files restricted to authenticated users.

🔹 2.1 dashboard.html – Task Management Dashboard

Purpose:

- Provides an interface for managing tasks.
- Includes task addition and task display sections.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0
    <title>Task Dashboard</title>
    <link rel="stylesheet" href="css/main.css">
    <script src="js/main.js" defer></script>
</head>
<body>
    <header>
        <h1>Task Manager</h1>
    </header>
    <main>
        <section>
            <h2>Add Task</h2>
            <input type="text" id="taskInput" placeholder="Enter task">
            <button id="addTaskBtn">Add Task</button>
        </section>
        <section>
            <h2>Task List</h2>
            <table>
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>Task</th>
                        <th>User</th>
                        <th>Actions</th>
                    </tr>
                </thead>
                <tbody id="taskList"></tbody>
            </table>
        </section>
    </main>
</body>
</html>
```

🔹 2.3 private/js/main.js – Task Management Logic

Purpose:

- Handles task addition, display, and deletion using Fetch API.
- Updates the task list dynamically using DOM manipulation.

Key Functionalities:

✔ Fetches existing tasks from the backend
✔ Adds new tasks to task.json
✔ Allows users to delete tasks

Practical Part

Task 2: Development of Static Files

```
// private/js/main.js
Tabnine | Edit | Test | Explain | Document
document.addEventListener("DOMContentLoaded", () => {
    // Get userID from localStorage
    const userID = localStorage.getItem('userID');
    if (!userID) {
        window.location.href = '/login.html';
        return;
    }

    // DOM Elements
    const taskInput = document.getElementById("taskInput");
    const addTaskBtn = document.getElementById("addTaskBtn");
    const taskList = document.getElementById("taskList");
    const userDisplay = document.getElementById("userDisplay");

    // Display username in header
    const displayUsername = () => {
        const username = localStorage.getItem('username');
        if (username && userDisplay) {
            userDisplay.textContent = `Welcome, ${username}!`;
        }
    };

    // Load Tasks from Server
    const loadTasks = async () => {
        try {
            const response = await fetch(`/tasks/${userID}`);
            if (!response.ok) throw new Error('Failed to load tasks');

            const tasks = await response.json();
            taskList.innerHTML = ""; // Clear existing tasks

            tasks.forEach(task => {
                const row = document.createElement("tr");
                row.innerHTML = `
                    <td>${task.id}</td>
                    <td>${task.task}</td>
                    <td>${task.username}</td>
                    <td>
                        <button class="delete-btn" data-id="${task.id}">Delete</button>
                    </td>
                `;
                taskList.appendChild(row);

                // Add delete event listener
                row.querySelector(".delete-btn").addEventListener("click", () =>
                    deleteTask(task.id, row)
                );
            });
        } catch (error) {
            console.error("Error loading tasks:", error);
            alert("Failed to load tasks. Please try again.");
        }
    };

    // Add Task
    addTaskBtn.addEventListener("click", async () => {
        const taskText = taskInput.value.trim();
        if (!taskText) {
            alert("Task cannot be empty!");
            return;
        }

        try {
            const response = await fetch('/tasks', {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify({
                    task: taskText,
                    userID: userID
                })
            })
        });

        if (!response.ok) throw new Error('Failed to add task');

        const result = await response.json();
        taskInput.value = ""; // Clear input
        loadTasks(); // Reload all tasks
    } catch (error) {
        console.error("Error adding task:", error);
        alert("Failed to add task. Please try again.");
    }
});

// Delete Task
const deleteTask = async (taskID, row) => {
    try {
        const response = await fetch(`/tasks/${taskID}/${userID}`, {
            method: "DELETE"
        });

        if (!response.ok) throw new Error('Failed to delete task');

        row.remove(); // Remove from UI
    } catch (error) {
        console.error("Error deleting task:", error);
        alert("Failed to delete task. Please try again.");
    }
};

// Logout functionality
const logoutBtn = document.getElementById('logoutBtn');
if (logoutBtn) {
    logoutBtn.addEventListener('click', async () => {
        try {
            await fetch('/logout', { method: 'POST' });
            localStorage.clear();
            window.location.href = '/login.html';
        } catch (error) {
            console.error('Logout error:', error);
            alert('Failed to logout. Please try again.');
        }
    });
}

// Initialize
displayUsername();
loadTasks();
});
```

Practical Part

Task 2: Development of Static Files

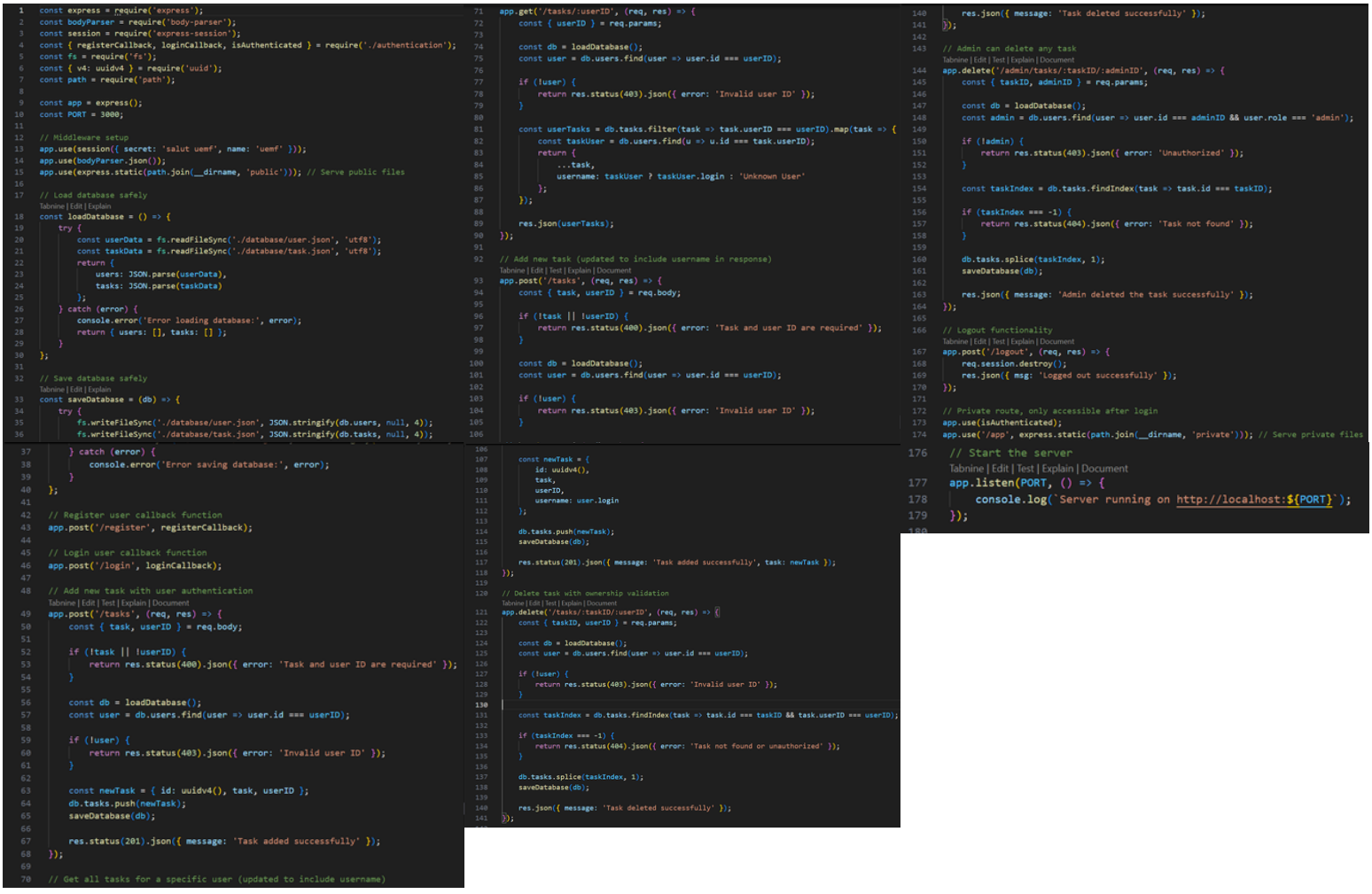
3. Backend Files (app.js & authentification.js)

These files handle user authentication and task management on the backend.

🔹 3.1 app.js – Main Backend File

Purpose:

- Handles routes for login, registration, and task management.
- Uses Express.js to create a REST API.



```
const express = require('express');
const bodyParser = require('body-parser');
const session = require('express-session');
const { registerCallback, loginCallback, isAuthenticated } = require('./authentication');
const fs = require('fs');
const { v4: uuidv4 } = require('uuid');
const path = require('path');

const app = express();
const PORT = 3000;

// Middleware setup
app.use(session({ secret: 'salut uemf', name: 'uemf' }));
app.use(bodyParser.json());
app.use(express.static(path.join(__dirname, 'public'))); // Serve public files

// Load database safely
Tabnine | Edit | Explain
const loadDatabase = () => {
    try {
        const userData = fs.readFileSync('./database/user.json', 'utf8');
        const taskData = fs.readFileSync('./database/task.json', 'utf8');
        return {
            users: JSON.parse(userData),
            tasks: JSON.parse(taskData)
        };
    } catch (error) {
        console.error('Error loading database:', error);
        return { users: [], tasks: [] };
    }
};

// Save database safely
Tabnine | Edit | Explain
const saveDatabase = (db) => {
    try {
        fs.writeFileSync('./database/user.json', JSON.stringify(db.users, null, 4));
        fs.writeFileSync('./database/task.json', JSON.stringify(db.tasks, null, 4));
    } catch (error) {
        console.error('Error saving database:', error);
    }
};

// Register user callback function
app.post('/register', registerCallback);

// Login user callback function
app.post('/login', loginCallback);

// Add new task with user authentication
Tabnine | Edit | Test | Explain | Document
app.post('/tasks', (req, res) => {
    const { task, userID } = req.body;

    if (!task || !userID) {
        return res.status(400).json({ error: 'Task and user ID are required' });
    }

    const db = loadDatabase();
    const user = db.users.find(user => user.id === userID);

    if (!user) {
        return res.status(403).json({ error: 'Invalid user ID' });
    }

    const newTask = { id: uuidv4(), task, userID };
    db.tasks.push(newTask);
    saveDatabase(db);

    res.status(201).json({ message: 'Task added successfully' });
});

// Get all tasks for a specific user (updated to include username)
app.get('/tasks/:userID', (req, res) => {
    const { userID } = req.params;

    const db = loadDatabase();
    const user = db.users.find(user => user.id === userID);

    if (!user) {
        return res.status(403).json({ error: 'Invalid user ID' });
    }

    const userTasks = db.tasks.filter(task => task.userID === userID).map(task => {
        const taskUser = db.users.find(u => u.id === task.userID);
        return {
            ...task,
            username: taskUser ? taskUser.login : 'Unknown User'
        };
    });

    res.json(userTasks);
});

// Add new task (updated to include username in response)
Tabnine | Edit | Test | Explain | Document
app.post('/tasks', (req, res) => {
    const { task, userID } = req.body;

    if (!task || !userID) {
        return res.status(400).json({ error: 'Task and user ID are required' });
    }

    const db = loadDatabase();
    const user = db.users.find(user => user.id === userID);

    if (!user) {
        return res.status(403).json({ error: 'Invalid user ID' });
    }

    const newTask = {
        id: uuidv4(),
        task,
        userID,
        username: user.login
    };

    db.tasks.push(newTask);
    saveDatabase(db);

    res.status(201).json({ message: 'Task added successfully', task: newTask });
});

// Delete task with ownership validation
Tabnine | Edit | Test | Explain | Document
app.delete('/tasks/:taskID/:userID', (req, res) => {
    const { taskID, userID } = req.params;

    const db = loadDatabase();
    const user = db.users.find(user => user.id === userID);

    if (!user) {
        return res.status(403).json({ error: 'Invalid user ID' });
    }

    const taskIndex = db.tasks.findIndex(task => task.id === taskID && task.userID === userID);

    if (taskIndex === -1) {
        return res.status(404).json({ error: 'Task not found or unauthorized' });
    }

    db.tasks.splice(taskIndex, 1);
    saveDatabase(db);

    res.json({ message: 'Task deleted successfully' });
});

// Admin can delete any task
Tabnine | Edit | Test | Explain | Document
app.delete('/admin/tasks/:taskID/:adminID', (req, res) => {
    const { taskID, adminID } = req.params;

    const db = loadDatabase();
    const admin = db.users.find(user => user.id === adminID && user.role === 'admin');

    if (!admin) {
        return res.status(403).json({ error: 'Unauthorized' });
    }

    const taskIndex = db.tasks.findIndex(task => task.id === taskID);

    if (taskIndex === -1) {
        return res.status(404).json({ error: 'Task not found' });
    }

    db.tasks.splice(taskIndex, 1);
    saveDatabase(db);

    res.json({ message: 'Admin deleted the task successfully' });
});

// Logout functionality
Tabnine | Edit | Test | Explain | Document
app.post('/logout', (req, res) => {
    req.session.destroy();
    res.json({ msg: 'Logged out successfully' });
});

// Private route, only accessible after login
app.use(isAuthenticated);
app.use('/app', express.static(path.join(__dirname, 'private'))); // Serve private files

// Start the server
Tabnine | Edit | Test | Explain | Document
app.listen(PORT, () => {
    console.log(`Server running on http://localhost:${PORT}`);
});
```

Practical Part

Task 2: Development of Static Files

🔹 3.2 authentification.js – Helper File for Authentication

Purpose:

- Manages user sessions, password hashing, and user verification.

```
// authentication.js
const fs = require('fs');
const bcrypt = require('bcrypt');
const { v4: uuidv4 } = require('uuid');


Tabnine | Edit | Explain
const loadUsers = () => {
    try {
        const data = fs.readFileSync('./database/user.json', 'utf8');
        return JSON.parse(data);
    } catch (error) {
        console.error('Error loading users:', error);
        return [];
    }
};

Tabnine | Edit | Explain
const saveUsers = (users) => {
    try {
        fs.writeFileSync('./database/user.json', JSON.stringify(users, null, 4));
    } catch (error) {
        console.error('Error saving users:', error);
    }
};

const registerCallback = async (req, res) => {
    try {
        let { login, password, role } = req.body;

        if (!login || !password) {
            return res.status(400).json({ error: 'Username and password are required' });
        }

        if (role !== 'user' && role !== 'admin') {
            role = 'user';
        }

        const users = loadUsers();

        // Check if user already exists
        const existingUser = users.find(user => user.login === login);
        if (existingUser) {
            return res.status(400).json({ error: 'Username already exists' });
        }

        // Hash password and create new user
        const hash = await bcrypt.hash(password, 10);
        const newUser = {
            id: uuidv4(),
            login,
            hash,
            role
        };

        users.push(newUser);
        saveUsers(users);

        res.status(201).json({ message: 'User registered successfully' });
    } catch (error) {
        console.error('Registration error:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
};

const loginCallback = async (req, res) => {
    try {
        const { login, password } = req.body;
        const users = loadUsers();
        const user = users.find(user => user.login === login);

        if (!user) {
            return res.status(401).json({ error: 'Invalid username or password' });
        }

        const isMatch = await bcrypt.compare(password, user.hash);
        if (!isMatch) {
            return res.status(401).json({ error: 'Invalid username or password' });
        }

        req.session.userID = user.id;
        req.session.role = user.role;

        res.json({
            message: 'Login successful',
            userID: user.id,
            username: user.login, // Add username to response
            role: user.role
        });
    } catch (error) {
        console.error('Login error:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
};

Tabnine | Edit | Explain
const isAuthenticated = (req, res, next) => {
    if (req.session && req.session.userID) {
        next();
    } else {
        res.status(401).json({ error: 'Authentication required' });
    }
};

module.exports = {
    registerCallback,
    loginCallback,
    isAuthenticated
};
```

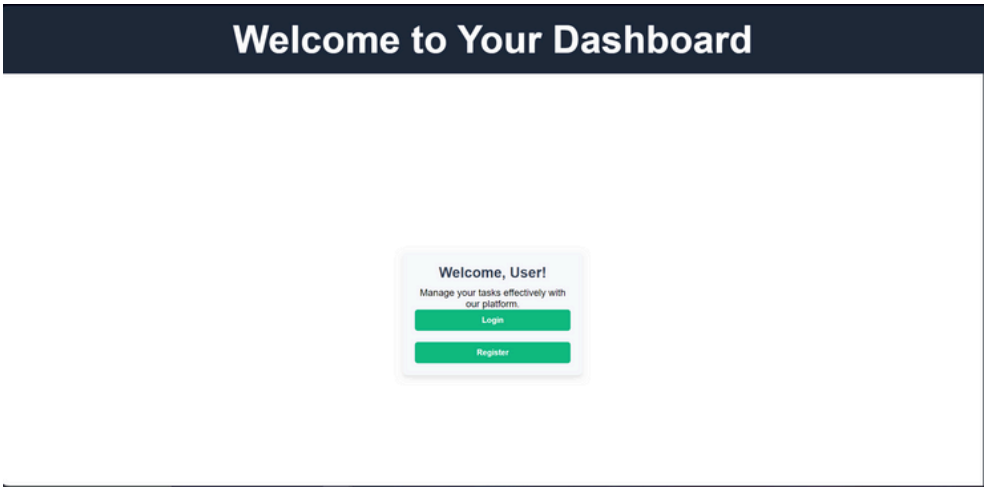
Practical Part

Task 3. Execution of the Project

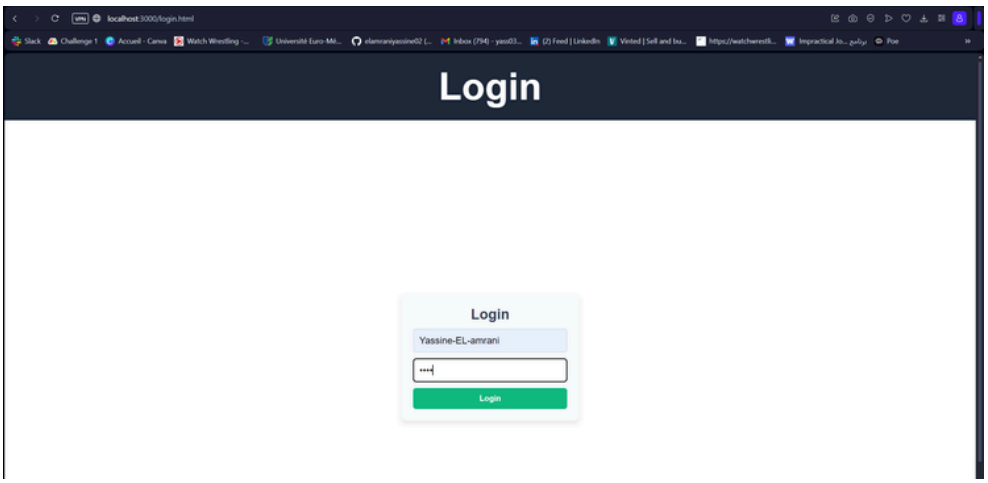
Start the Server
node app.js

```
PS C:\Users\User\Documents\uemf\S7\Plateforme NodeJs\taskApp> node app.js
express-session deprecated undefined resave option; provide resave option app.js:13
PS C:\Users\User\Documents\uemf\S7\Plateforme NodeJs\taskApp> node app.js
PS C:\Users\User\Documents\uemf\S7\Plateforme NodeJs\taskApp> node app.js
PS C:\Users\User\Documents\uemf\S7\Plateforme NodeJs\taskApp> node app.js
express-session deprecated undefined resave option; provide resave option app.js:13
express-session deprecated undefined saveUninitialized option; provide saveUninitia
express-session deprecated undefined resave option; provide resave option app.js:13
express-session deprecated undefined saveUninitialized option; provide saveUninitia
express-session deprecated undefined saveUninitialized option; provide saveUninitia
Server running on http://localhost:3000
```

idex.html page



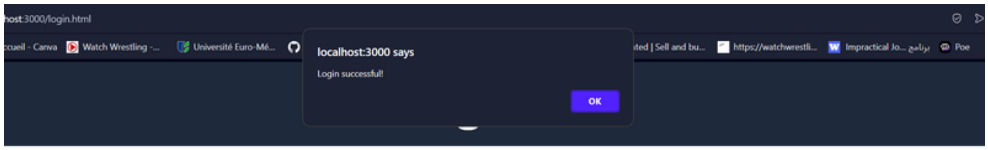
login page



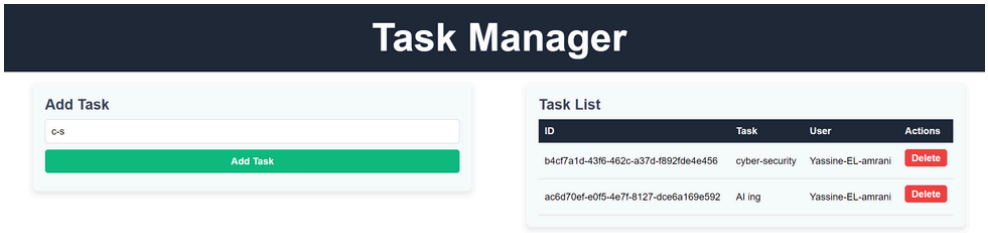
Practical Part

Task 3. Execution of the Project

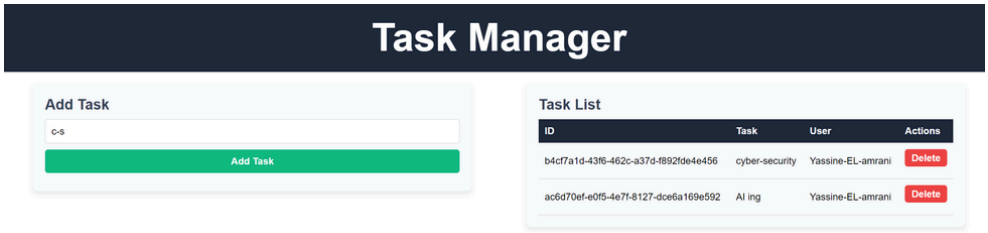
login secsefully



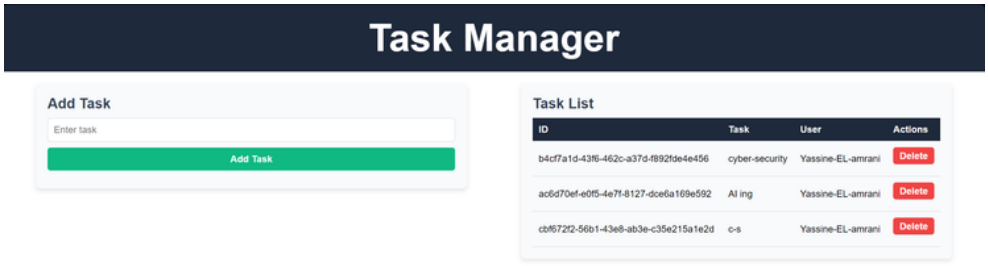
task manager page



add task



task added



Practical Part

Task 4: Documentation of Vulnerabilities and Errors

1. File Handling and Database Vulnerability (FS Operations)

Problem

- The project uses fs.readFileSync() and `fs.writeFileSync()`
  These methods are blocking and can cause performance issues.
- The database files (**user.json**, **task.json**) are stored in plain text vulnerable to unauthorized access.

Impact

- Data Corruption : If multiple requests modify the database simultaneously, it can cause data loss or corruption.
- Security Risk: Storing sensitive data (like passwords) in unencrypted JSON files can lead to data theft if an attacker gains access.

Solution

✓ Use a Database: Replace JSON files with SQLite, MongoDB, MySQL, or PostgreSQL to ensure better security and concurrency handling.
✓ File Permissions: Restrict access to database files to prevent unauthorized modifications.
✓ Async Operations: Use asynchronous file operations (fs.promises) instead of synchronous ones.
✓ Regular Backups: Implement a backup strategy to recover lost data.

2. Sensitive Data Storage (Password Hashing)

Problem

Passwords are hashed using bcrypt, but the default salt rounds (10) could be improved.
No password strength validation is enforced during registration.

Impact

Weak Passwords: Users can set weak passwords, making them vulnerable to attacks.
Fast Hashing: A low bcrypt salt rounds value makes password hashes easier to crack.

Solution

Practical Part

Task 4: Documentation of Vulnerabilities and Errors

✓ Increase Salt Rounds: Use `bcrypt.hash(password, 12)` (or 14 for higher security).
✓ Enforce Strong Passwords: Require uppercase, lowercase, numbers, and special characters.

---

3. Insecure Password Comparison (User Enumeration Risk)

Problem

- The login function leaks whether a username exists when a user enters an incorrect password.
- This helps attackers identify valid usernames.

Impact

- Attackers can enumerate usernames using brute-force attacks.

Solution

✓ Use a Generic Error Message: Always return `"Invalid username or password"` regardless of the actual issue.
✓ Implement Rate Limiting: Limit failed login attempts to prevent brute-force attacks.

4. Insecure Task Deletion

Problem

- Users can delete tasks just by modifying the URL (`/tasks/:taskID/:userID`).
- There's no ownership check, so a user could delete someone else's tasks.

Impact

- Attackers can delete all tasks by manipulating request parameters.

Solution

✓ Check Task Ownership: Verify that the task belongs to the requesting user before deleting it.
✓ Use JWT Authentication: Ensure requests include a valid JSON Web Token (JWT) for authorization.

Practical Part

Task 4: Documentation of Vulnerabilities and Errors

✓ Implement Role-based Access Control (RBAC): Allow only admins to delete other users' tasks.

5. Lack of Input Validation

Problem

- The application does not validate user inputs (e.g., empty passwords, overly long task names, SQL injection attempts).

Impact

- Attackers can submit invalid or malicious data, leading to errors, security flaws, or even database corruption.

Solution

✓ Use Input Validation: Ensure all fields follow a strict format and length restriction
✓ Sanitize Inputs: Use libraries like `express-validator` to filter and validate user input.

6. Lack of Error Handling and Logging

Problem

- Errors are logged only to the console, with no structured logging system.

Impact

- Harder to debug issues because error logs are stored persistently.

Solution

✓ Use a Logging Library : Implement winston or morgan to store detailed logs of errors.
✓ Standardize Error Responses: Always return consistent HTTP status codes and messages.

7. Missing Security Headers

Practical Part

Task 4: Documentation of Vulnerabilities and Errors

Problem

- The project does not set HTTP security headers (e.g., Content Security Policy, X-Frame-Options).

Impact

- Increases risk of Cross-Site Scripting (XSS), Clickjacking, and CSRF attacks.

Solution

✓ Use Helmet.js: Install and configure `helmet` to add security headers automatically.
✓ Enable CSP (Content Security Policy): Restrict script execution to trusted sources only

8. Cross-Site Scripting (XSS) and Content Injection

Problem

- User inputs are not escaped or sanitized before being displayed on web pages.

Impact

- Attackers can inject malicious JavaScript into forms, allowing them to steal cookies or session data.

Solution

✓ Escape User Input: Use express-validator or sanitize data before rendering.
✓ Use Secure Template Engines: Template engines like EJS automatically escape user input

9. Server-Side Authorization Checks for Task Deletion

Problem

- The server does not verify if a user owns a task before deleting it.

Impact

- Any logged-in user can delete any task by modifying the task ID.

Practical Part

Task 4: Documentation of Vulnerabilities and Errors

Solution

✓ Validate Task Ownership: Check the database to confirm the user owns the task before allowing deletion.
✓ Use JWT Authentication: Only authenticated users should be able to delete tasks.

10. Lack of Authentication Middleware

Problem

- No middleware exists to protect routes like `/tasks` or `/dashboard`.

Impact

- Any user (even unauthenticated visitors) can access protected pages.

Solution

✓ Implement JWT Authentication Middleware: Protect routes by verifying tokens.
✓ Use Role-based Access Control (RBAC): Ensure only admins can access certain routes.

Conclusion

This project highlights the critical importance of web application security. The analyzed Node.js task management system exhibited multiple vulnerabilities, including plaintext password storage, weak authentication, and insecure session management.

By implementing best security practices—such as bcrypt hashing, input validation, CSRF protection, brute-force prevention, and secure session handling—the application can be made significantly more secure.

Going forward, continuous security audits, penetration testing, and compliance with security standards (e.g., OWASP, NIST) will be crucial to maintaining a robust security posture.

By addressing these issues, we ensure a safer web environment for users and protect sensitive data from exploitation.