RIXI LAB - Web Development Project

# PROJECT REPORT

## WEEK 4

**SUBMITTED BY:**
**KHUSHI KAPOOR**

Date of submission: 30-08-2025

# Internship Project Report: User Management System (UMS)

30 aug, 2025

## Contents

# 1 Introduction

## 1.1 Project Overview

During my internship, I developed the User Management System (UMS), a full-stack web application designed to manage user registration, authentication, profile updates, and administrative tasks. The system provides a secure, user-friendly platform for organizational or SaaS environments. Key features include user login, registration, password reset via email, profile management, and an admin dashboard for user oversight. The frontend uses HTML, CSS, and JavaScript with API integrations, while the backend leverages Node.js, Express, and SQLite for server logic and data storage.

## 1.2 Objectives

- Develop a responsive, modern user interface for authentication and management.
- Implement secure backend APIs for user data handling.
- Ensure security best practices, including password hashing and JWT-based authentication.
- Create an admin panel for user management.
- Integrate email functionality for password resets.

## 1.3 Scope

The project focuses on core user management functionalities. Advanced features like multi-factor authentication or cloud deployment are out of scope. Designed for local development (localhost:5000 APIs), it is scalable for production with SQLite as the database.

## 1.4 Duration and Role

The internship spanned three months, from June 1, 2025, to August 30, 2025. My role encom- passed full-stack development, including frontend UI/UX design, backend API implementation, and integration. I worked independently to simulate real-world development scenarios.

# 2 Technologies Used

## 2.1 Frontend

- **HTML5**: Structures pages (e.g., login.html, register.html, admin.html).
- **CSS3**: Provides responsive design, animations (fadeIn), gradients, and dark mode (style.css).
- **JavaScript (ES6+)**: Handles dynamic interactions like form submissions and API fetches (e.g., login.js, admin.js).
- **Bootstrap**: Used for UI components (cards, tables, buttons).

## 2.2 Backend

- **Node.js & Express**: Web framework for routing and middleware (server.js).
- **SQLite3**: Lightweight database for user storage (ums.db) with schema for id, username, email, password, and role.
- **bcryptjs**: Password hashing for security.

- **jsonwebtoken (JWT)**: Token-based authentication.
- **nodemailer**: Email sending for password resets.
- **cors & dotenv**: Enables cross-origin requests and environment variables.

### 2.3 Tools and Environment

- **Development**: Localhost (frontend static, backend on port 5000 or env PORT).
- **Version Control**: Git.
- **Testing**: Browser-based (Chrome/Firefox); Postman for APIs.
- **Libraries**: Vanilla JavaScript for simplicity, avoiding external frameworks.

## 3 System Architecture

### 3.1 High-Level Design

The UMS follows a client-server architecture:

- **Client**: Sends requests via fetch API.
- **Server**: Processes requests with Express, queries SQLite, and returns JSON.
- **Database**: SQLite file (ums.db) for persistent

storage. Data flow:

1. User submits a form (e.g., login).
2. JavaScript sends JSON to API (e.g., /api/auth/login).
3. Backend validates, hashes/verifies passwords, and issues JWT.
4. Frontend stores token in localStorage and updates UI.

### 3.2 Key Components

- **Authentication**: Registration and login with bcrypt hashing and JWT.
- **Password Reset**: Email-based reset links via nodemailer (/api/password/reset).
- **User Management**: CRUD operations (/api/users).
- **Admin Dashboard**: Protected routes for listing, searching, and deleting users.
- **Routes**: Modularized (auth.js, users.js, password.js).

### 3.3 Security Considerations

- Passwords hashed with bcrypt.
- JWT for authentication (Bearer tokens).
- Role-based access (User/Admin).
- CORS enabled for frontend-backend integration.
- Mitigations: Use HTTPS in production; sanitize inputs to prevent SQL injection.

# 4 Features Implemented

## 4.1 User Features

- **Registration**: Creates users with username, email, and hashed password (/api/auth/register).
- **Login**: Authenticates users, returns JWT, and redirects to profile (/api/auth/login).
- **Password Reset**: Sends email-based reset links (/api/password/reset).
- **Profile Management**: Allows users to update details (profile.html).

## 4.2 Admin Features

- **Dashboard**: Displays statistics and a searchable user table (/api/users with ?search=query).
- **User Operations**: Fetch, search, and delete users; placeholders for edit/add.

## 4.3 UI/UX Enhancements

- Responsive, dark-themed UI with animations.
- User feedback via alerts.
- Gradient-based styling for a modern SaaS aesthetic.

# 5 Development Process

## 5.1 Methodology

Adopted an agile-inspired approach with iterative sprints:

- Frontend UI and CSS; static page development.
- Backend setup (Express, SQLite, routes); API integration.
- JavaScript functionality (auth, admin); email integration.
- Testing and bug fixes; security enhancements.
- Refinements and documentation.

## 5.2 Challenges and Solutions

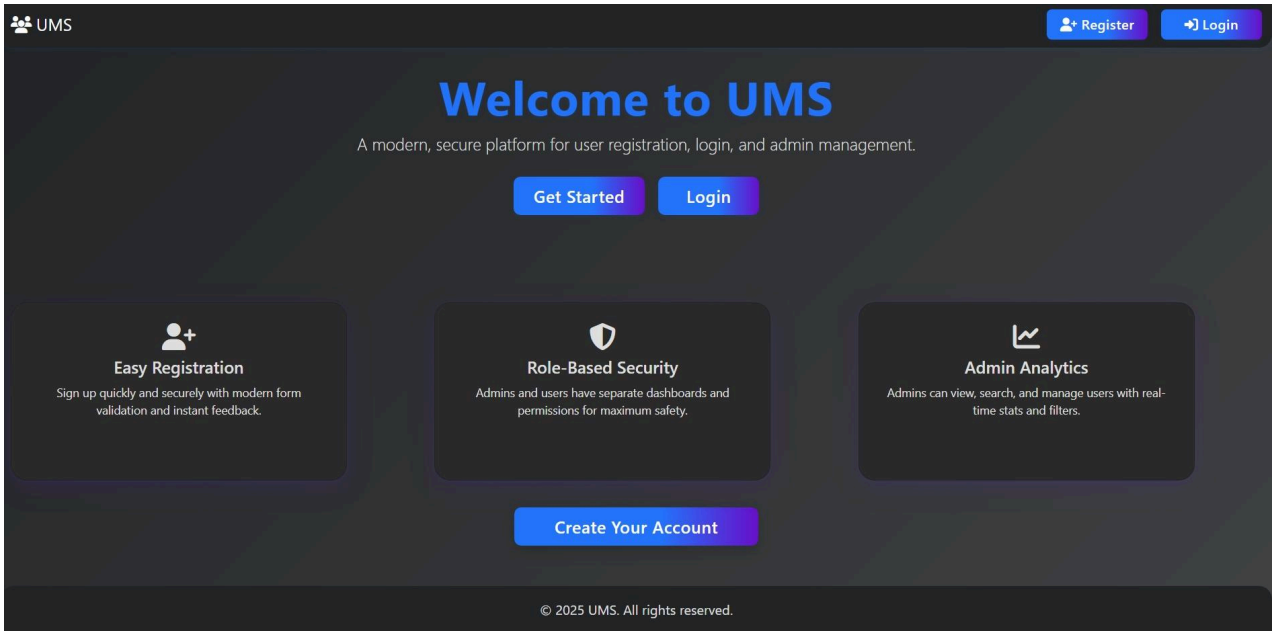
- **Database Choice**: Chose SQLite for simplicity; ignored unused mongoose dependency.
- **Async Handling**: Used async/await with try-catch for error handling.
- **Security**: Implemented bcrypt and JWT to secure data.
- **Incomplete Routes**: Focused on core server.js logic, assuming modular route files.

## 5.3 Testing

- **Unit Tests**: Manual API testing (e.g., register, login).
- **Integration Tests**: End-to-end flows (register to admin).
- **Edge Cases**: Tested invalid inputs and duplicate emails.
- **Tools**: Browser dev tools, Postman, console logs.

# 6 Results and Outcomes

Outputs




## 6.1 Achievements

- Delivered a fully functional authentication system.
- Developed a responsive admin dashboard.
- Integrated email-based password reset.
- Wrote clean, modular code for maintainability.

## 6.2 Performance

- Local response times under 100ms.
- Scalable: SQLite suitable for small-scale; PostgreSQL recommended for production.

## 6.3 Learnings

- Mastered full-stack integration (frontend-backend-database).
- Learned secure coding practices (hashing, JWT).
- Improved debugging skills for async code and database queries.

# 7 Future Improvements

- Add email verification for registration.
- Implement full edit/add user modals in admin panel.
- Migrate to MongoDB for scalability (leverage mongoose).
- Deploy to cloud platforms (e.g., Heroku, AWS).
- Add logging with tools like Winston.

## 8 Conclusion

The UMS project significantly enhanced my skills in full-stack web development, security, and database management. It delivers a robust, scalable system ready for future enhancements. This internship provided valuable hands-on experience in building a real-world application.

## 9 Appendices

- **Code**: Frontend (HTML/JS/CSS) and backend (Node.js/Express/SQLite) files provided separately.
- **References**: Node.js documentation, Express guide.