MINI PROJECT

COURSECODE: ENSI152

GROUP CODE: Y1-2024-25-G288

PROJECT REPORT

MASTERS OF COMPUTER APPLICATION

SUBMITTED BY :

AJAY KUMAR(2401560073)

BRAJESH KUMAR(2401560021)

UNDER THE SUPERVISION OF

DR. MOHAMMAD AIJAZ

School of Engineering & Technologies

K.R. MANGALAM UNIVERSITY

Sohna, Gurgaon(Haryana) 122103, India

Abstract

This project report describes the design and implementation of a full-stack task management web application built using modern web technologies. The application allows users to manage their daily tasks efficiently through a simple and intuitive interface. The frontend was built using **React.js** for a responsive and component-driven user experience, styled with **Tailwind CSS** for rapid and elegant UI design. The backend server was developed using **Node.js** and **Express.js**, providing a robust and scalable API, and the data is stored in a **MongoDB** database. This project demonstrates the seamless integration of frontend and backend systems, RESTful API development, and secure data handling in a real-world web application.

Introduction

In today's fast-paced world, managing tasks and responsibilities efficiently is essential. With the proliferation of digital platforms, individuals increasingly seek online tools to track and manage their to-do lists. However, many available solutions are either overly complex or lack flexibility for customization. This project aims to solve that problem by developing a lightweight, user-friendly web application that enables users to add, edit, delete, and manage tasks effectively.

Problem Statement

Users often struggle with managing daily tasks using traditional methods like paper planners or scattered digital notes. This results in inefficiencies and missed deadlines.

- Access of varied workshop in Sohna is limited and time consuming .
- Physical shops lack variety or change higher prices.
- Limited Customer Engagement

Objective

The objective is to create a responsive web application that offers:

- User registration and login
- CRUD operations on tasks
- A clean, minimal UI
- A RESTful backend with secure data storage

Target Audience

Students, working professionals, and teams who require a simple yet effective task management system accessible from any device.

Technology Stack Overview

Developing a modern, responsive, and dynamic web application requires the integration of various technologies across the frontend, backend, and database layers. The task management web application is built using a robust and scalable full-stack JavaScript ecosystem. Each technology in the stack was selected based on its performance, community support, ease of use, and compatibility with other tools. This section outlines the technologies used and their specific roles in the development of the application.

◇ Frontend Technologies

The frontend, or client-side, of the application is responsible for what users see and interact with. It involves building user interfaces, managing the application state, and communicating with the backend APIs.

1. React.js

React is a JavaScript library developed by Facebook for building interactive user interfaces. It uses a component-based architecture, where the UI is broken down into reusable pieces called components. This approach promotes maintainability, reusability, and better organization of code.

Why React?

- Virtual DOM for fast rendering
- One-way data binding for controlled data flow
- Rich ecosystem (React Router, Redux, etc.)
- React Hooks (useState, useEffect) for functional programming

Usage in the Project:

- Components such as TaskList, AddTaskForm, EditTaskForm, and Navbar
- State management for form inputs and API data
- Conditional rendering for authenticated vs unauthenticated views

2. Tailwind CSS

Tailwind CSS is a utility-first CSS framework used for styling the user interface. Unlike traditional CSS frameworks like Bootstrap, Tailwind does not come with pre-designed components. Instead, it provides utility classes that allow developers to create custom designs directly in HTML/JSX.

Why Tailwind CSS?

- Rapid UI development with utility classes
- Mobile-first and responsive design by default
- Customization via tailwind.config.js
- Cleaner code with less custom CSS

Usage in the Project:

- Layouts (flex, grid, space-x, gap-y, etc.)
- Buttons, modals, input fields, and cards
- Responsive behavior for mobile and desktop views

◇ Backend Technologies

The backend, or server-side, is responsible for handling business logic, processing client requests, interacting with the database, and ensuring application security.

3. Node.js

Node.js is a server-side runtime environment that allows JavaScript to run outside the browser. It is built on Google Chrome’s V8 engine and is known for its non-blocking, event-driven architecture.

Why Node.js?

- JavaScript on both frontend and backend (full-stack JS)
- High performance for I/O-heavy operations
- Large NPM ecosystem for ready-to-use packages
- Asynchronous programming using Promises and async/await

Usage in the Project:

- Running the server (e.g., server.js)
- Handling environment variables using dotenv
- Middleware integration and error handling

4. Express.js

Express.js is a minimal and flexible Node.js web framework that provides a set of features for building web and mobile applications. It simplifies routing, middleware configuration, and server logic.

Why Express.js?

- Simple and lightweight syntax
- Support for RESTful API development
- Middleware support (authentication, logging, etc.)
- Easy integration with MongoDB and other services

Usage in the Project:

- Setting up routes like /api/tasks, /api/users
- Middleware for JSON parsing, error handling, and token validation
- Building and serving a REST API

◇ Database Technology

Data persistence is a critical part of any application. In this project, a NoSQL database was chosen due to its flexibility and scalability for handling structured and unstructured data.

5. MongoDB

MongoDB is a document-oriented NoSQL database that stores data in JSON-like documents with flexible schemas. It is highly scalable and suitable for modern applications where schema design may evolve over time.

Why MongoDB?

- Schema-less flexibility for rapid development
- JSON-style document storage fits naturally with JavaScript
- Scalable and cloud-friendly (via MongoDB Atlas)
- Good integration with Node.js using Mongoose

Usage in the Project:

- Collections: users, tasks
- Each user has a list of tasks stored as separate documents
- Used Mongoose for defining schemas and handling database queries

Features Implemented

The Task Management Web Application is designed to offer a seamless and productive experience for users to manage their tasks. Below is a comprehensive breakdown of all the major features implemented in the system. These features were developed keeping usability, scalability, and performance in mind.

---

1. User Authentication (Register & Login)

User authentication is a core feature of the application that allows individual users to have their own workspace and secure their task data.

Registration

- New users can sign up using a form that captures their **username**, **email**, and **password**.
- Passwords are **hashed** using bcrypt before storing them in the MongoDB database to ensure security.
- Duplicate user registrations are prevented by checking existing emails.

Login

- Users can log in using their registered email and password.
- On successful login, a **JWT (JSON Web Token)** is generated and returned to the client.
- The token is stored on the client-side (in local storage or cookies) and sent with each request to authorize access to protected routes.

Token Verification

- Middleware on the backend verifies the JWT on every protected route, ensuring that only logged-in users can perform certain operations.

---

2. Task Management (CRUD Operations)

The main functionality of the application revolves around **CRUD** (Create, Read, Update, Delete) operations on tasks.

Create Task

- Users can create a new task using a simple form that includes fields such as **title** and **description**.
- The created task is immediately saved to the database and displayed in the UI.

Read / View Tasks

- Users can view a **list of their tasks** after logging in.
- Tasks are fetched from the backend and displayed dynamically using React.
- Each task shows essential information: title, description, status, and creation date.

Update Task

- Users can edit a task using a pre-filled form.
- This includes updating the task's **title**, **description**, and **status** (e.g., complete/incomplete).
- The changes are saved to the database, and the UI reflects updates in real-time.

Delete Task

- Users can delete tasks they no longer need.
- Upon confirmation, the task is removed from the database and instantly from the UI.

---

3. Responsive Design

The application is built with **mobile-first responsiveness** using **Tailwind CSS**.

- Layouts adjust dynamically for mobile, tablet, and desktop views.
- Tailwind’s utility classes (flex, grid, gap, breakpoints) are used for adaptive design.
- Buttons, forms, and cards scale appropriately with screen size to provide an optimal user experience on any device.

---

4. User-Friendly Interface

A clean, minimalistic interface helps users navigate and manage tasks without confusion.

- Task lists are displayed with clear labels and color-coded status indicators (e.g., green for completed, gray for pending).
- Input validation ensures users receive feedback when fields are missing or incorrect.
- Empty states, loading indicators, and success/error toasts enhance the interactivity of the application.

---

5. State Management with React Hooks

React’s useState and useEffect hooks are used to manage:

- Form data
- Authentication state (logged in/out)
- Dynamic updates when tasks are added, edited, or deleted

This allows for real-time UI updates and smooth user experiences without unnecessary page reloads.

---

6. Filtering and Sorting Tasks

Basic filtering functionality is implemented to improve usability:

- **Filter by Completion Status**: Show only completed or pending tasks.
- **Sort by Creation Date**: Tasks are displayed with the most recent ones at the top.

This feature makes it easier for users to focus on high-priority tasks and quickly find what they need.

---

7. RESTful API Integration

All task-related operations are handled via a **REST API** built with **Express.js**.

API Endpoints:

- GET /api/tasks – Get all user tasks
- POST /api/tasks – Create a new task
- PUT /api/tasks/:id – Update a specific task
- DELETE /api/tasks/:id – Delete a specific task

These endpoints follow REST principles and allow for easy testing, scalability, and integration with other frontends or mobile apps in the future.

---

8. Form Validation and Error Handling

Form validation ensures that users input valid data before submission:

- Required fields: Title and Description
- Minimum character length enforced
- Clear error messages displayed near inputs

On the backend, error handling middleware is used to return appropriate HTTP status codes and messages:

- 400 for bad requests
- 401 for unauthorized access
- 500 for server errors

---

9. Secure API Access (JWT Protected Routes)

Security is enforced using **JWT (JSON Web Tokens)**:

- Tokens are stored on the client side and sent with every request using the Authorization header.
- Middleware on the server verifies the token and decodes the user information before granting access to the database.

This ensures only authenticated users can perform operations on their own data.

---

10. Developer Tools Integration

Several tools were integrated to assist in development and testing:

- **Postman**: For manual testing of API endpoints
- **React Developer Tools**: For inspecting component state
- **Console Logging**: For debugging backend logic and tracing requests

This helped identify and fix bugs early in the development cycle.

Testing and Debugging

Manual Testing

All components and API routes were manually tested to ensure correctness.

Postman Testing

- Tested all backend routes using Postman.
- Verified user auth and protected routes.

Browser Debugging

Used Chrome DevTools for inspecting elements and React Developer Tools to track component state.

Console Debugging

Extensive use of console.log and try-catch blocks in backend routes for tracking issues.

Challenges Faced

Building a full-stack web application comes with a range of technical and non-technical challenges, especially when integrating multiple technologies across the frontend, backend, and database layers. This section highlights the most significant obstacles encountered during the development of the task management system, along with how they were identified and resolved.

---

1. Backend and Frontend Integration

One of the biggest challenges was integrating the **frontend (React)** with the **backend (Express.js API)**. While both were running on different ports during development (localhost:3000 for React and localhost:5000 for Express), making cross-origin requests triggered **CORS (Cross-Origin Resource Sharing)** issues.

Problem:

- Frontend API requests were being blocked due to different origins.
- Errors like Access-Control-Allow-Origin not allowed.

Solution:

- Installed and configured the **CORS middleware** in the backend:

javascript

CopyEdit

```
const cors = require("cors");

app.use(cors({ origin: "http://localhost:3000", credentials: true }));
```

- Ensured API endpoints returned proper HTTP status codes and JSON responses for seamless frontend consumption.

---

2. Implementing JWT Authentication

Using **JSON Web Tokens (JWT)** for authentication was initially difficult, especially in managing token generation, storage, and validation across sessions.

Problem:

- Users would lose their login state on page refresh.
- Tokens stored in local storage needed to be securely handled to avoid XSS vulnerabilities.
- Backend needed token verification for every protected route.

Solution:

- Used jsonwebtoken to sign and verify tokens.
- Created a middleware in Express to decode and verify the token on each request.
- Stored tokens securely in local storage (or cookies with proper security settings for deployment).
- Implemented a token check in React (useEffect) on app load to persist login state.
-

---

3. Managing State in React

As the application grew, managing **component state** and prop drilling became challenging, especially for the task list and user session.

Problem:

- State updates were not reflected properly in nested components.
- Re-rendering was inconsistent when updating tasks or user info.

Solution:

- Used **React Hooks** (useState, useEffect) effectively to manage state.
- Lifted state up to common parent components when needed.
- Considered using context or a global state manager (like Redux) for scalability.

---

4. Tailwind CSS Utility Overload

While **Tailwind CSS** offers great flexibility, overuse of utility classes led to cluttered JSX code and made components hard to read.

Problem:

- Long className attributes made components messy and difficult to maintain.
- Repetition in layout and color utility classes.

Solution:

- Refactored repeated styles into reusable **custom components**.
- Used Tailwind's @apply directive for creating custom CSS classes in index.css or tailwind.config.js.

---

5. MongoDBchema Design

MongoDB's flexibility was a double-edged sword. Designing the **right schema structure** was tricky when deciding between embedding and referencing documents.

Problem:

- Whether to embed tasks inside user documents or store them in a separate collection.
- Handling ObjectId references and population of related data.

Solution:

- Used **separate collections** for users and tasks.
- Linked tasks to users using userId reference.
- Used **Mongoose** to define strict schemas and handle validations.

---

6. Testing API with Postman and Debugging

Testing REST APIs with **Postman** was essential but at times confusing due to improper request payloads or header configurations.

Problem:

- Incorrect header setup (missing Content-Type or Authorization) resulted in 400/401 errors.
- Backend logs were unclear or missing context.

Solution:

- Added proper request logging in Express using morgan.
- Used clear error messages and consistent response formats.
- Documented API routes and tested each endpoint with different scenarios (valid/invalid data, unauthorized access).
-

---

7. Time Management & Scope Creep

During development, trying to implement too many features led to **scope creep** and unfinished features.

Problem:

- Wanting to add advanced features (task priority, calendar integration) without finalizing core functionality.
- Limited time to polish and test everything thoroughly.

Solution:

- Prioritized **MVP (Minimum Viable Product)** features: authentication, task CRUD, and UI.
- Used an agile mindset: build core first, refine later.

Conclusion

Project Summary

This project aimed to design and develop a full-stack web-based **Task Management Application** that enables users to efficiently create, track, update, and manage daily tasks. The system was built using a modern JavaScript technology stack comprising:

- **Frontend:** React.js with Tailwind CSS for a responsive and dynamic user interface.
- **Backend:** Node.js with Express.js for building a RESTful API and server-side logic.
- **Database:** MongoDB for storing user and task data securely and efficiently.

The application allows users to register and log in securely, manage personal tasks, and interact with a clean and intuitive UI. All interactions between the user and the system are handled asynchronously via API calls, ensuring a smooth and real-time experience.

Key Takeaways

The development of this project was not only a technical exercise but also an opportunity to apply theoretical knowledge in a practical, problem-solving environment. Several critical lessons and insights emerged from building this application:

1. Understanding Full-Stack Development Workflow

Building this application from the ground up provided practical experience in managing the complete web development workflow—starting from front-end UI creation to back-end logic, and finally to database integration. Each layer presented its own set of challenges, and resolving them enhanced my ability to think through the stack holistically.

2 Frontend and Backend Coordination

This project highlighted the importance of **synchronization between client-side and server-side operations**. Ensuring that React properly handles responses from Express, and that the backend is structured to provide meaningful responses, is crucial to seamless user interaction.

3. Security and Data Privacy

Integrating **JWT-based authentication** and secure password storage with hashing emphasized the importance of user data protection in modern web applications. The project enforced best practices around data validation, token management, and secure route access.

4. REST API Design

Designing RESTful routes and managing state transitions taught valuable lessons about modular, scalable, and testable backend architecture. Using Postman to test different API endpoints helped verify input handling and server responses before integrating with the frontend.

5. Clean UI with Tailwind CSS

Using Tailwind CSS was a highly efficient way to develop clean, responsive, and mobile-friendly interfaces. It made styling easier and more scalable than traditional CSS, particularly for rapid development.

Project Impact and Benefits

This task management system has practical utility in everyday life for individuals who want a personal productivity tool. It is scalable enough to be adapted for teams, and its modular architecture allows additional features like:

- Task deadlines and notifications
- User roles and shared workspaces
- Calendar and third-party integrations (e.g., Google Calendar)
- Real-time collaboration via sockets or websockets

The ability to manage and prioritize tasks in an organized way can improve productivity, reduce mental clutter, and promote accountability.

Future Scope

Although the current application fulfills its basic objectives, there are numerous areas for future development and enhancement:

1. Enhanced UI/UX

- Add animations and transitions using libraries like Framer Motion.
- Implement drag-and-drop task ordering.

2. Advanced Features

- Add categories or tags for task grouping.
- Integrate a calendar view for time-based task tracking.
- Enable notifications and reminders (via email or in-browser).

3. Role-Based Access Control

- Add user roles (admin, regular user).
- Implement group tasks and shared task lists.

4. Mobile App Integration

- Convert the web app into a Progressive Web App (PWA).
- Use React Native for cross-platform mobile versions.

5. Deployment and CI/CD

- Deploy frontend on platforms like Vercel and backend on Render or Railway.
- Integrate CI/CD pipelines using GitHub Actions for automated testing and deployment.

Final Reflection

The journey of creating this application was filled with both technical challenges and rewarding learning experiences. It required a solid understanding of the full software development lifecycle—from designing APIs and managing database schemas to creating interactive UI components and ensuring data consistency.

Completing this project successfully not only deepened my knowledge in web technologies but also enhanced soft skills like time management, debugging patience, and adaptability to new tools and frameworks.

In conclusion, this task management application serves as a practical, well-structured foundation for future full-stack applications. It validates the power and flexibility of the JavaScript ecosystem and opens the door for deeper exploration into cloud services, performance optimization, and enterprise-level architecture.