Mini Project Report

Project Title

Collaborative Project Management Web Application

Introduction

In modern workplaces, collaborative tools have become essential for enhancing team productivity and streamlined workflows. The colab_hub project was conceived as a comprehensive web application aimed at facilitating seamless communication and task management among project teams. Its design focuses on balancing user-friendliness with robust backend architecture to support real-time collaboration. This mini project lays the foundation for understanding full-stack development principles through practical implementation.

The rise of remote work and distributed teams has increased the need for effective collaboration platforms. colab_hub's feature set reflects this trend, targeting use cases such as project task allocation, status tracking, and direct messaging within teams.

Problem Statement

Managing projects with multiple collaborators often leads to communication gaps, delayed task completions, and scattered information across emails or spreadsheets. Existing solutions could be complex or costly for smaller teams or academic groups. This project aims to address these challenges by building a free, lightweight, and customizable collaboration platform tailored for academic purposes and small professional teams.

Objectives

The primary objectives of the colab_hub project are outlined below:

- Build a secure user authentication system for multiple users.

- Enable role-based access to differentiate between project owners, collaborators, and viewers.

- Develop efficient project and task management features to allow creation, assignment, modification, and deletion of tasks.

- Implement a communication system to foster collaboration within each project team.

- Design a responsive interface accessible via desktop and mobile devices.

- Ensure a scalable backend using APIs to prepare for future expansion.

Scope of the Project

This mini project covers the development of core collaboration functionalities. Advanced real-time updates, file sharing capabilities, and third-party integrations are excluded but considered for potential future work. The emphasis is on understanding full-stack concepts, technical implementation, and secure data management.

Tools and Technologies

Frontend

- React.js: Selected for its component-based structure, reusability, and large community support. Enables dynamic UI with state management.

- Vite: A build tool for faster frontend development and hot module replacement.

- Bootstrap: Used to ensure responsive and attractive UI design without extensive custom CSS.

Backend

- Node.js: Provides server-side JavaScript execution for scalable network applications.

- Express.js: A minimal and flexible web application framework used for REST API implementation.

-JSON Web Tokens (JWT): Ensures secure authentication and session management.

Database

- MongoDB Atlas: A cloud-hosted NoSQL database offering flexibility in storing project and user data.

Development Environment and Tools

- Visual Studio Code: The primary IDE used for both frontend and backend coding.

- Git and GitHub: For version control, branch management, and collaboration.

- Postman: API testing and debugging tool.

Literature Review

Collaborative platforms like Trello, Asana, and Microsoft Teams highlight the importance of easy task management and communication. Academic studies suggest that integration of chat, task visibility, and real-time updates improves group project outcomes. However, these platforms can be overwhelming or expensive for smaller teams.

This project draws inspiration from such giants but focuses on simplicity and educational value. Use of MERN-like stacks is common for rapid app development, and the choice of MongoDB Atlas allows easy database scaling without heavy local setup.

System Analysis and Design

Functional Requirements

- Users must be able to register and log in securely.

- Users can create projects and add team members.

- Tasks can be created with attributes like title, description, priority, status, and deadlines.

- Users assigned to a project see only tasks and messages relevant to that project.

- Users can update task states and add comments/messages.

functional Requirements

- The system should be secure from unauthorized access and attacks.

- APIs must respond quickly to ensure good user experience.

- The app should be responsive across devices.

- Codebase must be modular and maintainable.

Entity-Relationship Diagram

The data model includes entities such as User, Project, Task, and Message. Users can have multiple projects, and projects contain multiple tasks. Messages are linked to projects and users.

(The ER diagram can be visualized as one-to-many relationships between Users and Projects, and Projects and Tasks, with each Message linked to a Project and User.)

Architecture Diagram

The application architecture follows the client-server model with REST API communication between frontend and backend. Authentication middleware ensures token validation before accessing secure endpoints.

Implementation Details

Frontend Components

- Login/Register: Controlled forms for authentication inputs with validation.

- Dashboard: Overview of all user projects.

- Project Page: Displays project tasks and conversation.

- Task Manager: Create, update, and view tasks with status indicators and deadlines.

- Chat Interface: Basic message board for project communication (if implemented).

Backend API Endpoints

- /api/auth/register: User registration.

- /api/auth/login: Returns JWT on successful login.

- /api/projects: CRUD for projects.

- /api/tasks: Task management routes linked to projects.

- /api/messages: Handles user messages within projects.

Security Measures

- Passwords hashed via bcrypt before storage.

- JWT stored client-side with expiration.

- Secure route middleware to prevent unauthorized calls.

- Input sanitization to prevent injection attacks.

Database Schema

- User Schema: username, email, hashedPassword, createdAt.

- Project Schema: title, description, createdBy (User ref), members (array of User refs).

- Task Schema: title, description, status, priority, dueDate, project ref.

- Message Schema: content, sender ref, project ref, timestamp.

Testing and Validation

- Unit testing of backend APIs using Jest and Supertest.

- Manual functional testing of frontend components.

- Cross-browser and device responsiveness checks.

- API response time monitoring with Postman.

Challenges Faced

- Implementing JWT authentication and maintaining user session integrity.

- Handling asynchronous calls and UI state synchronization in React.

- Debugging CORS issues during frontend-backend interaction.

- Structuring data schema for flexible task and message management.

- Ensuring responsive design across multiple screen sizes.

Project Management

The project followed Agile-inspired sprints with incremental feature additions and testing. Version control via GitHub allowed tracking changes and collaboration feedback.

Timeline (Example)

- Week 1: Requirement gathering and tool setup.

- Week 2-3: Backend API development and database schema design.

- Week 4: Frontend scaffolding and authentication implementation.

- Week 5: Project and task management features.

- Week 6: UI polishing, testing, and debugging.

Future Scope & Improvements

- Integrate real-time updates via WebSockets or Firebase.

- Add file upload and document sharing capabilities.

- Enhance RBAC with more granular permissions for users.

- Add analytics dashboards to track project progress.

- Deploy as Docker containers for easier scalability.

Conclusion

The colab_hub mini project effectively served as a practical learning experience in full-stack development. By integrating modern web technologies in a single application, it demonstrated key concepts like RESTful API design, JWT-based security, and responsive frontend development. The project lays a solid foundation for extending features into a mature collaboration tool.

Detailed Technology Selection Justification

Frontend: React.js and Vite

React.js was chosen for its declarative, component-based design which promotes reusability and maintainability of the user interface. Its large ecosystem and community support ease development and troubleshooting. Vite was preferred over traditional build tools (like Webpack or Create React App) for its lightning-fast hot module replacement and minimal configuration, enabling rapid UI iterations essential in the development cycle.

Backend: Node.js with Express.js

Node.js offers an event-driven, non-blocking I/O model, making it well-suited for handling multiple simultaneous requests efficiently. Express.js, built on top of Node.js, simplifies server-side coding with a minimalistic yet powerful routing framework. These technologies together support rapid API development with clear modularization, key for a growing collaborative application like colab_hub.

Database: MongoDB Atlas

MongoDB’s document-oriented schema fits the flexible and evolving data structure needs of the project, such as dynamic task attributes and nested comments. The cloud-hosted Atlas service offloads database maintenance, eases scalability concerns, and provides a globally accessible infrastructure to the team during development and potential deployment phases.

Authentication: JWT

JWT tokens allow stateless session management, eliminating the complexity of server-side session storage. Tokens securely encapsulate user identity and permissions, simplifying API security implementation and enhancing overall application scalability.

UI Framework: Bootstrap

Bootstrap enables rapid design of responsive interfaces with its prebuilt grid system and styled components, ensuring colab_hub's accessibility across diverse devices without heavy custom CSS work.

Sample Workflows

User Registration and Authentication Workflow

1. User accesses the registration page and enters details (username, email, password).

2. Frontend performs form validation and submits data to backend `/api/auth/register`.

3. Backend hashes the password with bcrypt and stores user in MongoDB.

4. Upon successful registration, user is redirected to login page.

5. For login, user submits credentials to `/api/auth/login`.

6. Backend verifies credentials, generates JWT token, and sends it back.

7. Frontend stores the token securely (e.g., localStorage) and attaches it to subsequent API calls.

Project Creation and Collaboration Workflow

1. Authenticated user navigates to Dashboard and creates a new project by providing title and description.

2. Backend creates project record linked to user's ID and returns success response.

3. User invites collaborators by adding their email or username; backend updates the project members list.

4. Collaborators receive notification (future implementation) and can access project tasks and chat.

Task Management Workflow

1. Users within a project can create tasks specifying title, description, priority, status, and deadlines.

2. Creation triggers an API call to add the task linked to the project.

3. Team members can update task status (e.g., Pending, In Progress, Completed) to track progress.

4. Changes are reflected immediately upon refresh or real-time update (future scope with WebSocket).

***

Sample Code Snippets

Example: User Schema (MongoDB Model)

```javascript
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  passwordHash: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
});
```

```
userSchema.methods.setPassword = async function(password) {
  this.passwordHash = await bcrypt.hash(password, 10);
};

userSchema.methods.validatePassword = async function(password) {
  return await bcrypt.compare(password, this.passwordHash);
};

module.exports = mongoose.model('User', userSchema);
```

Example: Authentication Middleware in Express

```javascript
const jwt = require('jsonwebtoken');

function authenticateToken(req, res, next) {
  const token = req.headers['authorization']?.split(' ')[1];
  if (!token) return res.sendStatus(401);

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
}
```

```
module.exports = authenticateToken;
```

Example: React API Call for Login

```javascript
import axios from 'axios';

async function loginUser(credentials) {
  try {
    const response = await axios.post('/api/auth/login', credentials);
    if (response.status === 200) {
      localStorage.setItem('token', response.data.token);
      return true;
    }
  } catch (error) {
    console.error('Login failed:', error);
    return false;
  }
}
```

***

Testing Methodology

Unit Testing

Backend endpoints were tested for expected behavior using Jest along with Supertest to simulate HTTP requests. Major aspects under test included:

- Successful and failed user registration/login cases.

- Authorization checks blocking unauthorized API access.

- Project/task CRUD operations to ensure correct data manipulation.

Manual Functional Testing

Frontend components were subjected to usability testing manually to verify:

- Form validations prevent incorrect data submission.

- API integration reflected correctly in UI states.

- Navigation flows worked without errors.

Responsiveness Testing

The app interface was tested on different device simulators within browsers and on physical devices to ensure layouts adapt effectively.

API Testing

Postman was used for extensive API testing, including boundary condition tests and error response validations.

***

References

- React.js Documentation: https://reactjs.org/docs/getting-started.html

- Express.js Guide: https://expressjs.com/en/starter/installing.html

- MongoDB Atlas Docs: https://docs.atlas.mongodb.com/

- JWT Introduction: https://jwt.io/introduction/