Development of a Microservices-Based Project Management System in a Docker Environment

Stamatios Mavitzis
ID NUM: 2018030040
Department: Technical University of Crete - ECE
Professor: Evripidis Petrakis

Cloud and Fog Services (PLH513)
Winter semester 2025-2026

Contents

1. Introduction

1.1 Purpose and Overview

This is a project for the course **Services in the Cloud and Fog**. The purpose of this project is to design and implement a web-based **Project Management System (PMS)** that enables efficient team collaboration, task organization, and progress tracking through a modular microservices architecture. The system aims to simulate the core functionality of popular platforms such as Jira or Trello while remaining lightweight, extensible, and easy to deploy.

The project introduces a role-based management model, allowing users to perform actions according to their assigned permissions. It is built around a microservices design where the backend is separated into distinct services for handling users, teams, and tasks. These services communicate through RESTful APIs and store data in a relational database.

The system is fully containerized using **Docker**, ensuring portability and consistency across environments. It adopts a modular monolithic deployment structure that supports scalability and can be easily migrated to cloud platforms such as the **Google Cloud Platform (GCP)**.

Overall, this report presents the system's architecture, database schema, user roles, technologies used, implementation details, deployment steps, and evaluation of the final system.

1.2 Purpose and Objectives

This is a project for the course **Services in the Cloud and Fog**. This chapter presents the overall purpose of the project and its main objectives. The system is designed as a simplified project management platform that enables team organization, task tracking, and role-based collaboration.

The core objectives of the project are as follows:

- **Develop a Web-Based Management Platform:** Build a browser-accessible interface for managing users, teams, and tasks using a Flask backend and dynamic templates.
- **Implement Role-Based Access Control (RBAC):** Differentiate between `Admin`, `Team Leader`, and `Member` roles, each with tailored privileges and workflows.
- **Ensure Scalability and Portability:** Use Docker containerization to guarantee consistent behavior across environments (development, testing, production).
- **Design a Relational Database Schema:** Create a well-structured SQL database with referential integrity, normalized tables, and optimized indexing for queries and reporting.
- **Facilitate Efficient Task Tracking:** Allow users to create, assign, and monitor project tasks with deadlines, priorities, and progress updates.
- **Provide an Intuitive and Responsive User Interface:** Build a clean, minimal frontend using HTML, CSS, JavaScript, and Bootstrap for consistent styling and accessibility.
- **Promote Maintainability and Modularity:** Structure the project into reusable components (routes, templates, static assets, and utilities) following the MVC pattern.

By combining these elements, the system aims to deliver a lightweight yet powerful collaboration platform for small to medium teams, bridging the gap between manual task tracking and complex enterprise software.

1.3 Overview

The project is implemented as a full-stack web application composed of three core layers — the frontend, backend, and database — integrated through RESTful communication and managed in isolated Docker containers.

- **Frontend Layer:** Provides the graphical interface accessible via any modern web browser. It includes user-facing pages for login, dashboards, and task management, located under the `templates/` and `static/` directories.

- **Backend Layer:** Built with the Flask framework in Python, it manages user sessions, authentication, business logic, and database interactions. The backend is organized into modular blueprints, each serving specific user roles (Admin, Team Leader, Member).
- **Database Layer:** A PostgreSQL database stores persistent data such as user credentials, tasks, teams, comments, and attachments. The schema is defined in SQL files under `database_sql/` and enforced through foreign key relationships.

The overall architecture follows a **service-oriented design**, enabling independent management of core modules:

- **User Management Service** — Handles authentication, authorization, and role assignment.
- **Team Management Service** — Manages team creation, membership, and leadership roles.
- **Task Management Service** — Handles task creation, updates, deadlines, and attachments.
- **Frontend Interface** — Provides dynamic HTML templates and serves as the communication layer between users and the backend.

Containerization with Docker ensures that the entire system — including the Flask backend, PostgreSQL database, and supporting libraries — can be launched reproducibly across environments with minimal configuration. This design supports rapid deployment, simplifies maintenance, and allows easy scaling by adding containers or balancing workloads.

2. System Architecture

2.1 General Design

The system follows a modular, service-oriented architecture built primarily with the **Flask** web framework in Python. It separates concerns across layers for **user interaction**, **business logic**, and **data management**. The architecture consists of three main tiers:

- **Frontend Layer:** Delivers the user interface through HTML templates, CSS styling, and JavaScript scripts. It allows users to interact with the system via a web browser and communicates with the backend using HTTP requests.
- **Backend Layer:** Implemented using Flask, it handles routing, authentication, data processing, and logic orchestration. It exposes RESTful API endpoints defined within the `routes` module.
- **Data Layer:** Manages persistent storage through SQL scripts and a relational database schema stored in the `database_sql` directory. The `db.py` and `config.py` modules manage database connections and configuration parameters.

The architecture supports **containerized deployment** via Docker, enabling consistent behavior across environments and simplified scaling. The system services (backend, database) communicate internally using network bridges defined in `docker-compose.yml`.

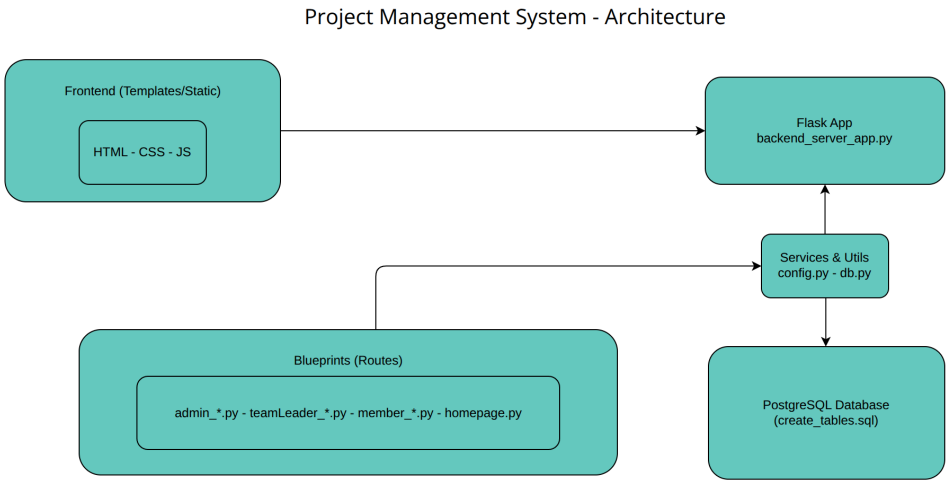


Figure 2.1: High-level architecture of the system showing main layers and interactions.

2.2 Main Components

The project directory is organized into several essential folders, each with a distinct responsibility:

- **database_sql/** – Contains SQL scripts defining the schema, table structures, and initial data for the application database. This folder ensures the database can be recreated consistently during deployment or testing.
- **routes/** – Defines the application's REST API endpoints and Flask routes. Each route corresponds to a specific functionality, such as user management, task handling, or team collaboration.
- **utils/** – Includes reusable utility functions that support the main logic of the system, such as validation helpers, date formatting, and database access abstractions.
- **templates/** – Stores HTML template files used by Flask's rendering engine (`Jinja2`). These templates dynamically display user and task information, supporting role-based interface rendering.
- **static/** – Contains all static assets used by the frontend interface, including the `.css` files for styling, the `.js` files that handle client-side logic and interactivity, as well as the `uploads/` directory that stores user-uploaded attachments (such as files added to tasks or comments).

- **config.py** – Centralized configuration file containing environment variables, database URIs, and API keys.
- **db.py** – Handles database initialization and connection management, providing helper functions for executing SQL queries.
- **backend_server_app.py** – The main entry point of the Flask backend server; initializes routes, configuration, and app context.
- **requirements.txt** – Lists all Python dependencies required for running the system.
- **source_run_flask.sh** – Shell script that runs the backend service in a development or production mode.
- **push_to_github.sh** – Utility script for committing and pushing project updates to version control.

This modular organization improves maintainability and scalability, allowing individual components to evolve independently without breaking the system structure.

2.3 Containerization and Deployment

The system leverages **Docker** and **Docker Compose** for simplified container-based deployment. Each service (e.g., Flask backend, database) runs inside its own isolated container, ensuring reproducible environments.

- **Dockerfile** – Defines the image for the Flask backend, including dependency installation, environment setup, and startup commands.
- **docker-compose.yml** – Describes multiple containers as services (e.g., backend, database), manages network links, and specifies persistent volume mounts for data storage.

The containerized architecture provides:

- Consistent environments between development, testing, and production.
- Isolation of dependencies for backend and database services.
- Scalability through container orchestration and versioned builds.

3. Database Design

3.1 Database Overview

The database serves as the persistent storage layer of the system, responsible for managing all core entities such as users, teams, tasks, comments, and their interrelations. A **PostgreSQL relational database** is used to ensure structured, reliable, and consistent handling of application data.

The schema is designed following the principles of **Third Normal Form (3NF)**, aiming to reduce redundancy and enforce clear logical dependencies between entities. Primary and foreign key constraints define the relationships between tables, supporting the operations performed by the microservices, such as user management, team assignment, task updates, and comment handling.

All database-related files are stored inside the `database_sql/` directory, which contains:

- **create_tables.sh** – A shell script intended for **local execution (non-Docker)**. It installs PostgreSQL when needed, creates the database user `nefos`, sets up the database `project_db`, and imports the schema and initial data from `database.sql`. It is used when running the system directly on a host machine.
- **database.sql** – The primary SQL script containing the full database schema. It defines all tables, relationships, constraints, indexes, and includes the initial seed data required for system initialization.
- **docker_init_restore.sh** – A helper script executed automatically when the PostgreSQL Docker container starts. It restores the schema and initial data inside the container, ensuring a consistent database state for Docker-based deployments.

This structure enables reliable database setup both locally and within the containerized environment, ensuring that backend services can seamlessly interact with the stored data for authentication, authorization, team management, task operations, and comment storage.

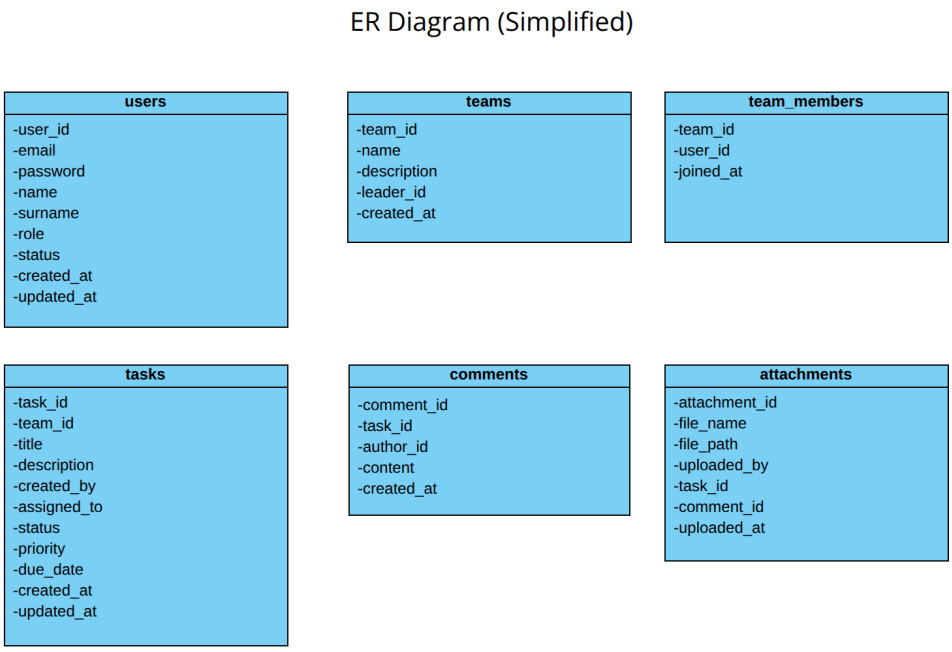


Figure 3.1: Simplified Entity–Relationship (ER) diagram showing the core database entities—users, teams, team_members, tasks, comments, and attachments—together with their primary attributes.

3.2 Entity–Relationship Model

The Entity–Relationship (ER) model describes how data entities interact and relate to each other within the system. The model reflects real-world relationships between users, teams, and tasks. Each entity encapsulates a distinct concept in the application domain, and relationships define the logical connections among them.

The main entities are as follows:

- **User:** Represents a registered user in the system. Each user has a unique identifier, authentication credentials, and role (e.g., administrator, project manager, member).
- **Team:** Groups users under a collaborative unit. Each team has a name, description, and is managed by a specific user (team leader).
- **Task:** Represents an individual work item or project activity. Tasks have attributes such as title, description, status, deadline, and priority. Each task is assigned to one or more users and belongs to a team.
- **Project:** Represents a higher-level organizational entity grouping tasks and teams. It defines overall goals, timelines, and ownership.

- **Role:** Defines access permissions and responsibilities within the system (e.g., Admin, Developer, Viewer).
- **Attachment:** Stores metadata for uploaded files associated with a task or project, linking the filesystem to database references.

The relationships between the entities in the database schema are:

- A **User** can belong to multiple **Teams** (via the `team_members` mapping table).
- A **Team** can have multiple **Users** as members.
- A **Team** can have multiple **Tasks**.
- A **Task** is assigned to at most one **User**, but a user may be assigned many tasks.
- A **Task** can have multiple **Comments**.
- A **Comment** is written by a single **User**.
- An **Attachment** can belong either to a **Task** or to a **Comment**, depending on where it was uploaded.

Primary keys and foreign keys enforce referential integrity. Composite keys are used in junction tables to manage many-to-many relationships between users and tasks.

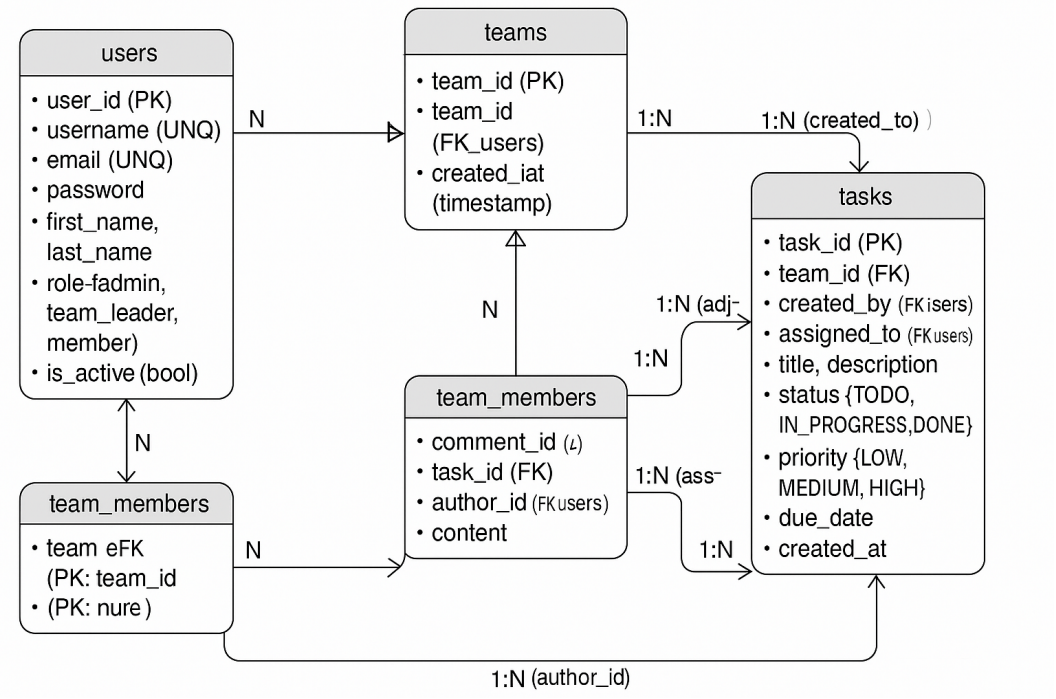


Figure 3.2: Complete ER diagram of the system including all entities, relationships, and cardinalities.

3.3 Tables Description

The database schema consists of several interrelated tables designed for relational integrity, scalability, and efficient data access. Tables use integer primary keys, foreign key references, and constraints such as `NOT NULL`, `UNIQUE`, and composite primary keys for mapping relations.

Table 3.1: Users Table

Column	Type	Constraints	Description
user_id	INT	PK, AUTO_INCREMENT	Unique identifier for each user.
username	VARCHAR(50)	UNIQUE, NOT NULL	Login username chosen by the user.
email	VARCHAR(100)	UNIQUE, NOT NULL	User email address.
password	TEXT	NOT NULL	Password hash for authentication.
name	VARCHAR(50)	NOT NULL	User's first name.
surname	VARCHAR(50)	NOT NULL	User's last name.
role	ENUM(user_role)	NOT NULL, DEFAULT 'MEMBER'	Application role (ADMIN, TEAM_LEADER, MEMBER).
status	ENUM(user_status)	NOT NULL, DEFAULT 'PENDING'	Account status (ACTIVE, INACTIVE, PENDING).
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp when the user was created.
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp of the last user record update.

Table 3.2: Teams Table

Column	Type	Constraints	Description
team_id	INT	PK, AUTO_INCREMENT	Unique identifier for each team.
name	VARCHAR(100)	UNIQUE, NOT NULL	Name of the team.
description	TEXT	NULL	Optional description of the team.
leader_id	INT	FK → users(user_id), NULL	User who leads the team; set to `NULL` if the leader is removed.
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp when the team was created.

Table 3.3: Team_Members (Mapping Table)

Column	Type	Constraints	Description
team_id	INT	PK (composite), FK → teams(team_id)	Team to which the user belongs.
user_id	INT	PK (composite), FK → users(user_id)	User that is a member of the team.
joined_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Date and time when the user joined the team.

Table 3.4: Tasks Table

Column	Type	Constraints	Description
task_id	INT	PK, AUTO_INCREMENT	Unique identifier for each task.
team_id	INT	FK → teams(team_id), NULL	Team responsible for this task.
title	VARCHAR(200)	NOT NULL	Task title or short label.
description	TEXT	NULL	Detailed information about the task.
created_by	INT	FK → users(user_id), NULL	User who created the task.
assigned_to	INT	FK → users(user_id), NULL	User to whom the task is assigned.
status	ENUM(task_status)	NOT NULL, DEFAULT 'TODO'	Task state (TODO, IN_PROGRESS, DONE).
priority	ENUM(task_priority)	NOT NULL, DEFAULT 'MEDIUM'	Task priority level (LOW, MEDIUM, HIGH).
due_date	DATE	NULL	Optional task due date.
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp when the task was created.
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp of the last task update.

Table 3.5: Comments Table

Column	**Type**	**Constraints**	**Description**
comment_id	INT	PK, AUTO_INCREMENT	Unique identifier for each comment.
task_id	INT	FK → tasks(task_id), NULL	Task to which the comment belongs.
author_id	INT	FK → users(user_id), NULL	User who wrote the comment.
content	TEXT	NOT NULL	Text content of the comment.
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp when the comment was created.

Table 3.6: Attachments Table

Column	Type	Constraints	Description
attachment_id	INT	PK, AUTO_INCREMENT	Unique identifier for each attachment.
file_name	VARCHAR(255)	NOT NULL	Original name of the uploaded file.
file_path	TEXT	NOT NULL	File location in the storage system.
uploaded_by	INT	FK → users(user_id), NULL	User who uploaded the file.
task_id	INT	FK → tasks(task_id), NULL	Related task (if the file is attached directly to a task).
comment_id	INT	FK → comments(comment_id), NULL	Related comment (if the file is attached to a comment).
uploaded_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp when the file was uploaded.

4. Roles and Functionality

4.1 User Roles

The system uses a role-based access model with three user roles stored directly in the `users.role` field: `ADMIN`, `TEAM_LEADER`, and `MEMBER`. Each role determines the level of access to teams, tasks, and management actions.

- **Administrator (Admin):** Full access to all teams, users, and tasks. Can create or delete users, assign roles, manage teams, and remove any system content.
- **Team Leader:** Leads a team, creates tasks, assigns tasks to members, updates task details, and manages team membership.
- **Team Member:** Views tasks belonging to their team, updates tasks assigned to them, writes comments, and uploads attachments.

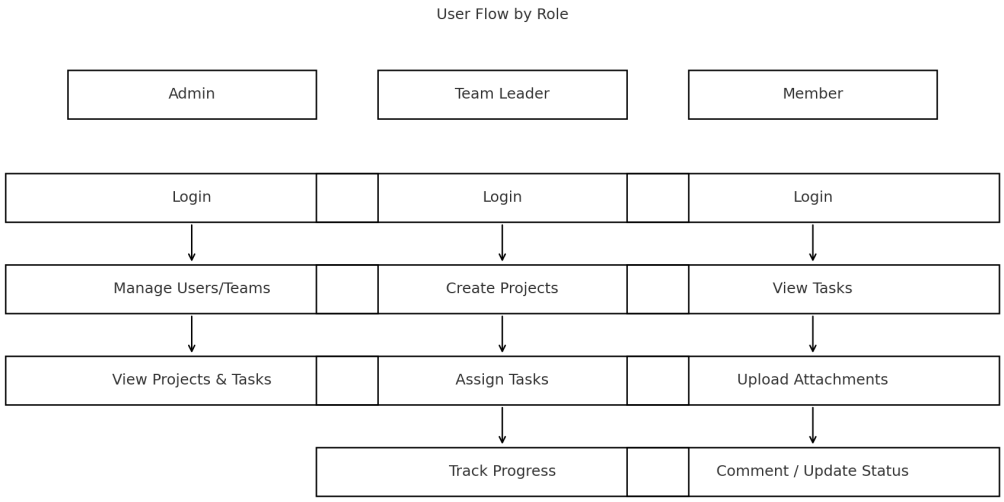


Figure 4.1: Role-based access flow.

4.2 Core Functionalities

4.2.1 Team Management

- Admins and Team Leaders can create teams and manage membership.
- Team visibility depends on the user's role (admin sees all; leaders and members see only their teams).

4.2.2 Task Management

- Tasks are created by Admins or Team Leaders and assigned to a single member.
- Each task includes title, description, priority, deadline, status, and assigned user.
- Members update their own tasks' status and may add comments or attachments.

4.2.3 Comments and Attachments

- Users can comment on tasks for communication.
- Attachments (documents, images) are uploaded through comments or task updates and stored in `static/uploads/`.

4.2.4 System Logging

- Key actions (task updates, comments, user operations) can be recorded for tracking and debugging.

Overall, the platform enables structured collaboration within teams through clear role-based permissions, task workflows, and communication support.

5. Technologies and Tools

The development of the project management system was based on a modern, containerized software stack that ensures scalability, maintainability, and ease of deployment. Each tool and technology was carefully selected to fulfill a specific role within the system's architecture, ranging from backend logic to frontend presentation and cloud-based deployment.

5.1 Backend Technologies

Python

The core application logic is implemented in **Python 3.12**, chosen for its readability, extensive libraries, and strong community support. Python provides an ideal balance between rapid development and high-level abstraction, making it suitable for RESTful APIs, database manipulation, and server-side automation.

Flask Framework

Flask is the primary web framework used for developing the backend. It offers a lightweight and modular design, allowing flexible integration with various extensions. The Flask app is initialized in `backend_server_app.py` and defines routes, authentication, and configuration management. Key reasons for choosing Flask include:

- Simple setup and minimal dependencies.
- Clear separation between routing, templating, and configuration.
- Built-in development server and debugger.
- Easy integration with SQL databases and Docker containers.

Jinja2 Templating Engine

Flask uses the **Jinja2** engine for rendering dynamic HTML pages. Templates stored in the `templates/` directory are populated with server-side data and rendered for each user request. This enables role-based interfaces, such as different dashboards for administrators and team members.

5.2 Frontend Technologies

HTML, CSS, and JavaScript

The system's frontend is built using standard web technologies:

- **HTML5** provides the document structure and defines the user interface elements.
- **CSS3** (stored in the `static/css/` folder) is used for styling, ensuring responsive design and consistent visual presentation.
- **JavaScript** (in the `static/script/` folder) adds interactivity, handles form validation, and manages asynchronous requests (AJAX) between the frontend and backend.

Bootstrap Framework

The frontend uses the **Bootstrap** CSS framework to provide a consistent, responsive, and user-friendly interface. Bootstrap components such as forms, buttons, and layout grids simplify the design process and ensure compatibility across devices.

5.3 Database Technologies

PostgreSQL

The system employs a relational database (e.g., **PostgreSQL** or **MySQL**) for structured data storage. It ensures transactional integrity, supports complex queries, and is well integrated with Flask through libraries like `psycopg2` or `SQLAlchemy`. Database schema and initialization scripts are located in the `database_sql/` directory.

Database Access Layer

The system interacts with the database using raw SQL queries through a lightweight connection layer implemented in `db.py`. This approach offers full control over the executed SQL statements and keeps the backend dependencies minimal.

5.4 Containerization and Deployment Tools

Docker

Docker is used to containerize the backend service and its dependencies, ensuring that the application behaves identically across all environments. The configuration for building the container image is defined in the `Dockerfile`. Docker enables lightweight, isolated environments that improve portability and reduce dependency conflicts.

Docker Compose

Docker Compose orchestrates multiple containers—such as the Flask backend and the database—defined in the `docker-compose.yml` file. It simplifies deployment by allowing the entire stack to be launched with a single command:

```
1 docker compose up --build
```

Listing 5.1: Running the system with Docker Compose

5.5 Development and Version Control Tools

Visual Studio Code

Visual Studio Code (VS Code) was the primary Integrated Development Environment (IDE) for the project. Its built-in debugging tools, Python extensions, and Git integration streamlined both coding and testing workflows.

Git and GitHub

Version control is handled through **Git**, with remote repositories hosted on **GitHub**. This setup provides collaborative development capabilities, allowing code versioning,

pull requests, and issue tracking. The repository includes utility scripts such as `push_to_github.sh` for automated updates.

```
#!/bin/bash

# ---------------------------------------------
# Automated Git Commit & Push Script
# ---------------------------------------------

# Stop the script on any error
set -e

# Customize your commit message (or pass it as an argument)
COMMIT_MSG=${1:-"Auto-commit: update project files"}

# Print header
echo "---------------------------------------------"
echo "Starting Git Auto Push..."
echo "---------------------------------------------"

# Add all files
git add .

# Commit changes
git commit -m "$COMMIT_MSG"

# Pull latest remote changes (avoid conflicts)
git pull origin main --rebase

# Push to GitHub
git push origin main

echo "---------------------------------------------"
echo "Successfully pushed to GitHub!"
echo "---------------------------------------------"
```

Listing 5.2: Automated Git push helper (push_to_github.sh)

5.6 Summary of the Technology Stack

Table 5.1 provides an overview of the complete technology stack used throughout the project.

Table 5.1: Technology Stack Overview

Category	**Technology**	**Purpose / Description**
Programming Language	Python 3.12	Core development language for backend services.
Web Framework	Flask	Lightweight framework for routing and API logic.
Templating Engine	Jinja2	Dynamic HTML rendering with embedded Python logic.
Frontend	HTML5, CSS3, JavaScript, Bootstrap	User interface and responsive design.
Database	PostgreSQL / MySQL	Persistent data storage and query processing.
Containerization	Docker, Docker Compose	Container-based deployment and service orchestration.
Server	Gunicorn	WSGI server for production hosting.
Version Control	Git, GitHub	Code versioning, collaboration, and repository management.
IDE	Visual Studio Code	Development, debugging, and testing environment.
Dependency Management	pip, virtualenv	Python package management and environment isolation.

Together, these technologies create a cohesive and modular ecosystem that supports continuous development, integration, and deployment. This combination enables high maintainability, reliable version control, and the ability to scale or extend the system with minimal configuration changes.

6. Implementation

6.1 Backend

Project Structure and Blueprints

The backend is implemented in **Flask** and organized using Blueprints to separate logic by role and feature. The main application entry point is `backend_server_app.py`, while individual functionalities are defined inside the `routes/` directory. Representative modules include:

- `routes/homepage.py` — public homepage and role selection.
- `routes/admin_authenticate.py`, `routes/teamLeader_authenticate.py` — role-specific authentication flows.
- `routes/admin_mainpage.py`, `routes/teamLeader_mainpage.py`, `routes/member_mainpage.py` — dashboards and business logic per role.
- `routes/admin_teamLeader_member_options.py` — shared sign-in/sign-up options and user-role navigation.

Each Blueprint defines its own routes, templates, and (where necessary) database calls. A minimal example (from the homepage) follows:

```
from flask import Blueprint, render_template

homepage_bp = Blueprint("homepage", __name__)

@homepage_bp.route("/")
def index():
    """Public landing page."""
    return render_template("index.html")
```

Listing 6.1: Minimal route example (homepage)

Configuration and Environment

Application-wide configuration resides in `config.py`. It defines the Flask secret key and database parameters using environment variables (with safe defaults for development). Example (sensitive values in env):

```python
import os

# Flask Secret Key
SECRET_KEY = "supersecretkey"

# Uploads Configuration
UPLOAD_FOLDER = os.path.join("static", "uploads")
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif', 'pdf'}

# Database Configuration
DB_CONFIG = {
    "host": os.getenv("DB_HOST", "localhost"),
    "database": os.getenv("DB_NAME", "project_db"),
    "user": os.getenv("DB_USER", "postgres"),
    "password": os.getenv("DB_PASSWORD", "xotour")
}
```

Listing 6.2: Configuration excerpt (config.py)

Database Layer and Access

Direct SQL access is implemented with **psycopg2**. The helper `db.py` centralizes connection handling to PostgreSQL and exposes a function to obtain a connection/cursor pair. Example pattern:

```python
def get_db_connection():
    """
    Establishes a connection to the PostgreSQL database.
    Returns a connection object or a string with the error message.
    """
    try:
        conn = psycopg2.connect(
            host=DB_CONFIG["host"],
            database=DB_CONFIG["database"],
            user=DB_CONFIG["user"],
            password=DB_CONFIG["password"]
```

```
        )
        return conn
    except Exception as e:
        return str(e)
```

Listing 6.3: Database connection pattern (db.py)

Routes use parameterized queries for safety. For instance, a member task view might query by the current user and task id:

```
rows = fetch_all("""
    SELECT t.task_id, t.title, t.description, t.status, t.priority, t.
    ↪ due_date
    FROM tasks t
    JOIN team_members tm ON tm.team_id = t.team_id
    WHERE tm.user_id = %s
    ORDER BY t.due_date NULLS LAST, t.priority DESC
""", (current_user_id,))
```

Listing 6.4: Example: parameterized query in a route

Upload Handling

File uploads are validated via `utils/file_utils.py` using the configured whitelist:

```
from config import ALLOWED_EXTENSIONS

def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in
    ↪ ALLOWED_EXTENSIONS
```

Listing 6.5: Upload type validation (utils/file_utils.py)

Database Initialization

The database schema is provided as a PostgreSQL dump / DDL in `database_-sql/database.sql` and can be applied via a helper script (e.g., `create_tables.sh`) or a Docker entrypoint. Typical Compose orchestration ensures the database container is healthy before the app starts.

Service Endpoints (Overview)

Key endpoints are grouped by role-specific Blueprints. The table lists representative routes discovered from the codebase (exact names may be extended in the full implementation).

Table 6.1: Representative backend endpoints by role

Route	Method	Module	Purpose
`/`	GET	`homepage.py`	Public landing page.
`/admin/login`, `/admin/signup`	GET/POST	`admin_authenticate.py`	Admin authentication views and handlers.
`/member/login`, `/member/signup`	GET/POST	`member_authenticate.py`	Member authentication views and handlers.
`/admin` (dashboard)	GET	`admin_mainpage.py`	Admin home, overviews of teams/tasks.
`/teamLeader` (dashboard)	GET	`teamLeader_mainpage.py`	Team leader views, manage teams/-tasks.
`/member` (dashboard)	GET	`member_mainpage.py`	Member views, assigned tasks, updates.
`/teamLeader-manageTeams`	GET/POST	`teamLeader_mainpage.py`	Create/edit teams, assign members.

Login Flow (Sequence)

Figure 6.1 illustrates a typical login flow (e.g., Member). It applies analogously to Admin / Team Leader with role checks and redirects.

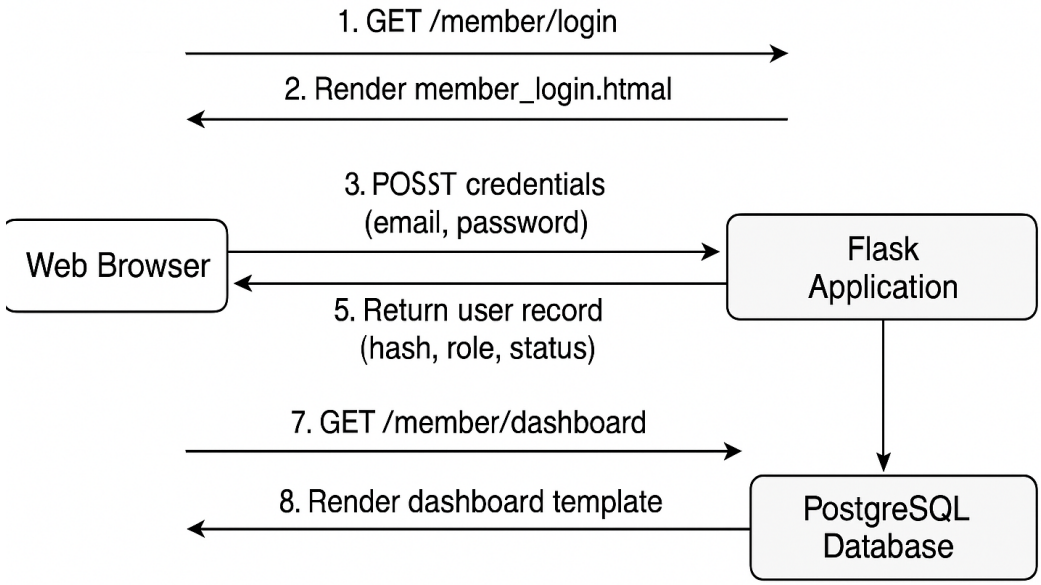


Figure 6.1: Member login sequence showing interaction between Web Browser, Flask Application, and PostgreSQL Database.

6.2 Frontend

Templates

Dynamic pages are rendered via **Jinja2** templates in `templates/`. The project contains role-specific templates for onboarding and dashboards, for example:

- `index.html` (homepage), `admin_login.html`, `admin_signup.html`, `member_login.html`, `member_signup.html`, `teamLeader_login.html`, `teamLeader_signup.html`.
- Task/Team views: `member_viewTask.html`, `teamLeader_viewTask.html`, `member_viewTeam.html`, etc.

Templates use standard Jinja constructs:

- `{{ variable }}` output from route context.
- `{% for row in rows %}` loops to render task lists.
- `{% if role == 'ADMIN' %}` conditional sections per role.

Static Assets

Static files are organized under `static/`:

- `static/css/` — global styles, layout utilities, responsive rules.
- `static/script/` — JavaScript for form validation, AJAX calls, and UI behaviors.
- `static/images/` — logos, icons, and decorative imagery.
- `static/uploads/` — user-submitted files linked from the `attachments` table.

Forms and UX

Login/Signup pages submit to role-specific endpoints (POST). Validation errors are rendered inline (e.g., missing fields, invalid credentials). Dashboards present tabular task lists with actions (view, update status, attach file). Team Leader and Admin views include management controls (create team, assign members, review tasks).

Sessions, Roles, and Status

Authentication sets a server-side session (using `SECRET_KEY`). After successful login, routes use the session's `user_id` and `role` to enforce access. The database schema encodes authorization state directly:

Routes gate actions with role checks (e.g., Admin may manage all teams; Team Leaders manage their own team; Members update only their assigned tasks). Status checks (e.g., deny login for `PENDING`) strengthen account control.

Input Validation and Query Safety

All database operations use **parameterized queries**. Inputs from forms are validated server-side (type, range, required fields). Uploaded files are restricted via `ALLOWED_EXTENSIONS`; paths are sanitized before saving to `static/uploads/`.

Data Access and Ownership

Read/write queries include ownership constraints (e.g., member can only see tasks in teams they belong to). Attachment and comment actions check that the user is either the author, assignee, team member, or an elevated role (Team Leader/Admin).

Error Handling and Logging

Database exceptions are trapped and rendered as user-friendly messages (with safe, generic wording). Sensitive details are not exposed. Server logs (INFO/WARN/ERROR) record context for debugging and auditing (e.g., failed logins, forbidden access attempts).

6.3 Data Model (Operational View)

Entities Implemented

The deployed schema (from `database_sql/database.sql`) contains the following core tables: `users`, `teams`, `team_members`, `tasks`, `comments`, `attachments`. Below is a compact view of two important entities as implemented.

Table 6.2: Implemented table (tasks)

Column	**Type**	**Notes**
`task_id`	`integer, PK`	Unique id.
`team_id`	`integer, FK`	Task belongs to a team.
`title`	`varchar(200), NOT NULL`	Short name.
`description`	`text`	Details.
`created_by`, `assigned_to`	`integer`	User ids, FK to `users`.
`status`	`public.task_status, NOT NULL`	Enum (`TODO`, `IN_PROGRESS`, `DONE`, ...).
`priority`	`public.task_priority, NOT NULL`	Enum (`LOW`, `MEDIUM`, `HIGH`).
`due_date`	`date`	Optional deadline.
`created_at`, `updated_at`	`timestamp`	Audit fields.

6.4 Putting It Together (Dockerized Runtime)

The application and database run as containers orchestrated by `docker-compose.yml`. The Flask app depends on the Postgres service and loads its configuration from environment variables. At startup:

1. PostgreSQL container initializes (with mounted volume for persistence).
2. Flask container builds from `Dockerfile` (installs `requirements.txt`).
3. The backend waits for DB availability, then serves requests (e.g., via Gunicorn in production).

This organization ensures reproducible development and deployment, clear separation of concerns, and a maintainable codebase structured by role and feature.

7. Installation and Execution

There are three different ways to install and run the system, depending on the user's workflow and environment:

1. **Setup from GitHub (Local Development Mode):** The full project can be cloned from GitHub and executed locally using a Python virtual environment and the Flask development server. This method is ideal for testing, debugging, and extending the application's source code.
2. **PgAdmin Desktop (Database Initialization and Inspection):** PgAdmin is used to manage and inspect the PostgreSQL database. Through the graphical interface, users can register the running PostgreSQL server, execute SQL queries, and verify that all tables and relations are created correctly. This method supports database preparation and validation during development.
3. **Docker Deployment (Containerized Execution):** The system can be executed entirely through Docker and Docker Compose, which automatically deploy the backend service and PostgreSQL database in isolated containers. This provides the most consistent and production-ready environment, requiring only Docker to run the whole application stack with a single command.

These three approaches allow the system to be executed in flexible environments, ranging from traditional local development to fully containerized deployment.

7.1 Setup from Github

This section describes the manual setup of the system on a local machine using Python's virtual environment and the Flask development server. This approach is primarily used during development, debugging, and academic demonstrations. The following steps assume the user is running a Unix-like environment (Linux or macOS). Windows users can follow equivalent steps using PowerShell or WSL.

Step 1: Clone the Repository

The full source code is stored in a Git repository. Clone it using:

```
git clone https://github.com/username/project-management-system.git
cd project-management-system
```

Listing 7.1: Cloning the project repository

Step 2: Create and Activate a Virtual Environment

To isolate dependencies, create a virtual environment inside the project folder:

```
python3 -m venv venv
source venv/bin/activate
```

Listing 7.2: Setting up Python virtual environment

Step 3: Install Dependencies

The Python dependencies are defined in the `requirements.txt` file:

Install all dependencies with:

```
pip install -r requirements.txt
```

Step 4: Configure Environment Variables

Configuration values such as database credentials and secret keys can be adjusted in `config.py`:

```
import os

SECRET_KEY = "supersecretkey"
UPLOAD_FOLDER = os.path.join("static", "uploads")
```

Listing 7.3: config.py excerpt

Step 5: Initialize the Database

The database schema is defined in the SQL file `database_sql/database.sql`. You can load it manually into PostgreSQL:

```
psql -U postgres -d project_db -f database_sql/database.sql
```

Listing 7.4: Importing the database schema

Alternatively, run the included shell script:

```
chmod +x database_sql/create_tables.sh
./database_sql/create_tables.sh
```

Listing 7.5: run the executable file create_tables.sh

```
#!/bin/bash
# ----------------------------------------
# PostgreSQL Server and Database Setup Script
# Creates user 'nefos' (password: xotour) and database 'project_db'
# Automatically installs PostgreSQL if not present and imports database.sql
# ----------------------------------------

DB_USER="nefos"
DB_PASS="xotour"
DB_NAME="project_db"
SQL_FILE="database.sql"

echo "Starting PostgreSQL setup..."
echo "----------------------------------------"

# Check if PostgreSQL is installed
if ! command -v psql > /dev/null 2>&1; then
  echo "PostgreSQL not found. Installing..."
  sudo apt update -y
  sudo apt install -y postgresql postgresql-contrib
else
  echo "PostgreSQL is already installed."
fi

# Ensure PostgreSQL service is running
echo "Ensuring PostgreSQL service is running..."
sudo systemctl enable postgresql
sudo systemctl start postgresql

# Switch to postgres user and execute SQL commands
sudo -u postgres psql <<EOF
-- Drop database if exists
```

```
DROP DATABASE IF EXISTS ${DB_NAME};
-- Drop role if exists
DROP ROLE IF EXISTS ${DB_USER};

-- Create user and database
CREATE ROLE ${DB_USER} WITH LOGIN PASSWORD '${DB_PASS}';
CREATE DATABASE ${DB_NAME} OWNER ${DB_USER};
GRANT ALL PRIVILEGES ON DATABASE ${DB_NAME} TO ${DB_USER};
EOF

echo "Database '${DB_NAME}' and user '${DB_USER}' created successfully."

# Import schema & data from SQL file
if [ -f "$SQL_FILE" ]; then
  echo "Importing schema and data from ${SQL_FILE} ..."
  sudo -u postgres psql -d "${DB_NAME}" -f "${SQL_FILE}" > /dev/null 2>&1
  if [ $? -eq 0 ]; then
    echo "Database import completed successfully."
  else
    echo "Error occurred during import. Check SQL file for issues."
  fi
else
  echo "SQL file ${SQL_FILE} not found. Skipping import."
fi

# Test connection with created user
echo "Testing connection..."
PGPASSWORD="${DB_PASS}" psql -U "${DB_USER}" -d "${DB_NAME}" -h localhost -c
    ↪ "\dt" > /dev/null 2>&1
if [ $? -eq 0 ]; then
  echo "Connection test successful! Setup completed."
else
  echo "Connection test failed. Check credentials or PostgreSQL status."
fi

echo "----------------------------------------"
echo "Setup finished. Database '${DB_NAME}' ready for use."
```

Listing 7.6: create_tables.sh

Step 6: Start the Flask Server

You can run the system using Flask's built-in development server or via the provided script.

```
#!/bin/bash
source venv/bin/activate
export FLASK_APP=backend_server_app.py
export FLASK_ENV=development
flask run --host=0.0.0.0 --port=5000
```

Listing 7.7: source_run_flask.sh

Run the application with:

```
./source_run_flask.sh
```

Once launched, open your browser at:

http://localhost:5000

The homepage will appear, allowing you to choose between Admin, Team Leader, or Member login options.

7.2 PgAdmin Desktop

PgAdmin 4 was used as the primary graphical administration tool for interacting with the PostgreSQL database during development. While the database was running inside a Docker container, PgAdmin provided an intuitive interface to inspect schemas, browse tables, execute SQL queries, and verify that the backend logic was correctly reflected in the stored data.

Figure 7.1: PgAdmin server registration interface used.

Installation

PgAdmin can be installed on Linux, Windows, or macOS. For Debian/Ubuntu-based systems, installation is typically done through the official PgAdmin repository:

```
sudo apt install pgadmin4
```

After installation, PgAdmin launches as a local web application and can be accessed through the default system browser.

Connecting PgAdmin to the Dockerized Database

To connect PgAdmin to the PostgreSQL instance running inside Docker, a new server registration must be created. This connection allows PgAdmin to interact directly with the database container using the port exposed in `docker-compose.yml`.

- **Step 1:** Open PgAdmin → `Create` → `Server`.

- **Step 2:** In the **General** tab, set:
  - **Name:** `Project PostgreSQL`
- **Step 3:** In the **Connection** tab, configure:
  - **Host:** `localhost`
  - **Port:** `5432`
  - **Username:** `postgres`
  - **Password:** `xotour`
  - **Save Password:** Enabled
- **Step 4:** Click `Save` to register the server.

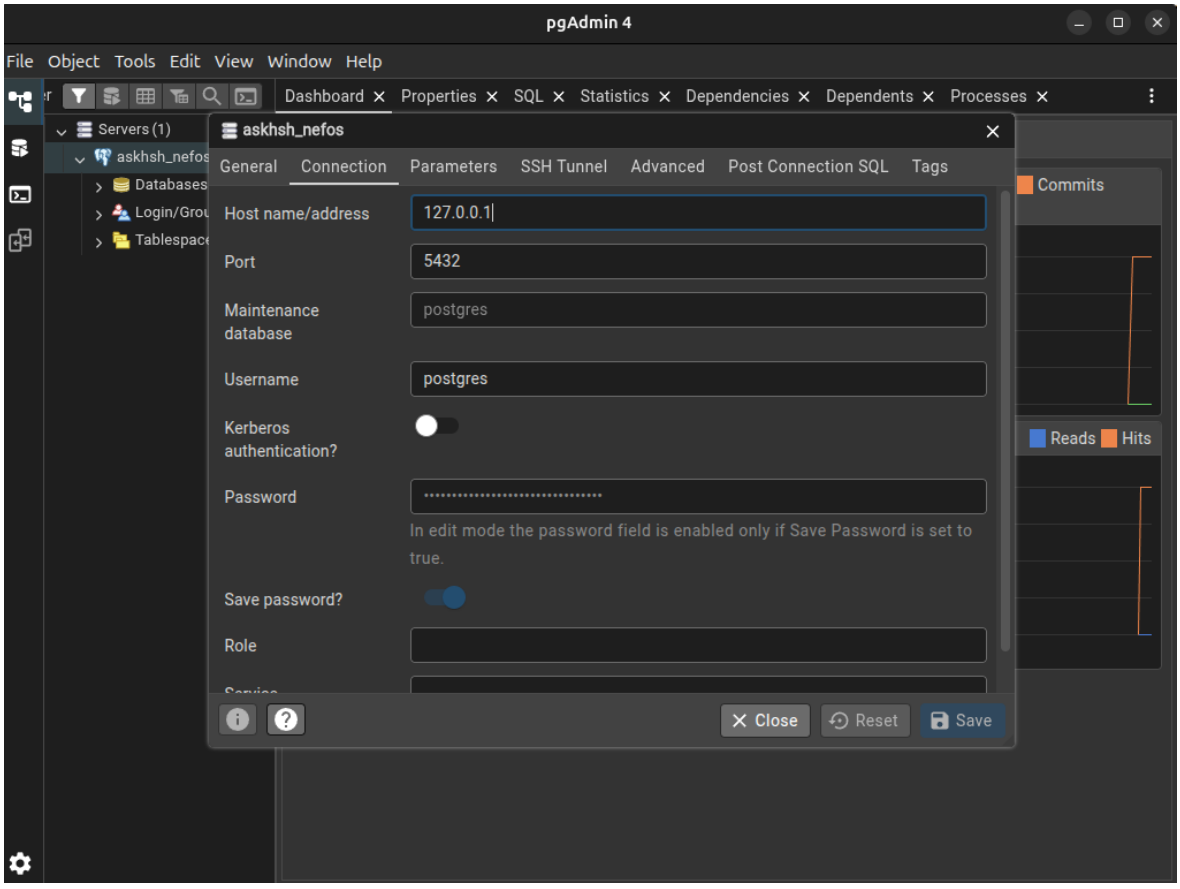


Figure 7.2: PgAdmin server registration interface settings.

Once connected, PgAdmin displays the full database tree, including:

- Tables and their relationships
- Constraints and indexes

- Stored data for users, teams, tasks, comments, and attachments
- Active sessions and connection statistics
- The SQL *Query Tool* for executing manual queries

Initializing the Database Schema

Before connecting to the database through PgAdmin, it is necessary to create all required tables and initialize the schema. This is done by running the executable script located in the `database_sql/` directory:

```
bash database_sql/create_tables.sh
```

The script automatically loads the SQL schema into the PostgreSQL container and prepares the database with the correct structure. After running it successfully, all tables (users, teams, tasks, comments, attachments, and mapping tables) become available for inspection in PgAdmin.

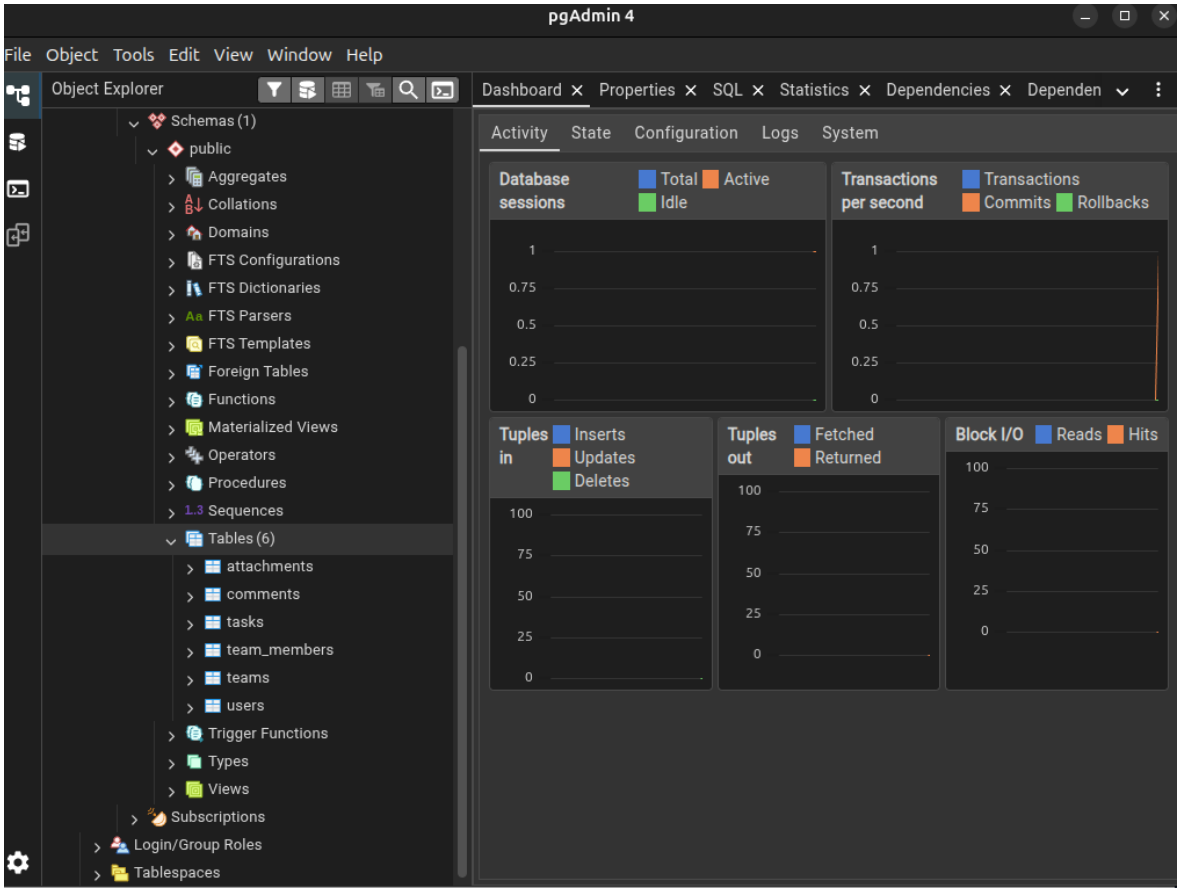


Figure 7.3: PgAdmin view of the database tables, including users, teams, tasks, comments, and attachments.

Usage in Development

Throughout development, PgAdmin was used extensively to:

- Verify that SQL initialization scripts created the correct schema.
- Inspect foreign key relationships and ensure referential integrity.
- Debug issues related to authentication, task assignment, and team membership.
- Confirm that backend routes performed the expected CRUD operations.
- Execute manual SQL `SELECT`, `INSERT`, `UPDATE`, and `DELETE` queries.

This greatly accelerated debugging and validation, providing real-time insight into the state of the containerized PostgreSQL database.

7.3 Docker Deployment

For a production-ready, isolated setup, the system uses Docker containers to encapsulate both the Flask backend and the PostgreSQL database. The container images are configured via a `Dockerfile` and orchestrated with `docker-compose.yml`.

Dockerfile

The `Dockerfile` defines the environment for the Flask backend:

```
FROM python:3.12-slim

# Set working directory
WORKDIR /app

# Copy all project files into the container
COPY . /app

# Install required system tools and Python dependencies
RUN apt-get update && apt-get install -y netcat-traditional && \
    pip install --no-cache-dir -r requirements.txt && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

# Expose Flask port
```

```
EXPOSE 5000

# Run Flask app
CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

Listing 7.8: Dockerfile

Docker Compose Configuration

The `docker-compose.yml` file orchestrates the multi-container environment:

```
version: "3.9"

services:
  db:
    image: postgres:16
    container_name: project_db
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: xotour
      POSTGRES_DB: postgres
    ports:
      - "5433:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./database_sql/docker_init_restore.sh:/docker-entrypoint-initdb.d/
    ↪ docker_init_restore.sh
      - ./database_sql/database.sql:/docker-entrypoint-initdb.d/database.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      retries: 5
      timeout: 5s
    restart: unless-stopped

  web:
    build: .
    container_name: project_app
    depends_on:
      db:
        condition: service_healthy
    ports:
```

```
      - "5000:5000"
    volumes:
      - .:/app
    environment:
      FLASK_APP: backend_server_app.py
      FLASK_ENV: development
      DB_HOST: db
      DB_USER: postgres
      DB_PASSWORD: xotour
      DB_NAME: postgres
    command: flask run --host=0.0.0.0 --port=5000
    restart: unless-stopped

volumes:
  pgdata:
```

Listing 7.9: docker-compose.yml

Running the Application with Docker

Once the configuration files are in place, you can build and start the containers:

```
# Build images and start containers
docker compose up --build

# (Optional) Run in detached mode
docker compose up -d
```

Listing 7.10: Docker Compose build and run commands

Docker will automatically:

1. Pull the latest PostgreSQL image.
2. Build the Flask backend image using the Dockerfile.
3. Mount the SQL initialization script to create tables automatically.
4. Expose ports 5000 (Flask) and 5432 (PostgreSQL).

You can verify the services are running:

```
docker ps
```

Listing 7.11: Checking container status

Logs can be viewed using:

```
docker compose logs -f
```

Database Persistence

The named Docker volume `pgdata` preserves database data between container restarts. You can inspect volumes using:

```
docker volume ls
```

Stopping and Cleaning Up

To stop the system gracefully:

```
docker compose down
```

To remove containers, networks, and volumes:

```
docker-compose down -v
```

Accessing the Running Application

Once the containers are up, the web interface will be available at:

http://localhost:5000

You can log in using the credentials created during initialization (from SQL inserts in `database_sql/database.sql`), or register as a new user.

Containerized Workflow Summary

The following table summarizes the deployment structure:

Table 7.1: Dockerized System Overview

Service	**Container**	**Function**
Flask Backend	`flask_app`	Runs the Flask web application on port 5000 using Gunicorn.
PostgreSQL DB	`project_db`	Hosts the relational database, initialized automatically via SQL dump.
Docker Volume	`pgdata`	Persistent storage for database data.
Network Bridge	`project_management_default`	Enables communication between containers.

This containerized setup ensures consistent behavior across environments and simplifies deployment to production or cloud servers.

8. Conclusion

This dissertation presented the design and implementation of a containerized, role-based **Project Management System** developed using the Flask web framework and deployed through Docker. The system was designed to demonstrate how lightweight web technologies and containerization can be combined to build modular, scalable, and maintainable software solutions for collaborative team environments.

8.1 Summary of Achievements

In essence, the system demonstrates a practical application of cloud-native principles — containerization, modularization, and portability — in the context of collaborative project management.

8.2 Reflection and Insights

Developing the system provided valuable insights into the challenges of integrating multiple technologies within a unified architecture. While Flask proved to be an excellent lightweight framework for rapid development, careful planning was required to manage dependencies and ensure smooth communication between backend logic, the relational database, and the user interface.

Docker's role in the project was particularly significant. By encapsulating both the Flask application and the PostgreSQL service within containers, it eliminated environment inconsistencies and simplified deployment. This containerized approach mirrors real-world DevOps practices and prepares the system for scalability on cloud platforms such as AWS, Azure, or Kubernetes clusters.

8.3 Limitations

Despite achieving its core objectives, the system has certain limitations that could be addressed in future work:

- The current notification mechanism is limited to in-app updates; it could be expanded to include real-time notifications or email alerts using tools like Flask-SocketIO or Celery.
- User interface design, though functional and responsive, could be improved with advanced frontend frameworks such as React or Vue.js for richer interactivity.
- The system's current authentication flow is based on traditional session cookies; future versions could adopt JWT (JSON Web Tokens) for stateless, API-oriented communication.
- Performance testing and load balancing across multiple containers remain unexplored areas that could enhance system scalability.

Although the developed Project Management System (PMS) successfully demonstrates the core functionalities of a modular, web-based management platform, several limitations and constraints were identified during the development and testing process. These limitations provide valuable insight into areas that could be refined or extended in future iterations of the system.

Security Constraints

The current implementation does not employ secure password hashing or encryption mechanisms. User credentials are stored in plain text for simplicity, which makes the system unsuitable for production environments. Additionally, there is no support for HTTPS communication, CSRF protection, or session expiration policies. In a real-world deployment, these mechanisms would be essential to prevent unauthorized access, data breaches, and session hijacking.

Testing and Reliability

Functional testing was primarily conducted manually, focusing on verifying individual CRUD operations and ensuring that templates rendered correctly. There is no automated testing framework such as `pytest` or `unittest` integrated into the project. Without continuous testing, regression issues may arise when extending functionality.

Future work should introduce unit tests, integration tests, and possibly continuous integration (CI) pipelines to ensure stability and maintainability.

Scalability and Performance

The PMS is currently optimized for small-scale use and has been evaluated only in a local Dockerized environment. Performance benchmarks under concurrent user loads were not conducted. The architecture, while modular, may require adjustments (such as asynchronous request handling, caching, or message queuing) to scale efficiently in high-traffic conditions. Database query optimization and connection pooling could also improve response times for larger datasets.

Error Handling and Logging

Error handling in the current system is basic, with limited validation of user input and minimal feedback when unexpected conditions occur. Logs are not centralized, and there is no standardized logging or monitoring framework in place. Future work could incorporate structured logging, exception tracking (e.g., via `Sentry` or `Flask-Logging`), and clearer user-facing error messages.

User Interface and Usability

The front-end interface, while functional, prioritizes simplicity over design. Certain features such as drag-and-drop task management, progress dashboards, and real-time updates are not yet implemented. Accessibility and responsive design could also be improved to enhance user experience across different devices and screen sizes.

Deployment and Maintainability

The Docker-based deployment simplifies local setup but lacks orchestration support for production environments. There is no configuration for container networking, health checks, or scaling strategies (e.g., Kubernetes or Docker Compose services). Documentation for setup and maintenance is limited and should be expanded to facilitate onboarding for new developers or administrators.

Future Enhancements

The project currently focuses on functionality and modular architecture rather than completeness. Future enhancements could include:

- Implementation of secure authentication (OAuth2, JWT, or session tokens).
- Role-based dashboards and visual analytics.
- Real-time notifications and WebSocket integration.
- Versioned APIs for extensibility.
- Integration with external services (email, cloud storage, CI/CD tools).

In conclusion, while the PMS achieves its educational objectives by demonstrating modular design, database integration, and role-based task management, it remains a prototype that can evolve into a robust enterprise solution through systematic security hardening, testing, and scalability improvements.

8.4 Final Remarks

In conclusion, the development of this system demonstrates how open-source technologies can be effectively combined to build a robust and extensible project management solution. The application balances simplicity with functionality, showing that containerized Python microservices can rival heavier enterprise systems in flexibility and maintainability. Beyond its technical implementation, the project emphasizes the importance of modular design, database integrity, and reproducible deployment — key aspects of modern software engineering practice.

This work provides a solid foundation for future research and practical extensions, whether in academic, professional, or cloud-native development contexts.

A. Appendix A: API Endpoints Reference

This appendix provides an accurate overview of all API endpoints implemented in the current Flask-based project management system. Only the endpoints present in the actual codebase are included. Each endpoint is associated with its HTTP method, required access role, and a brief description.

A.1 Authentication and Session Management

Table A.1: Authentication and Session Endpoints

Endpoint	**Method**	**Access**	**Description**
`/`	GET	Public	Homepage (landing page).
`/admin-signup`	GET/POST	Public	Register a new Admin account.
`/admin-login`	GET/POST	Public	Admin login page and authentication.
`/teamLeader-signup`	GET/POST	Public	Register a new Team Leader account.
`/teamLeader-login`	GET/POST	Public	Team Leader login and authentication.
`/member-signup`	GET/POST	Public	Register a new Member account.
`/member-login`	GET/POST	Public	Member login and authentication.
`/logout`	GET	Authenticated	Logout (Admin/TeamLeader/Member).
`/member-logout`	GET	Member	Logout Member specifically.
`/teamLeader-logout`	GET	Team Leader	Logout Team Leader specifically.

A.2 Dashboard Views

Table A.2: Role-Based Dashboards

Endpoint	Method	Access	Description
`/admin-mainpage`	GET	Admin	Admin main dashboard (users, teams, overview).
`/teamLeader-mainpage`	GET	Team Leader	Team Leader dashboard and summary view.
`/member-mainpage`	GET	Member	Member dashboard showing personal tasks and teams.

A.3 Team Management Endpoints

Table A.3: Team Operations (Implemented Endpoints Only)

Endpoint	Method	Access	Description
`/admin-manageTeams`	GET/POST	Admin	Create or manage teams.
`/admin-viewTeam/<team_id>`	GET	Admin	View a specific team and its details.
`/teamLeader-viewTeam/<team_id>`	GET	Team Leader	View details of a team led by the user.
`/member-viewTeam/<team_id>`	GET	Member	View a team the member belongs to.
`/member-teamsIncluded`	GET	Member	View all teams the member is assigned to.

A.4 Task Management Endpoints

Table A.4: Task Operations (Implemented Endpoints Only)

Endpoint	Method	Access	Description
`/teamLeader-viewTask/<task_id>`	GET	Team Leader	View task details.
`/teamLeader-editTask/<task_id>`	GET/POST	Team Leader	Edit an existing task.
`/teamLeader-deleteTask/<task_id>`	POST	Team Leader	Delete a task.
`/teamLeader-manageTasksProjects`	GET	Team Leader	Overview of tasks and projects.
`/member-viewTask/<task_id>`	GET	Member	View task assigned to the member.
`/member-tasks`	GET	Member	List all tasks assigned to the member.

A.5 Comments

Table A.5: Comment Endpoints (Actual Implementation)

Endpoint	Method	Access	Description
/member-addComment/<task_id>	POST	Member	Add a comment to a task.
/teamLeader-addComment/<task_id>	POST	Team Leader	Add a comment to a task.

Note: The system currently does **not** implement comment deletion or file attachments despite these features being described in the initial design.

A.6 Administrative Operations

Table A.6: Admin Operations (Implemented Endpoints Only)

Endpoint	Method	Access	Description
/admin-manageUsers	GET	Admin	View and manage registered users.
/activate_user/<username>	POST	Admin	Activate a user account.
/deactivate_user/<username>	POST	Admin	Deactivate a user account.
/admin-show_tasks_and_projects	GET	Admin	Overview of tasks and projects in the system.

A.7 Response Status Codes

Table A.7: Response Codes Used in the System

Code	Type	Meaning
200 OK	HTML	Successful request; page rendered.
302 Found	Redirect	Used after successful login or form submission.
400 Bad Request	HTML	Invalid or incomplete form input.
403 Forbidden	HTML	User lacks the required role/permission.
404 Not Found	HTML	Requested resource does not exist.
500 Internal Server Error	HTML	Application error (logged server-side).

This appendix accurately reflects all API endpoints implemented in the system.

B. Appendix B: Selected Source Code Listings

This appendix presents representative source code excerpts from the implementation of the Project Management System (PMS). These examples include backend routes, database schema definitions, Docker configurations, frontend templates, and interface assets, providing an integrated overview of how the system's components interact.

Table B.1: Mapping of Appendix B Listings to Actual Source Files

Listing Title	**Actual File Path**
Homepage route	`routes/homepage.py`
Administrator signup route	`routes/admin_authenticate.py`
Team Leader dashboard route	`routes/teamLeader_mainpage.py`
Admin dashboard logic	`routes/admin_mainpage.py`
Schema creation script	`database_sql/database.sql`
Sample data population	`database_sql/docker_init_restore.sh`
Dockerfile	`Dockerfile`
docker-compose.yml	`docker-compose.yml`
Homepage template	`templates/index.html`
Team management dashboard template	`templates/teamLeader_manageTeams.html`
Team Leader styling	`static/css/teamLeader.css`
Admin interactivity script	`static/script/script.js`
Local startup script	`source_run_flask.sh`
Git push automation script	`push_to_github.sh`

B.1 Backend Implementation

```
from flask import Blueprint, render_template
homepage_bp = Blueprint("homepage", __name__)

@homepage_bp.route("/")
def index():
    """Renders the homepage."""
    return render_template("index.html")
```

Listing B.1: Homepage route (routes/homepage.py)

```
@admin\_authenticate\_bp.route("/admin-signup", methods=["GET", "POST"])
def admin_signup():
    """Handles admin registration."""
    if request.method == "POST":
        username = request.form.get("username")
        email = request.form.get("email")
        password = request.form.get("password")

        conn = get_db_connection()
        cur = conn.cursor()
        cur.execute("""
            INSERT INTO users (username, email, password, role, status)
            VALUES (%s, %s, %s, 'ADMIN', 'INACTIVE');
        """, (username, email, password))
        conn.commit()
        flash("Admin account created successfully!", "success")
        return redirect(url_for("admin_authenticate_bp.admin_login"))
```

Listing B.2: Administrator signup route (routes/admin_authenticate.py)

```
@teamLeader_mainpage\_bp.route("/teamLeader-mainpage")
def teamLeader_mainpage():
    if "teamLeader_email" not in session:
        flash("Please log in first.", "error")
        return redirect("/teamLeader-login")

    leader_email = session["teamLeader_email"]
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("SELECT username, email FROM users WHERE email = %s;", (
    ↪ leader_email,))
    leader = cur.fetchone()
    return render_template("teamLeader_mainpage.html", leader=leader)
```

Listing B.3: Team Leader dashboard route (routes/teamLeader_mainpage.py)

```
@admin_mainpage\_bp.route("/admin-mainpage")
def admin_mainpage():
    if "admin_email" not in session:
        flash("Please log in as Admin.", "error")
```

```
        return redirect("/admin-login")

    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("SELECT username, email, role, status FROM users ORDER BY
    ↪ user_id;")
    users = cur.fetchall()
    cur.close(); conn.close()
    return render_template("admin_mainpage.html", users=users)
```

Listing B.4: Admin dashboard logic (routes/admin_mainpage.py)

B.2 Database Definition and Initialization

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(100) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password TEXT NOT NULL,
    role VARCHAR(50) DEFAULT 'MEMBER',
    status VARCHAR(20) DEFAULT 'ACTIVE'
);
CREATE TABLE teams (
    team_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    leader_id INT REFERENCES users(user_id)
);
CREATE TABLE tasks (
    task_id SERIAL PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    description TEXT,
    status VARCHAR(20) DEFAULT 'TODO',
    team_id INT REFERENCES teams(team_id),
    assigned_to INT REFERENCES users(user_id)
);
```

Listing B.5: Schema creation script (database_sql/create_tables.sql)

```
INSERT INTO users (username, email, password, role)
VALUES
```

```
('admin', 'admin@example.com', 'admin123', 'ADMIN'),
('leader', 'leader@example.com', 'leader123', 'TEAM_LEADER'),
('member', 'member@example.com', 'member123', 'MEMBER');

INSERT INTO teams (name, description, leader_id)
VALUES ('DevOps', 'Containerization and CI/CD', 2);
```

Listing B.6: Sample data population (database_sql/insert_data.sql)

B.3 Containerization and Deployment

```
FROM python:3.12-slim
WORKDIR /app
RUN apt-get update && apt-get install -y libpq-dev gcc
COPY . .
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "backend_server_app:app"]
```

Listing B.7: Dockerfile for backend container

```
version: "3.9"

services:
  db:
    image: postgres:16
    container_name: project_db
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: xotour
      POSTGRES_DB: postgres
    ports:
      - "5433:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./database_sql/docker_init_restore.sh:/docker-entrypoint-initdb.d/
    ↪ docker_init_restore.sh
      - ./database_sql/database.sql:/docker-entrypoint-initdb.d/database.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
```

```
      retries: 5
      timeout: 5s
    restart: unless-stopped

  web:
    build: .
    container_name: project_app
    depends_on:
      db:
        condition: service_healthy
    ports:
      - "5000:5000"
    volumes:
      - .:/app
    environment:
      FLASK_APP: backend_server_app.py
      FLASK_ENV: development
      DB_HOST: db
      DB_USER: postgres
      DB_PASSWORD: xotour
      DB_NAME: postgres
    command: flask run --host=0.0.0.0 --port=5000
    restart: unless-stopped

volumes:
  pgdata:
```

Listing B.8: docker-compose.yml linking backend and database

B.4 Frontend Templates and Assets

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Welcome to App!</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css')
    ↪ }}">
</head>
<body>
  <h1>Welcome to App!</h1>
```

```
  {% with messages = get_flashed_messages(with_categories=true) %}
    {% for category, message in messages %}
      <li class="{{ category }}">{{ message }}</li>
    {% endfor %}
  {% endwith %}
</body>
</html>
```

Listing B.9: Homepage template (templates/index.html)

```
<table>
  <thead>
    <tr><th>Team Name</th><th>Description</th></tr>
  </thead>
  <tbody>
    {% for team in teams %}
      <tr><td>{{ team.name }}</td><td>{{ team.description or '-' }}</td></tr>
    {% endfor %}
  </tbody>
</table>
```

Listing B.10: Team management dashboard (templates/teamLeader_manageTeams.html)

```
table
  border-collapse: collapse;
  width: 80%;
  margin: 20px auto;

th, td
  border: 1px solid #ccc;
  padding: 10px;

th  background-color: #0d6efd; color: white;
trnth-child(even) { background-color: #f2f2f2 }
```

Listing B.11: Team Leader page styling (static/css/teamLeader.css)

```
function activateUser(username) {
  fetch(`/activate_user/${username}`, { method: "POST" })
    .then(res => res.json())
    .then(data => alert(data.message || "User activated"))
    .catch(err => alert(`${err}`));
}
```

Listing B.12: Admin interactivity (static/script/script.js)

B.5 Utility and Automation Scripts

```
#!/bin/bash
source venv/bin/activate
export FLASK_APP=backend_server_app.py
flask run --host=0.0.0.0 --port=5000
```

Listing B.13: Local startup script (source_run_flask.sh)

```
#!/bin/bash
git add .
git commit -m "Auto update from local dev"
git push origin main
```

Listing B.14: Git push automation script (push_to_github.sh)

C. Appendix C: User Interaction Flow

Introduction

This appendix provides a detailed walkthrough of the interaction flow between the system's primary user roles, namely the **Team Member** and the **Team Leader**. It outlines how users authenticate, access and manage tasks, exchange comments, and optionally upload attachments during their collaboration process. The described sequence demonstrates how a Member submits progress updates or feedback on a task, how the Team Leader reviews this information, and how new tasks or updates are created in response. This overview complements the technical implementation described in earlier chapters by illustrating the practical use of the system's core functionalities.

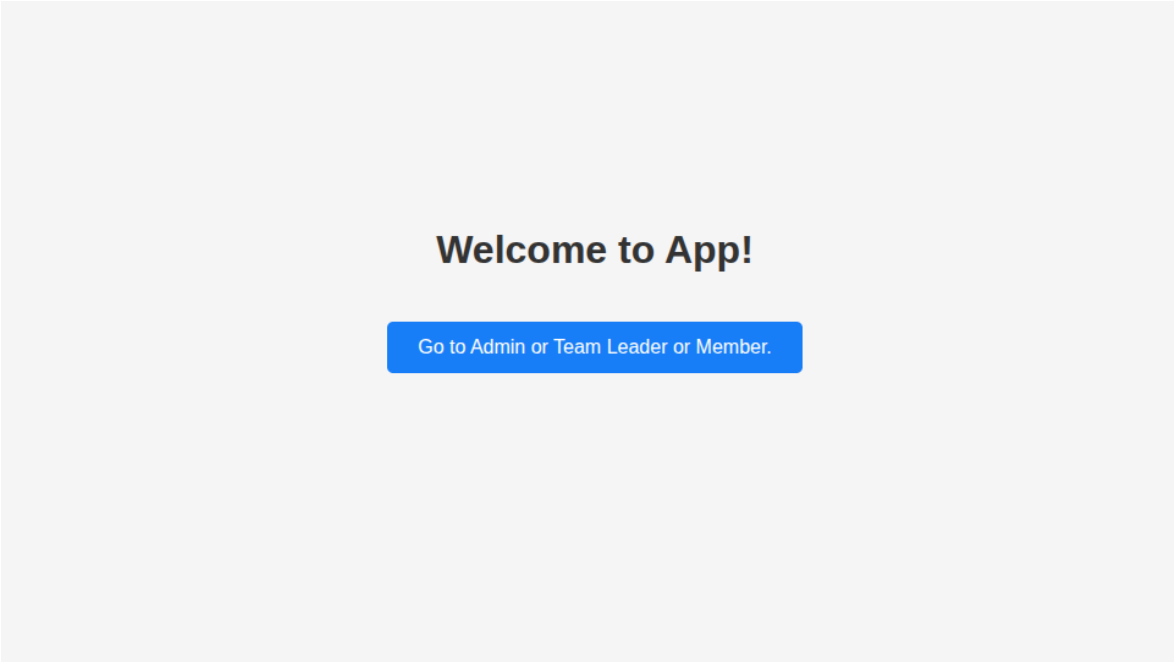


Figure C.1: Landing page welcoming the user and providing navigation to the role-selection page.

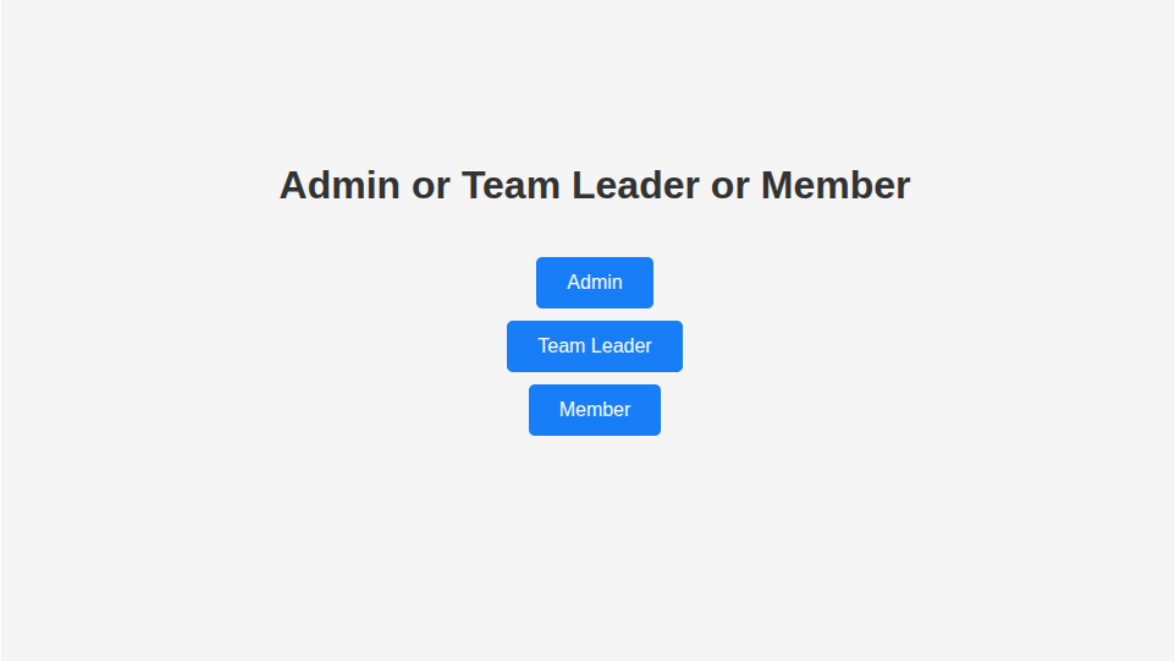


Figure C.2: ole selection screen where the user chooses to proceed as Admin, Team Leader, or Member.

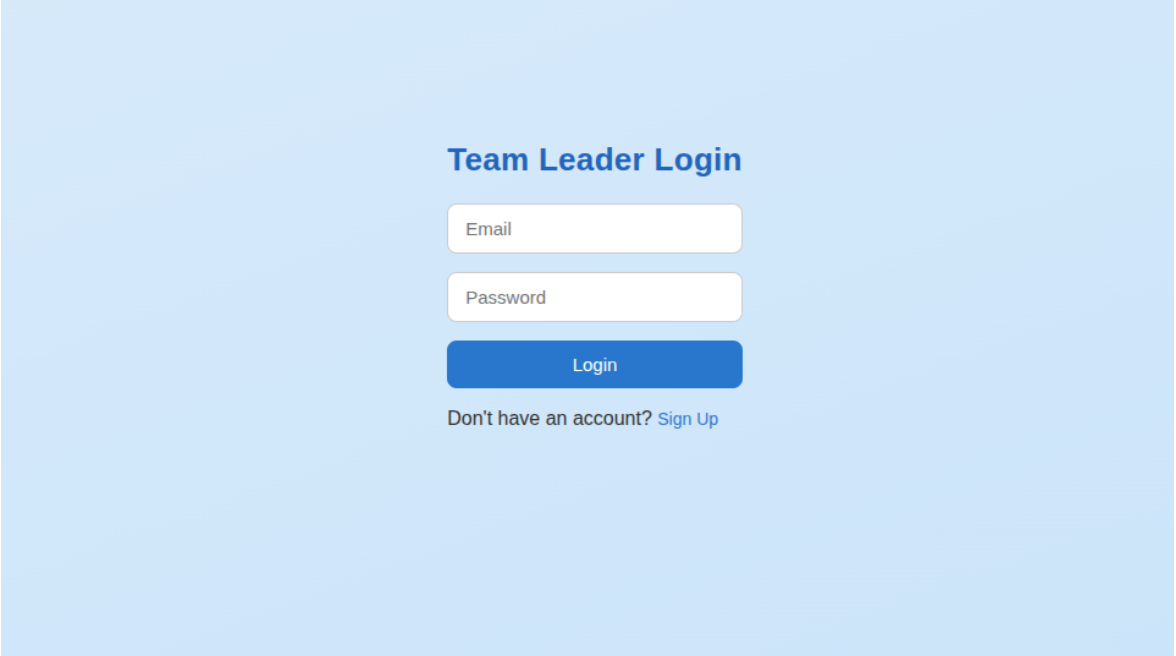


Figure C.3: Team Leader login page prompting the user to enter email and password.

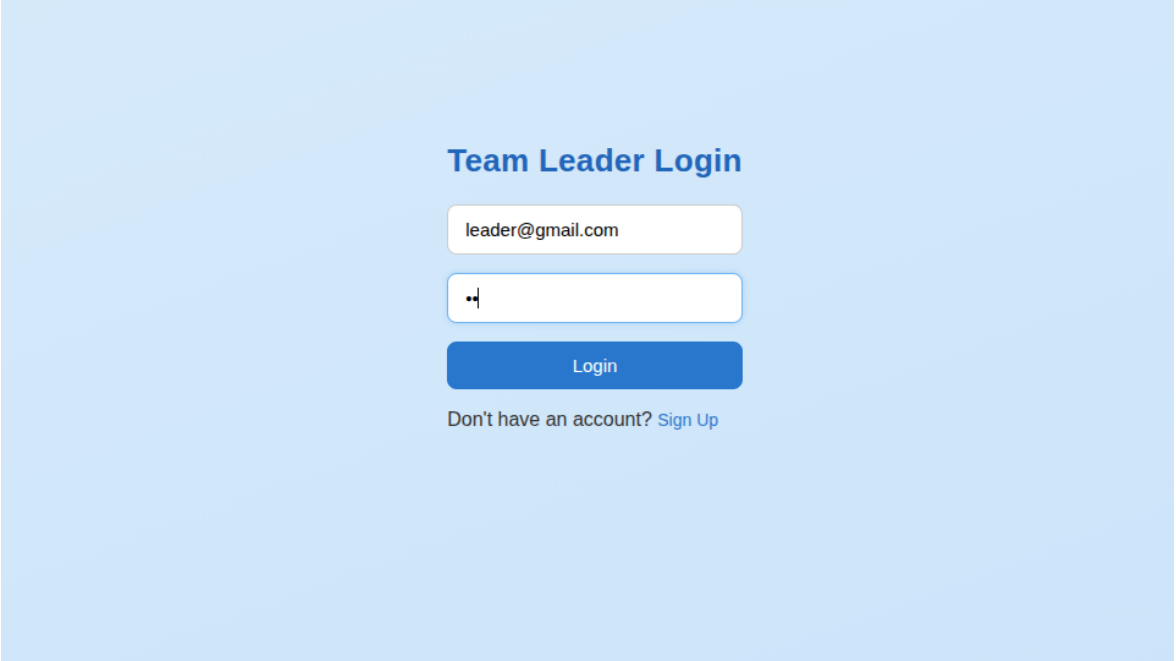


Figure C.4: Team Leader login page prompting the user to enter email and password. Password is "123".

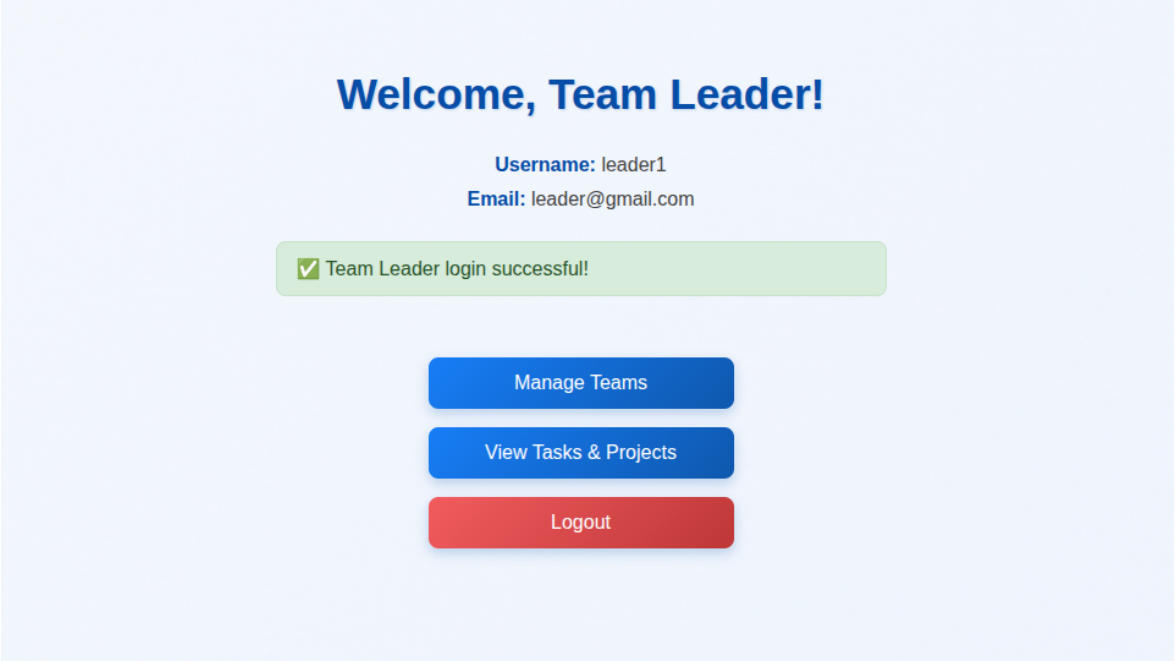


Figure C.5: Team Leader login attempt with credentials filled in.

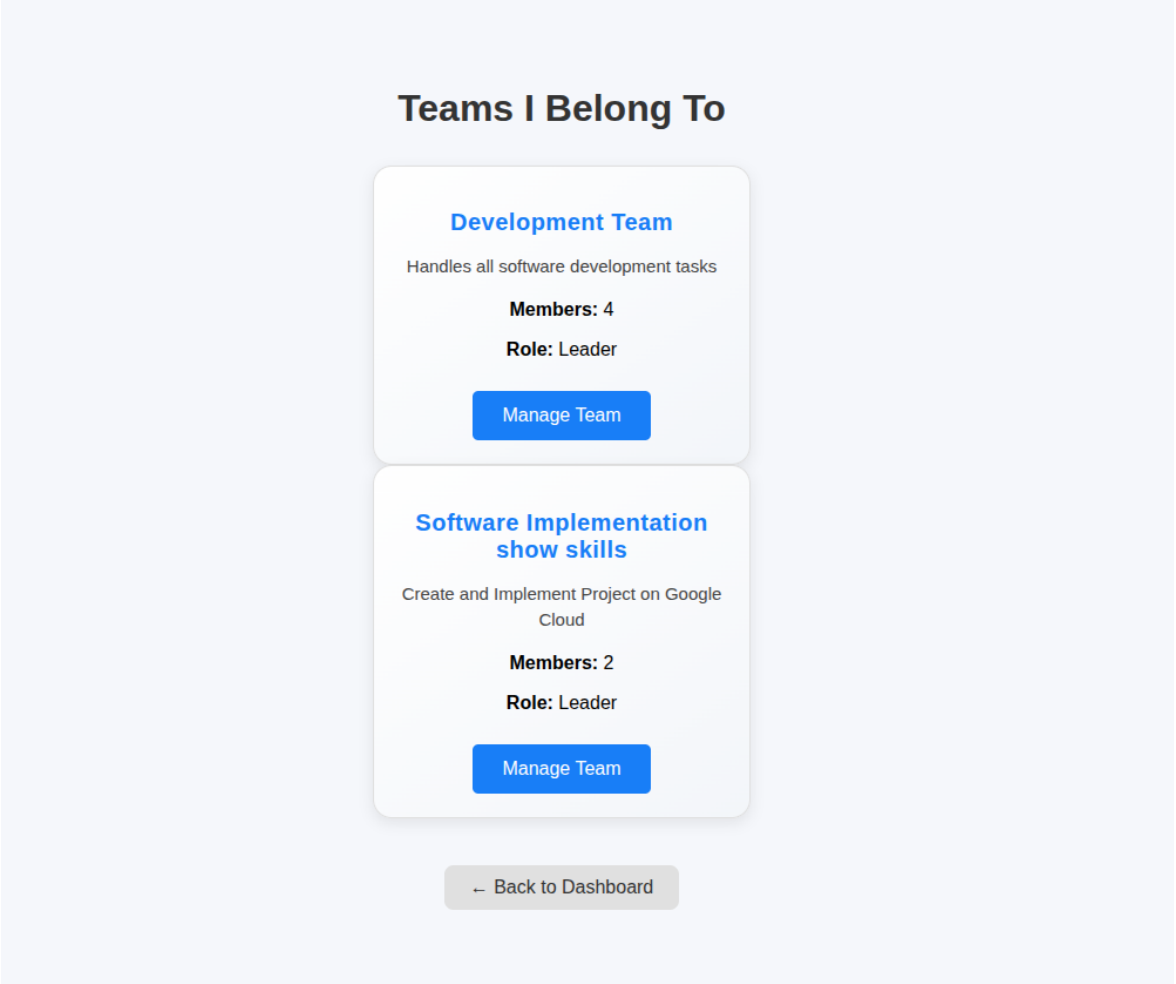


Figure C.6: eam Leader dashboard displaying account information and navigation options.

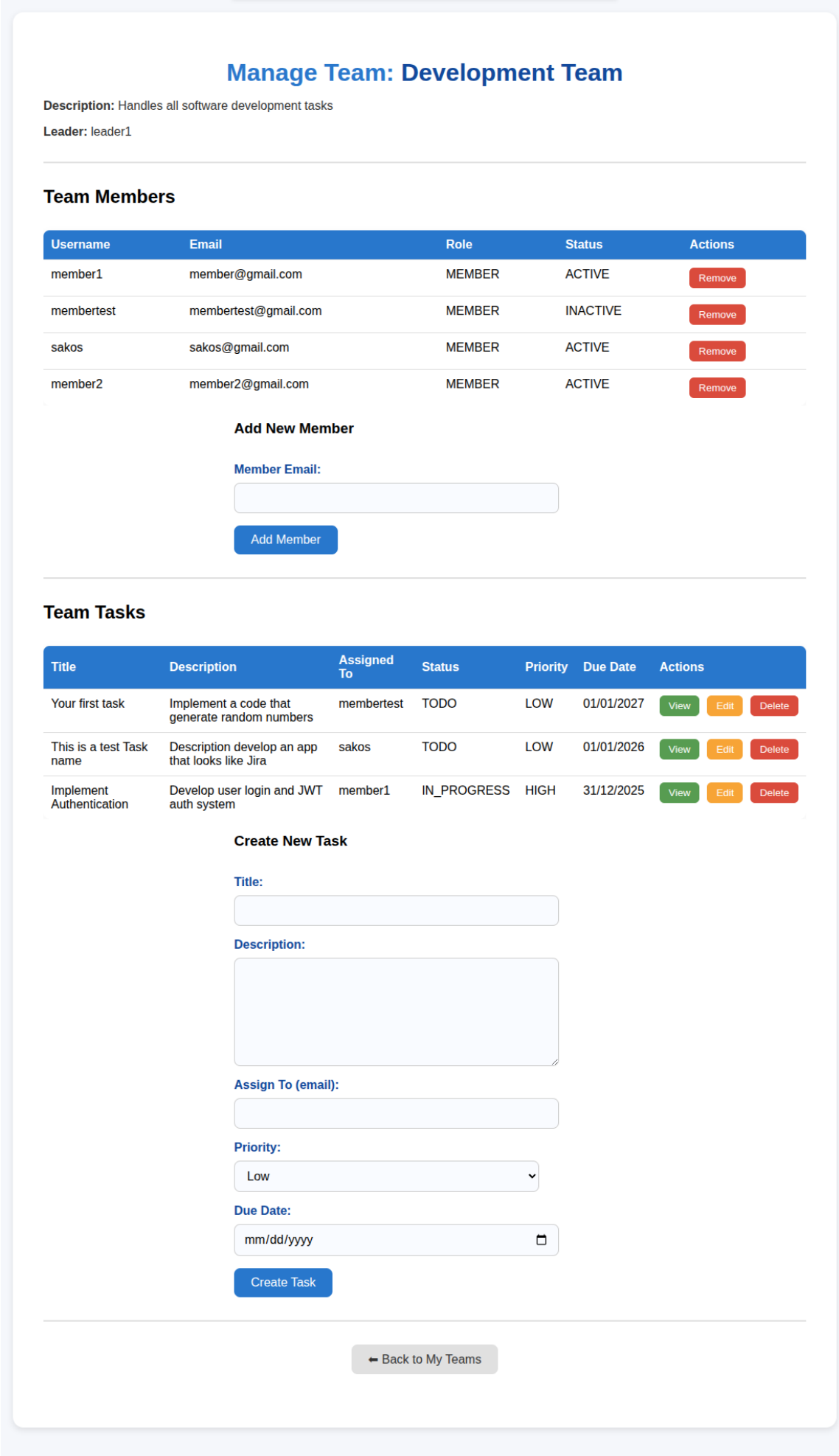
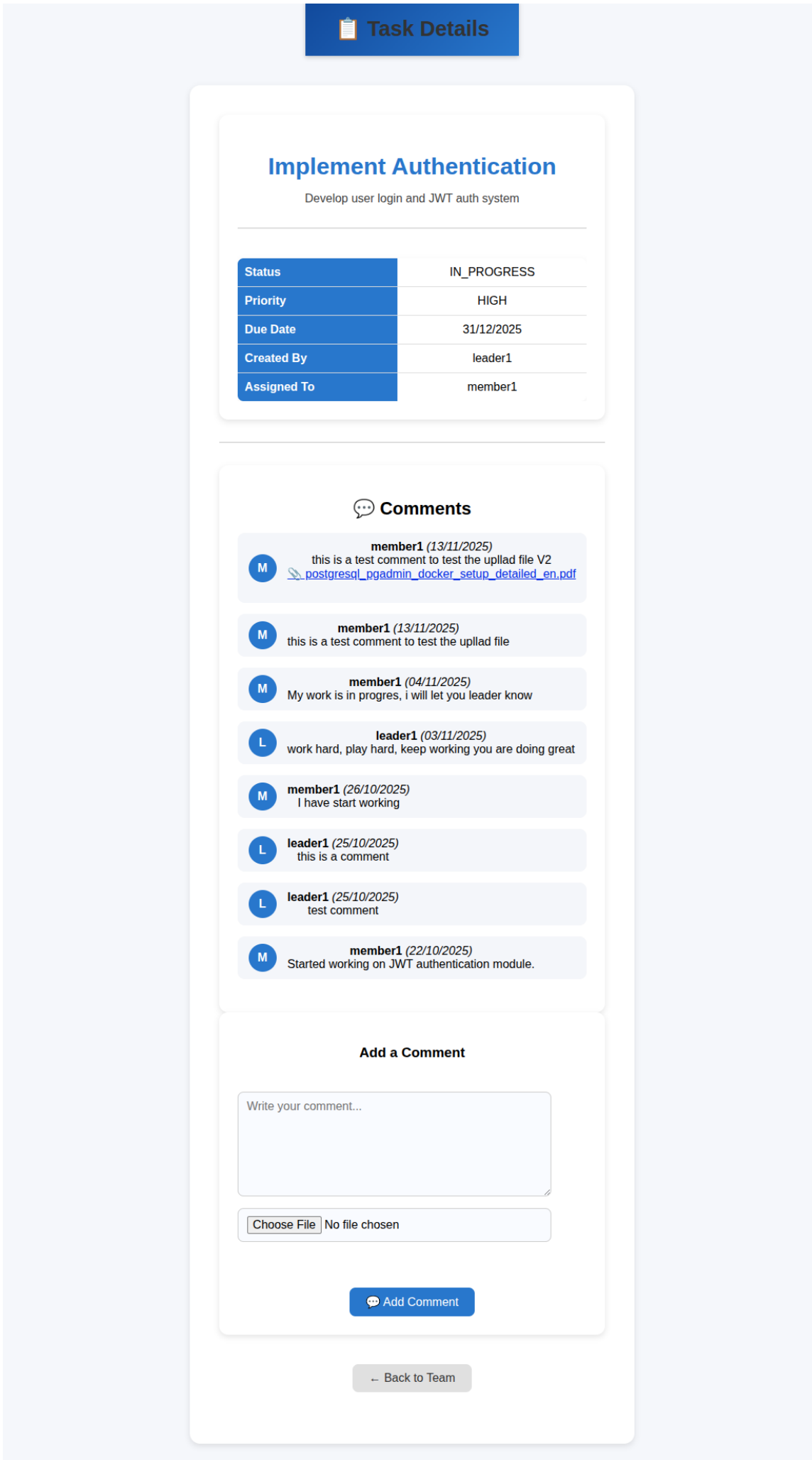


Figure C.7: Teams overview page showing all teams the Team Leader belongs to, with


Task Details
Implement Authentication
Develop user login and JWT auth system
Status
IN_PROGRESS
Priority
HIGH
Due Date
31/12/2025
Created By
leader1
Assigned To
member1
Comments
M
member1 (13/11/2025)
this is a test comment to test the upllad file V2
postgresql_pgadmin_docker_setup_detailed_en.pdf
M
member1 (13/11/2025)
this is a test comment to test the upllad file
M
member1 (04/11/2025)
My work is in progres, i will let you leader know
L
leader1 (03/11/2025)
work hard, play hard, keep working you are doing great
M
member1 (26/10/2025)
I have start working
L
leader1 (25/10/2025)
this is a comment
L
leader1 (25/10/2025)
test comment
M
member1 (22/10/2025)
Started working on JWT authentication module.
Add a Comment
Write your comment...
Choose File
No file chosen
Add Comment
← Back to Team

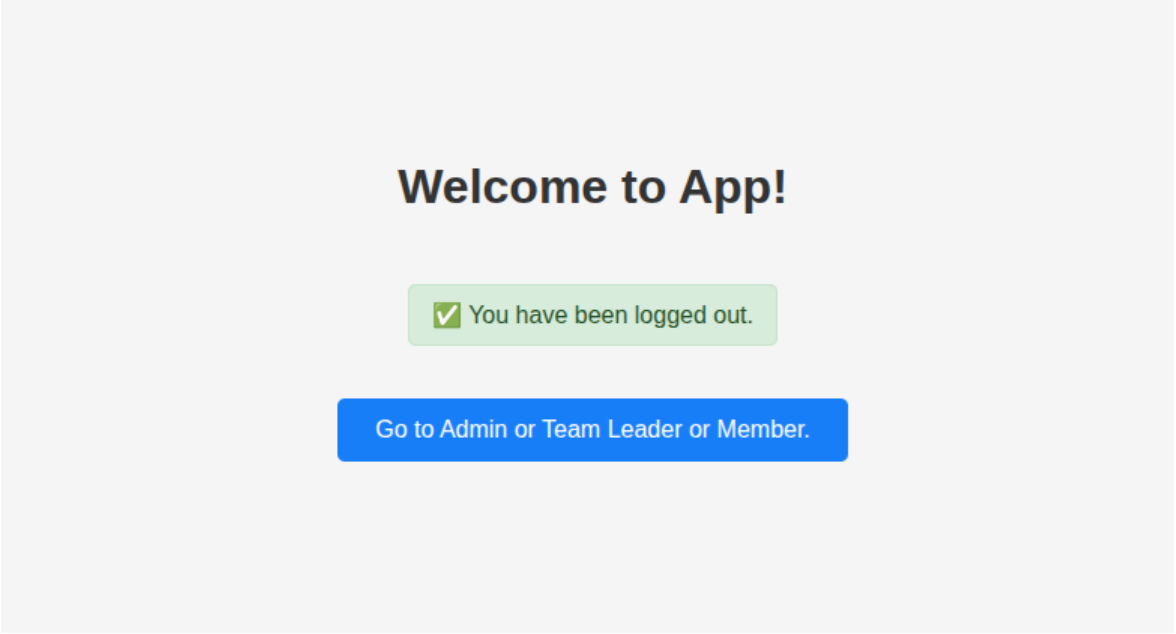


Figure C.9: Landing page displayed after a successful logout, providing navigation back to role selection.

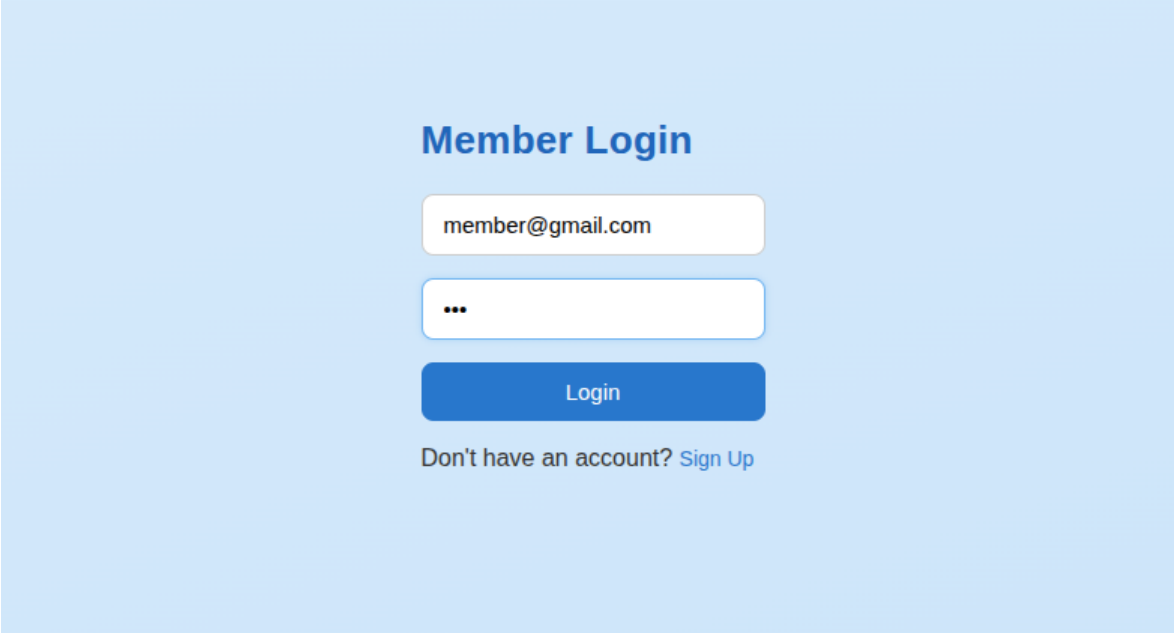


Figure C.10: Member login page with credentials filled in for authentication. Password is "123"

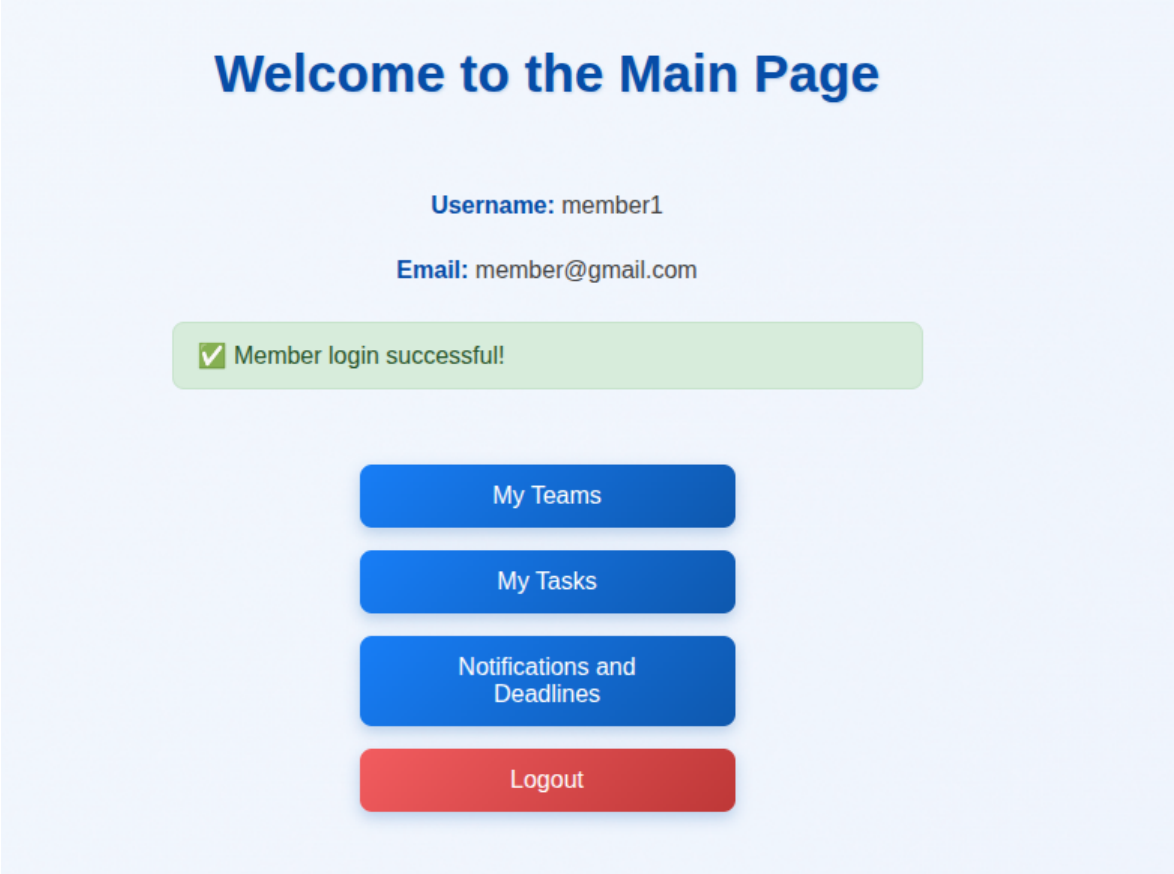


Figure C.11: Member dashboard showing account information and navigation to teams, tasks, notifications, and logout.

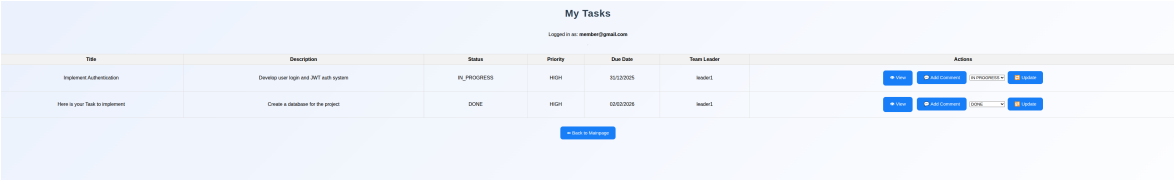


Figure C.12: My Tasks" page listing all tasks assigned to the logged-in member, with options to view, comment, and update status.

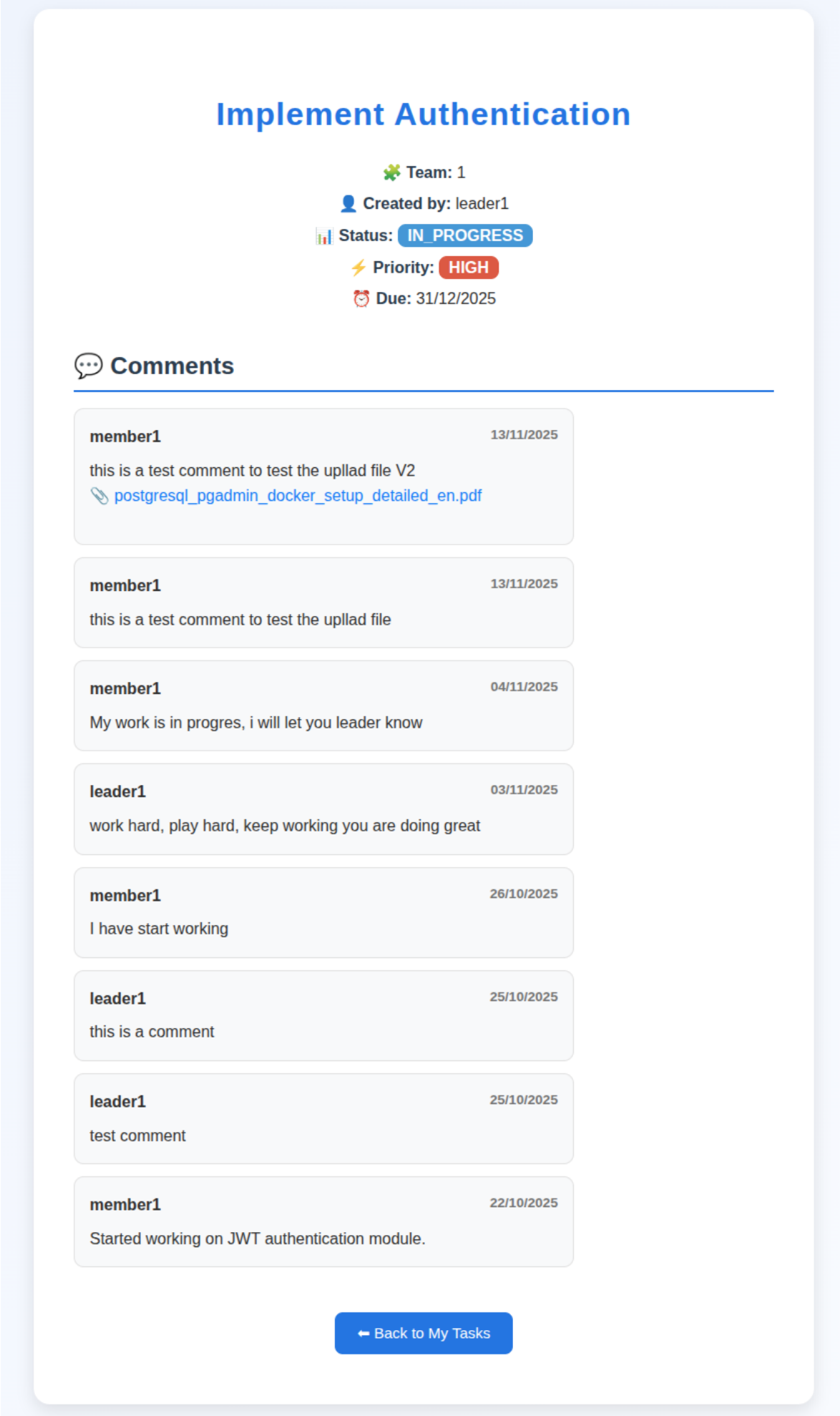


Figure C.13: Task details page for a member, including status, priority, due date,