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

Stamatios Mavitzis
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 Objectives

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.

1.2 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`.

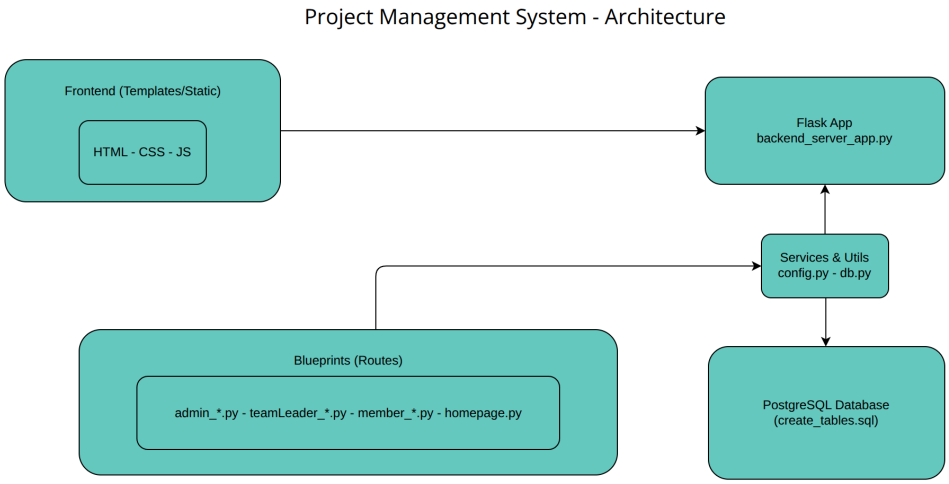

Figure 2.1: High-level architecture of the system showing main layers and interactions.

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.
- **config.py** – Centralized configuration file containing environment variables, database URIs, and API keys.
- **db.py** – Defines database initialization, connection management, and ORM logic if used.

- **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.

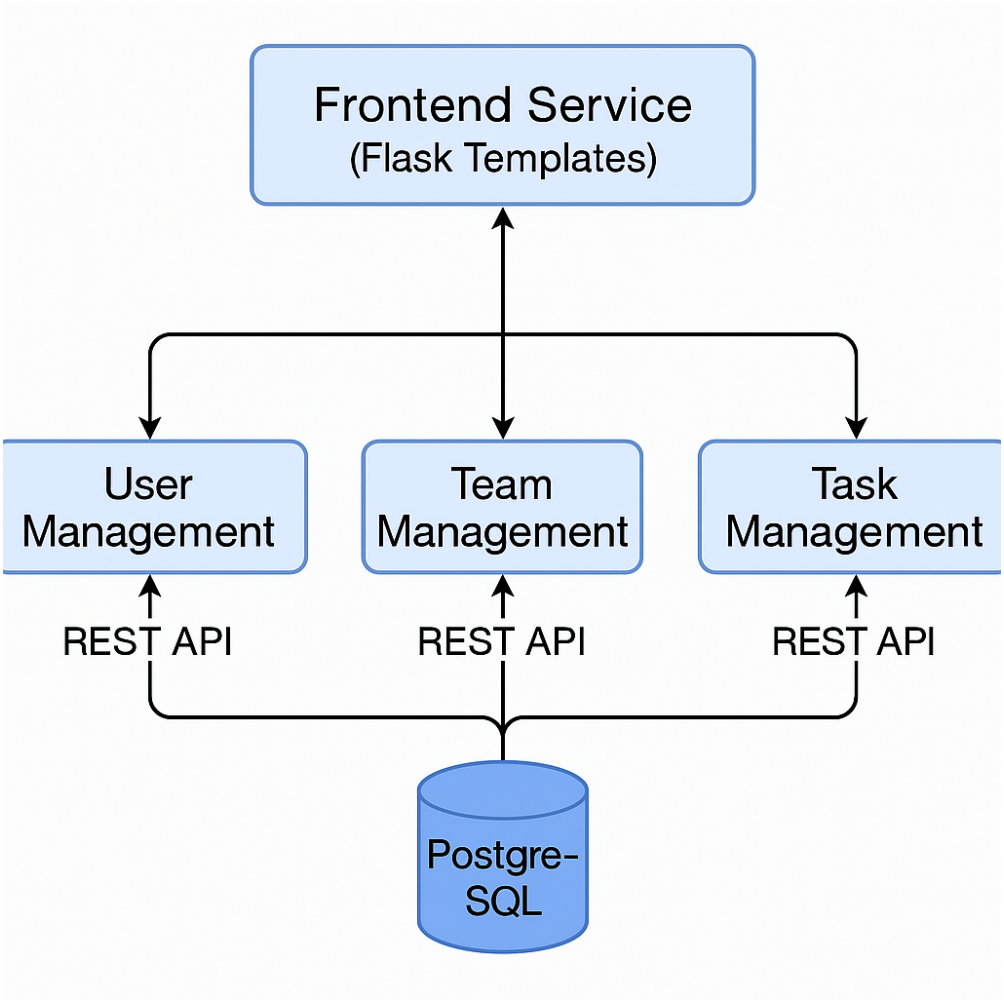

Figure 2.2: Container-based deployment structure of the system using Docker and Docker Compose.

3. Database Design

3.1 Database Overview

The database serves as the persistent data storage layer for the system, responsible for managing all core entities, including users, teams, tasks, and their relationships. It is implemented as a **relational database** that ensures data integrity, consistency, and efficient retrieval. The schema was designed following principles of **third normal form (3NF)** to eliminate redundancy and enforce logical dependencies.

All SQL schema creation and initialization scripts are stored under the directory `database_sql/`, which contains:

- SQL scripts for table creation (`create_tables.sql`)
- Sample data insertion scripts (`insert_data.sql`)
- Constraint and index definitions (`constraints.sql`)

This tight integration allows the backend logic to manipulate persistent data while ensuring transactional integrity across all operations such as task creation, team assignment, and user role updates.

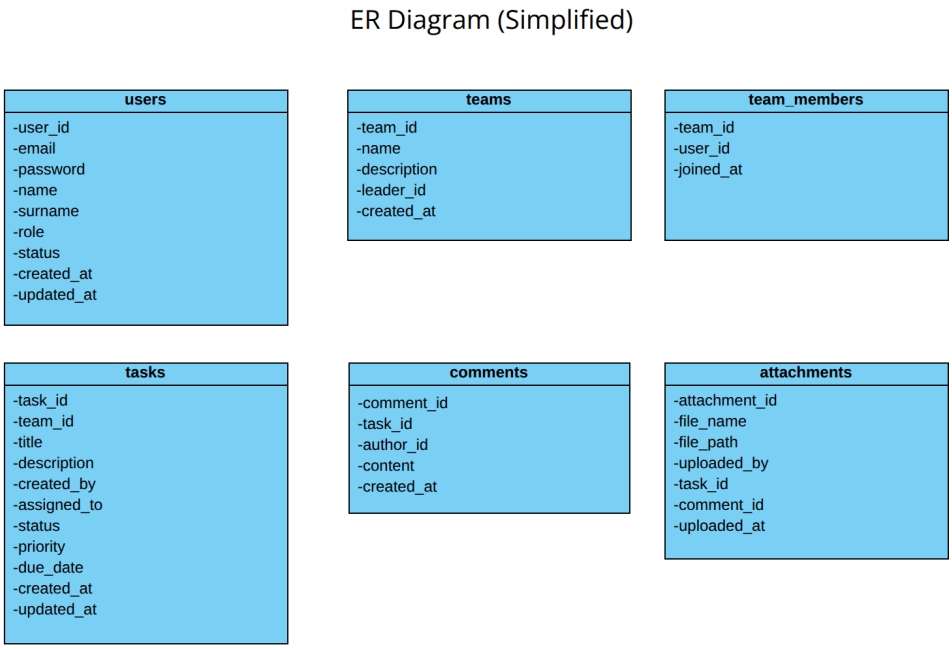

Figure 3.1: Simplified Entity–Relationship Diagram showing the main entities and their associations.

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.

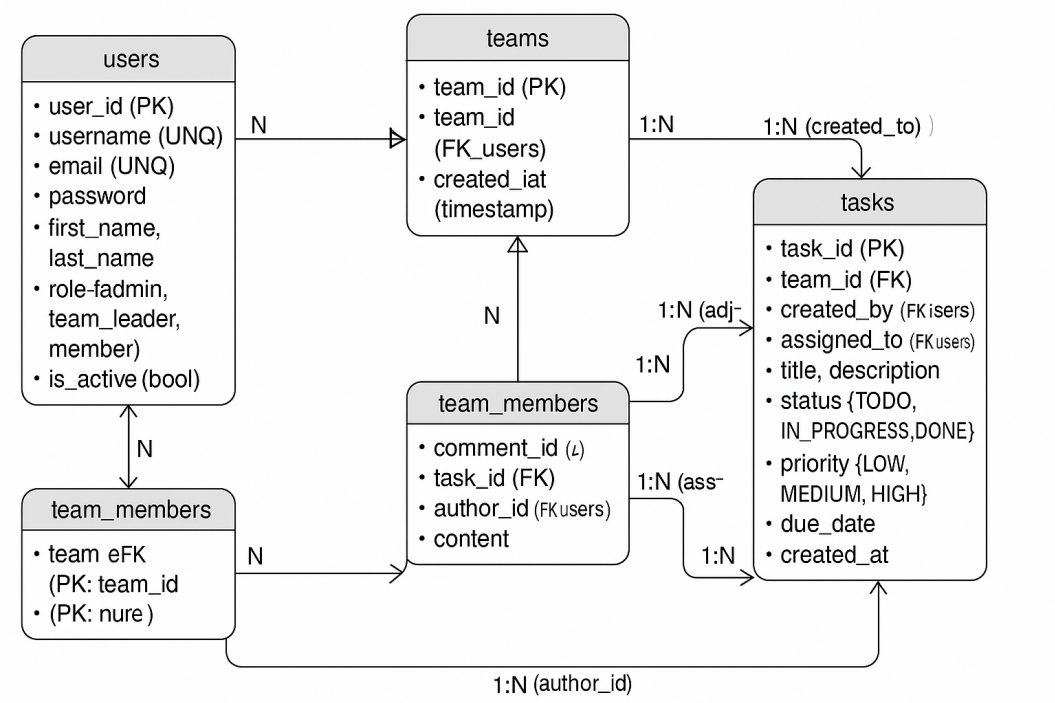

Figure 3.2: Complete ER diagram of the system including all entities, relationships, and cardinalities.

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 these entities are:

- A **User** can belong to multiple **Teams**.
- A **Team** can manage multiple **Tasks**.
- A **Task** can be assigned to multiple **Users** (many-to-many relationship).
- A **Project** can contain multiple **Tasks** and **Teams**.
- Each **Attachment** belongs to a single **Task**.

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.

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` and `UNIQUE`.

Table 3.1: Teams Table

Column	Type	Constraints	Description
team_id	INT	PK, AUTO_INCREMENT	Unique identifier for each team.
name	VARCHAR(100)	NOT NULL	Team name.
description	TEXT	NULL	Optional description of the team.
leader_id	INT	FK → users(user_id)	User who leads the team.

Table 3.2: Tasks Table

Column	Type	Constraints	Description
task_id	INT	PK, AUTO_INCREMENT	Unique identifier for each task.
title	VARCHAR(100)	NOT NULL	Task title or short label.
description	TEXT	NULL	Detailed information about the task.
status	VARCHAR(30)	NOT NULL	Task state (Pending, Active, Done).
deadline	DATE	NULL	Optional due date.
priority	VARCHAR(20)	DEFAULT 'Normal'	Task urgency level.
team_id	INT	FK → teams(team_id)	Team responsible for this task.

Table 3.3: Projects Table

Column	Type	Constraints	Description
project_id	INT	PK, AUTO_INCREMENT	Unique identifier for each project.
name	VARCHAR(100)	NOT NULL	Project title.
description	TEXT	NULL	Brief overview of the project.
owner_id	INT	FK → users(user_id)	Project owner or manager.
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Date of project creation.

Table 3.4: Roles Table

Column	Type	Constraints	Description
role_id	INT	PK, AUTO_INCREMENT	Unique identifier for each role.
role_name	VARCHAR(50)	UNIQUE, NOT NULL	Role name (Admin, Manager, Member).
permissions	TEXT	NULL	List or JSON of allowed actions.

Table 3.5: Attachments Table

Column	Type	Constraints	Description
attachment_id	INT	PK, AUTO_INCREMENT	Unique identifier for each attachment.
task_id	INT	FK → tasks(task_id)	Associated task.
file_path	VARCHAR(255)	NOT NULL	File location in the system.
uploaded_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Time when the file was uploaded.

Table 3.6: User_Task (Mapping Table)

Column	Type	Constraints	Description
user_id	INT	FK → users(user_id)	Associated user.
task_id	INT	FK → tasks(task_id)	Associated task.

Each table is designed for data normalization and referential integrity, with foreign keys ensuring valid references across related entities. Indexes are defined on commonly queried columns such as `username`, `team_id`, and `task_id` to optimize join operations and query performance.

3.3.1 Database Normalization and Integrity Constraints

The database schema was designed following strict normalization principles to ensure data consistency, minimize redundancy, and maintain logical clarity. Normalization helps separate independent concepts into distinct tables while preserving valid relationships among them.

First Normal Form (1NF)

All tables comply with the first normal form:

- Each attribute contains only atomic (indivisible) values.
- No repeating groups or arrays exist within a single column.
- Each record is uniquely identified by a primary key (e.g., `user_id`, `task_id`).

Second Normal Form (2NF)

The schema satisfies the second normal form by ensuring that:

- Every non-key attribute depends entirely on the table's primary key.
- Partial dependencies (where an attribute depends on only part of a composite key) are eliminated.
- For example, in the `user_task` junction table, both `user_id` and `task_id` form a composite primary key.

Third Normal Form (3NF)

All tables meet third normal form by ensuring:

- Non-key attributes are dependent only on the primary key.
- There are no transitive dependencies (attributes depending on non-key attributes).

- Referential dependencies (e.g., role of a user, team of a task) are expressed through foreign key relationships.

Referential Integrity and Constraints

To ensure data accuracy and consistency across related tables, the following integrity constraints are applied:

- **Primary Keys (PK):** Each entity table defines a unique primary key, such as `user_id` in `users` or `task_id` in `tasks`, ensuring each record can be distinctly identified.
- **Foreign Keys (FK):** Relationships are maintained through foreign key constraints, for example:
  - `tasks.team_id` → `teams.team_id`
  - `users.role_id` → `roles.role_id`
  - `attachments.task_id` → `tasks.task_id`
- **Unique Constraints:** Columns like `username` and `email` in the `users` table are constrained to prevent duplicates.
- **Default Values:** Timestamps such as `created_at` and `uploaded_at` use default values to automatically record creation times.
- **Cascade Operations:**
  - `ON DELETE CASCADE` ensures that deleting a parent record (e.g., a team) automatically removes associated child records (e.g., its tasks).
  - `ON UPDATE CASCADE` propagates key changes safely across related tables.

Indexes and Optimization

To enhance query performance, indexes are created on columns frequently used in search and join operations. Examples include:

- `users.email` and `users.username` – for fast user lookups.
- `tasks.status` and `tasks.deadline` – for efficient filtering and task scheduling queries.

- `user_task(user_id, task_id)` – for optimized many-to-many mapping access.

This normalization and indexing strategy ensures that the system remains scalable as data volume grows, while maintaining both read performance and referential consistency across all entities.

4. Roles and Functionality

4.1 User Roles

The main roles implemented in the system are outlined below:

- **Administrator (Admin):**
  - Has full access to all system components and data.
  - Can create, modify, or delete any user, team, or project.
  - Manages global settings, including database maintenance, backups, and configuration parameters in `config.py`.
  - Can assign or revoke roles from other users.
- **Project Manager:**
  - Responsible for managing specific projects and their corresponding teams.
  - Can create and assign tasks to team members.
  - Can monitor task progress, update statuses, and modify project details.
  - Has permission to add or remove users from teams under their supervision.
- **Team Member:**
  - Can view and update tasks assigned to them.
  - Can upload attachments, comment on tasks, and mark progress.
  - Cannot delete tasks or modify other users' assignments.
  - Can collaborate within their assigned team and communicate updates to the project manager.
- **Guest or Viewer:**
  - Has read-only access to selected public information such as project overviews or reports.

- Cannot modify, create, or delete any record in the database.
- Intended for external users or stakeholders who need visibility but not editing privileges.

Role assignments are stored in the `roles` table of the database and linked to users through a foreign key (`role_id`). This linkage is dynamically enforced by the backend logic in the `routes/` module, which checks user permissions before executing restricted actions.

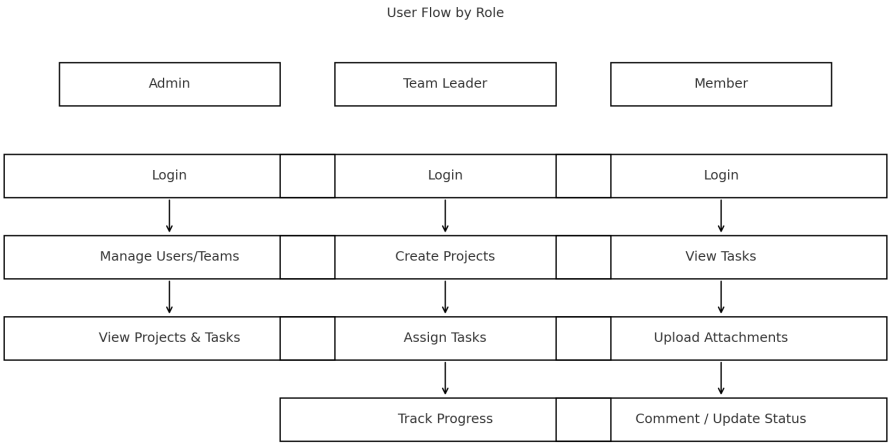

Figure 4.1: User flow diagram showing role-based access and permitted operations.

Each role corresponds to a specific set of API permissions:

- **Admin:** Full CRUD access on all endpoints.
- **Project Manager:** CRUD operations on teams, projects, and tasks within their scope.
- **Team Member:** Update and read permissions for assigned tasks only.
- **Guest:** Read-only access to public endpoints.

4.2 Core Functionalities

The system offers a wide range of core functionalities, distributed across different modules of the application. These functionalities are implemented in the `routes/`, `utils/`,

and `templates/` directories and supported by the Flask backend defined in `backend_server_app.py`.

2. Team Management

- **Team Creation:** Project managers or administrators can create new teams, assign members, and designate team leaders.
- **Member Management:** Supports adding/removing users dynamically, reflected instantly in the `teams` table.
- **Role-Based Access:** Only authorized users can modify team membership or view team details.

3. Task Management

- **Task Creation and Assignment:** Tasks can be created by project managers and assigned to one or more team members.
- **Task Tracking:** Users can update the status of their assigned tasks (Pending, In Progress, Completed).
- **Attachments and Comments:** Team members can upload supporting documents or images, stored in the `static/uploads/` directory and linked through the `attachments` table.
- **Priority and Deadlines:** Each task can have deadlines and priority levels defined to help with scheduling and workload balancing.

4. Project Management

- **Project Definition:** Projects serve as the highest-level organizational unit, grouping tasks and teams.
- **Progress Monitoring:** Managers can view progress summaries, track milestones, and adjust workloads accordingly.
- **Data Consistency:** Relationships between projects, teams, and tasks are enforced through foreign keys and backend validation.

6. Notifications and Logging

- System events (such as task updates or new team creation) can trigger internal notifications or logs.
- The `utils/` module may generate logs for debugging or activity tracking.
- Future extensions may include email or in-app notifications using Flask extensions.

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 project integrates the **Bootstrap** CSS framework to achieve a clean, responsive, and mobile-friendly design. Bootstrap components such as modals, cards, and forms accelerate frontend development and maintain UI consistency across pages.

5.3 Database Technologies

PostgreSQL / MySQL

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.

SQLAlchemy (optional ORM Layer)

In some configurations, the system may use **SQLAlchemy** as an Object Relational Mapper (ORM) to simplify database operations. This layer allows developers to interact with the database using Python classes rather than raw SQL queries, improving maintainability and abstraction.

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.

5.6 Additional Tools and Libraries

5.7 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 around Blueprints to separate concerns by role and feature. The main entry point is `backend_server_app.py`, and feature-specific logic resides under `routes/`. Representative modules include:

- `routes/homepage.py` — public homepage (index).
- `routes/admin_authenticate.py`, `routes/member_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/up options and navigation.

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

```
1 from flask import Blueprint, render_template
2
3 homepage_bp = Blueprint("homepage", __name__)
4
5 @homepage_bp.route("/")
6 def index():
7     """Public landing page."""
8     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):

```
import os

# Flask Secret Key
SECRET_KEY = os.getenv("SECRET_KEY", "supersecretkey")  # override in production!

# Allowed upload types
ALLOWED_EXTENSIONS = {"png", "jpg", "jpeg", "gif", "pdf"}

\subsection*{Database Layer and Access}
\noindent
Direct SQL access is implemented with \textbf{psycopg2}. The helper \texttt{db.py} centralizes connection handling to PostgreSQL and exposes a function to obtain a connection/cursor pair. Example pattern:
\begin{lstlisting}[language=Python, caption={Database connection pattern (db.py)}]
import psycopg2
from psycopg2.extras import RealDictCursor
from config import DB_CONFIG

def fetch_one(query, params=None):
    with get_conn() as conn:
        with conn.cursor(cursor_factory=RealDictCursor) as cur:
            cur.execute(query, params or ())
            return cur.fetchone()

def fetch_all(query, params=None):
    with get_conn() as conn:
        with conn.cursor(cursor_factory=RealDictCursor) as cur:
            cur.execute(query, params or ())
            return cur.fetchall()

def execute(query, params=None):
    with get_conn() as conn:
        with conn.cursor() as cur:
```

```
32              cur.execute(query, params or ())
33              conn.commit()
```

Listing 6.2: Configuration excerpt (config.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 6.3: 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 6.4: 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 imple-

mentation).

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.

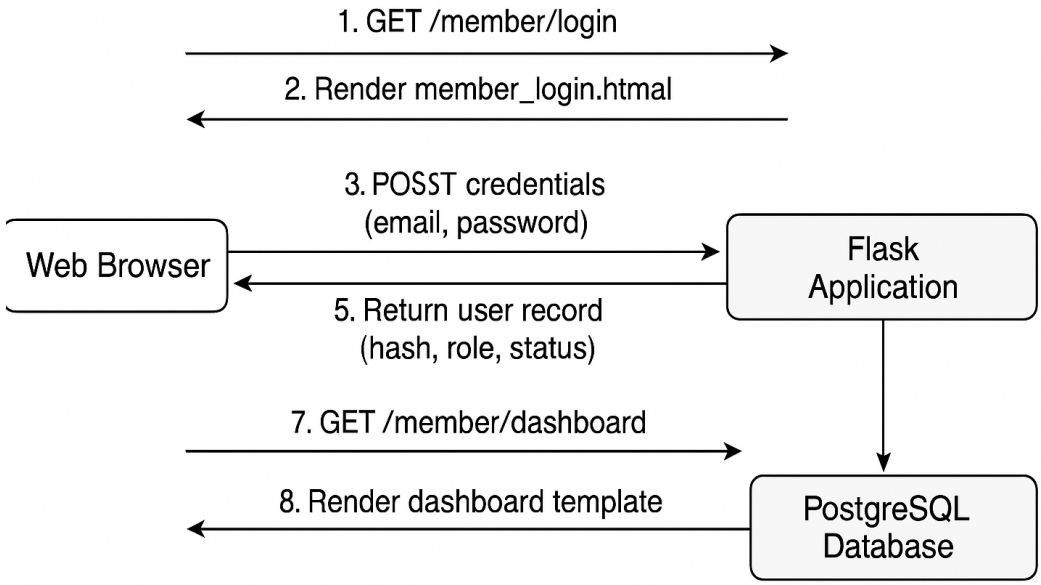

Figure 6.1: Member login sequence showing interaction between Web Browser, Flask Application, and PostgreSQL Database.

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

7.1 Local Setup

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")

\subsection*{Step 5: Initialize the Database}
\noindent
The database schema is defined in the SQL file \texttt{database\
   _sql/database.sql}.
You can load it manually into PostgreSQL:
\begin{lstlisting}[language=Bash, caption={Importing the
   database schema}]
psql -U postgres -d project_db -f database_sql/database.sql
```

Listing 7.3: config.py excerpt

Alternatively, run the included shell script:

```
#!/bin/bash
psql -U postgres -d project_db -f database_sql/database.sql
echo "Database initialized successfully."
```

Listing 7.4: 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 7.5: source_run_flask.sh

Run the application with:

```
bash 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 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:

```
# ------------------------------
# Stage 1: Base Python Environment
# ------------------------------
FROM python:3.12-slim

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y libpq-dev gcc && rm -rf
    /var/lib/apt/lists/*

# Copy project files
COPY . .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt
```

```
17
18 # Expose Flask port
19 EXPOSE 5000
20
21 # Default environment variables
22 ENV FLASK_APP=backend_server_app.py
23 ENV FLASK_RUN_HOST=0.0.0.0
24 ENV FLASK_ENV=production
25
26 # Run the app using Gunicorn for production
27 CMD ["gunicorn", "--bind", "0.0.0.0:5000", "backend_server_app:
    app"]
```

Listing 7.6: Dockerfile

Docker Compose Configuration

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

```
1 version: '3.8'
2
3 volumes:
4   pgdata:
```

Listing 7.7: docker-compose.yml

Running the Application with Docker

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

```
1 # Build images and start containers
2 docker compose up --build
3
4 # (Optional) Run in detached mode
5 docker compose up -d
```

Listing 7.8: 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 7.9: 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.

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.

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.

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.

Future Work

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. API Endpoints Reference

This appendix provides a comprehensive overview of all API endpoints implemented within the Flask-based project management system. The documentation covers both public and authenticated routes used by Administrators, Team Leaders, and Members. Unless explicitly stated otherwise, all authenticated routes require a valid session established after a successful login. Each endpoint is associated with an HTTP method, access role, and a short description of its purpose. Responses are typically rendered as HTML templates via Flask's Jinja2 engine, though JSON responses may also be produced for API extensions.

A.1 Authentication and Session Management

Table A.1: Authentication and Session Endpoints

Endpoint	**Method**	**Access**	**Description**
`/`	GET	Public	Homepage (landing page).
`/admin/login`	GET	Public	Render Admin login page.
`/admin/login`	POST	Public	Authenticate Admin and create session.
`/admin/signup`	GET	Public	Display Admin registration form.
`/admin/signup`	POST	Public	Register a new Admin user with validation.
`/teamLeader/login`	GET	Public	Render Team Leader login page.
`/teamLeader/login`	POST	Public	Authenticate Team Leader and create session.
`/member/login`	GET	Public	Render Member login page.
`/member/login`	POST	Public	Authenticate Member and create session.
`/member/signup`	GET	Public	Display Member signup form.
`/member/signup`	POST	Public	Create new Member account (default role).
`/logout`	GET	Authenticated	Terminate active session and redirect to homepage.

A.2 Dashboard Views

Table A.2: Role-Based Dashboards

Endpoint	Method	Access	Description
`/admin`	GET	Admin	Central dashboard: view teams, tasks, and users.
`/teamLeader`	GET	Team Leader	Manage own teams, projects, and assigned members.
`/member`	GET	Member	Display assigned tasks, team details, and progress.

A.3 Team Management Endpoints

Table A.3: Team Operations

Endpoint	Method	Access	Description
`/teamLeader-manageTeams`	GET	Team Leader	View and manage teams owned by the user.
`/teams/create`	POST	Admin / Team Leader	Create new team with name, description, and leader.
`/teams/{team_id}`	GET	TL/Admin/Member	Display detailed view of a specific team.
`/teams/{team_id}/edit`	POST	Admin / Team Leader	Modify existing team information.
`/teams/{team_id}/members/add`	POST	Admin / Team Leader	Add user to a team.
`/teams/{team_id}/members/remove`	POST	Admin / Team Leader	Remove user from a team.
`/teams/{team_id}/delete`	POST	Admin	Permanently delete a team and related records.

A.4 Task Management Endpoints

Table A.4: Task Operations

Endpoint	Method	Access	Description
`/tasks`	GET	Authenticated	List all visible tasks for the logged-in user.
`/teams/{team_id}/tasks`	GET	TL/Admin/Member	Retrieve tasks associated with a team.
`/teams/{team_id}/tasks/create`	POST	Admin / Team Leader	Create a new task under a team.
`/tasks/{task_id}`	GET	Authenticated	View task details if user has permission.
`/tasks/{task_id}/assign`	POST	Admin / Team Leader	Assign a task to one or more users.
`/tasks/{task_id}/status`	POST	Member / TL / Admin	Update a task's status (e.g., TODO → DONE).
`/tasks/{task_id}/edit`	POST	Admin / Team Leader	Edit metadata such as title or deadline.
`/tasks/{task_id}/delete`	POST	Admin / Team Leader	Remove a task permanently.

A.5 Comments and Attachments

Table A.5: Comments Endpoints

Endpoint	Method	Access	Description
`/tasks/{task_id}/comments`	GET	Authenticated	Display all comments for a task.
`/tasks/{task_id}/comments/add`	POST	Member / TL / Admin	Add comment to a task.
`/comments/{comment_id}/delete`	POST	TL / Admin / Author	Remove a comment if authorized.

Table A.6: Attachment Endpoints

Endpoint	Method	Access	Description
`/tasks/{task_id}/attachments`	GET	Authenticated	List all attachments related to a task.
`/tasks/{task_id}/attachments/upload`	POST	Member / TL / Admin	Upload a file attachment (validated).
`/attachments/{attachment_id}/download`	GET	Authenticated	Download attachment if permitted.
`/attachments/{attachment_id}/delete`	POST	TL / Admin / Uploader	Delete attachment.

A.6 Administrative Operations

Table A.7: Administrative and System Operations

Endpoint	Method	Access	Description
`/admin/users`	GET	Admin	View list of registered users.
`/admin/users/{user_id}/role`	POST	Admin	Modify a user's role (e.g., MEMBER → TEAM_LEADER).
`/admin/users/{user_id}/status`	POST	Admin	Activate or deactivate user account.
`/admin/projects`	GET	Admin	View or manage projects (optional feature).
`/admin/logs`	GET	Admin	Access system or application logs.

A.7 Common Parameters and Status Codes

Table A.8: Response Codes and Meanings

Code	**Type**	**Meaning**
`200 OK`	HTML/JSON	Request successful; content delivered.
`302 Found`	Redirect	Redirect after POST (login, form submission).
`400 Bad Request`	HTML/JSON	Invalid data or incomplete request.
`401 Unauthorized`	HTML/JSON	Authentication required or expired.
`403 Forbidden`	HTML/JSON	Access denied; role or ownership mismatch.
`404 Not Found`	HTML/JSON	Requested resource unavailable.
`500 Internal Server Error`	HTML/JSON	Unexpected error; logged server-side.

This appendix serves as a functional reference for developers, maintainers, and future contributors to the system.