TECHNICAL UNIVERSITY OF CRETE

ELECTRICAL & COMPUTER ENGINEERING

Services in the Computing Cloud & Fog

Semester Assignment

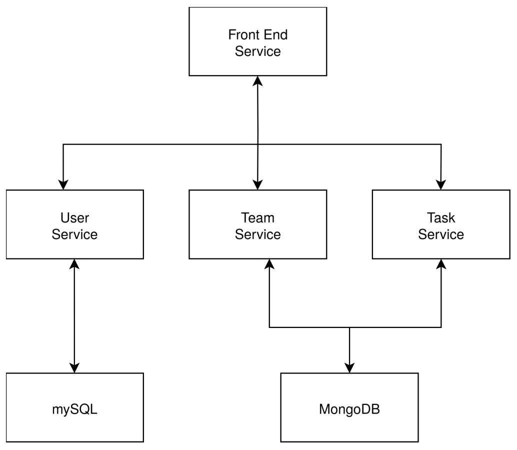
Due by: January 9th 2026
Course Instructor: Prof. Evripidis Petrakis
Assignment Supervisors: Prountzos Athanasios, Tzavaras Aimilios

Name	Georgios-Angelos Gravalos
AM	2021030001

This report provides a brief explanation of the development methods that concern the project.
If you're looking for information on how to run the project locally on your machine, you can take a look at the `README.md` file in the root directory of the project.

Introduction

The goal of this assignment was to develop a Jira/Trello-like application, a project management application, based on the microservices architecture.
The goal of the microservices architecture is to decouple the application into smaller, independently-deployed services, which are developed and managed separately. Therefore, we split the project in 4 distinct services : User service, Team service, Task service, and Front End service.
Each of the services lives in its own Docker container, has its own dependencies file (requirements.txt) and Dockerfile, as well as its own source code files.
The 3 back-end services communicate with each other synchronously, with REST APIs over HTTP, as well as with their respective databases. They also communicate with the Front End service.



Implementation Technologies

For **the back-end**, we used Python + Fast API framework, whereas the **front-end** was implemented with React + Vite. More details on the implementation will follow below, separately for each service.

User Service (Port 8001)

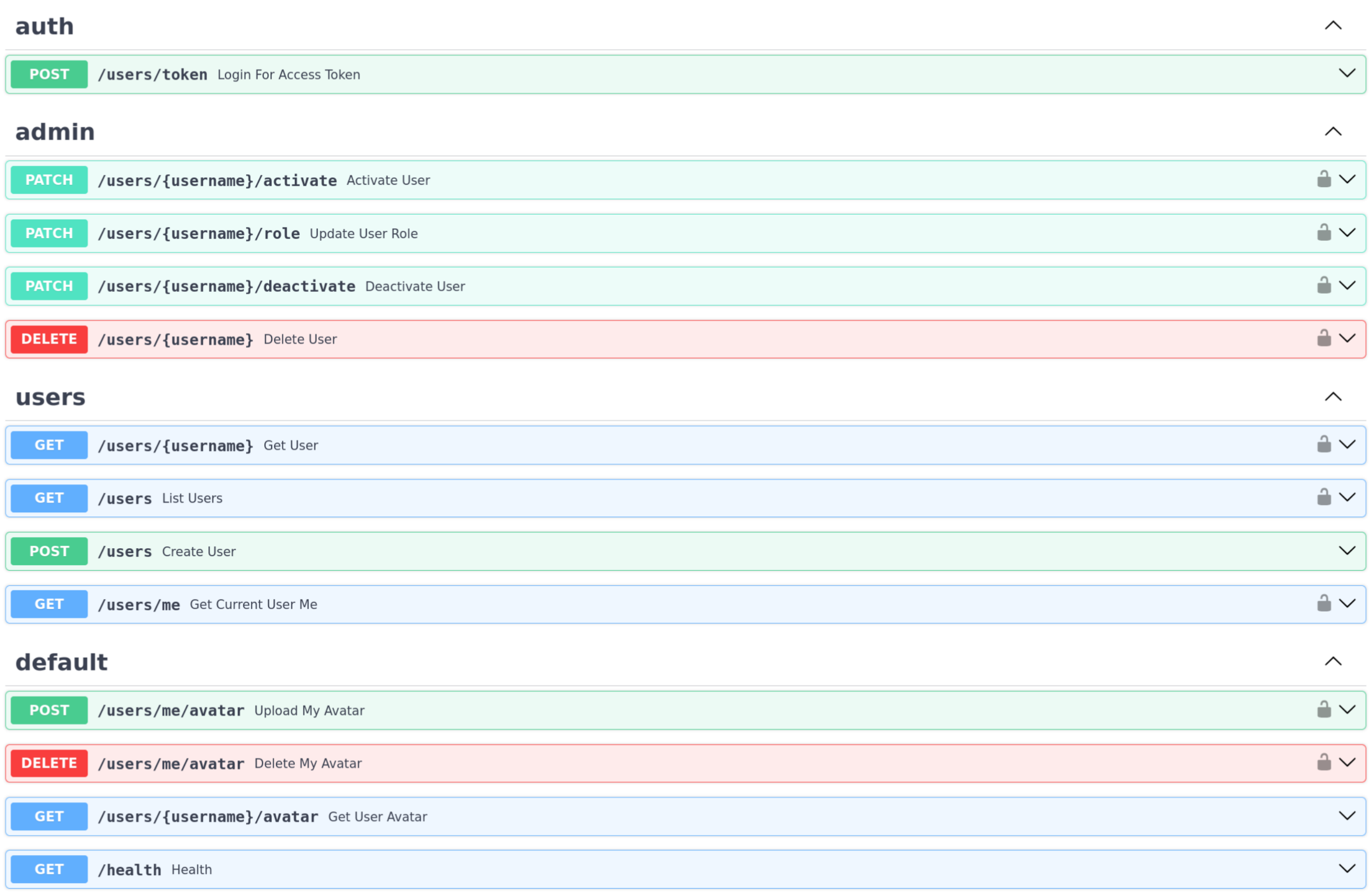
The User Service **fulfills all of the assignment's demands**. It handles user registration, log in, authentication and authorization.
The service was built with Python + FastAPI framework, and it uses a relational mySQL database, since it is ideal for structured user data (username, first and last name, email, password, role, active status).
Some key features of the service are:

- All user identification/authentication processes are done using JWTs. The User service is the only service that produces the tokens, and they are in turn passed to the Team and Task service, so they can validate user identities and roles.
- Password hashing before it is stored in the database. During log in, the input password is also hashed and compared with the password hash stored in the MySQL.
- Utilization of SQLAlchemy to handle queries to the database, instead of relying on string concatenation to execute the queries, which would introduce the risk of SQL-i. SQLAlchemy sanitizes the input by ensuring that the user input is treated strictly as data literals, and not as executable SQL code.
- Syncronous inter-service communication with the other 2 services, since often times it is required that the team or task service need to validate the user's identity for the Role-Based Access Control features (some functions can only be accessed by the admin, or the team leader, or both. Therefore they're restricted to simple members)

The files of the service are:

- `main.py`
- `db.py`
- `Dockerfile`
- `requirements.txt`
- `models.py` : Contains the SQLAlchemy data models that map Python classes to MySQL tables. This is the User class with all its parameters (username,password, etc.) and the Role enum eclass (MEMBER, TEAM_LEADER, ADMIN)
- `schemas.py` : Uses Pydantic models to validate format of ingoing requests, as well as outgoing responses. An example of why this is important is two of the schemas in `schemas.py` : UserCreate and UserOut. UserCreate has the username, first and last name, email, password. When we access an endpoint to get user info, we cannot use this schema, since a user's password should not be visible by anyone other than the user themselves. This is where the UserOut schema comes in, which provides username, first and last name, email and other information relevant to a user's profile such as activity status, role, and avatar file path.
- `security.py` : It handles all the security logic, such as password hashing, JWT token creation, authorization and authentication by accessing a user's token to reveal their identity and rights within the app (from their role).
- `routes.py` : This is where we define the entire logic of the service, and map it to REST endpoints.

This file structure are not unique to the User Service. The Team and Task service follow the exact same logic.
The endpoints were tested with both the Swagger UI, and Postman.
These are the endpoints that can be found at port 8001:



auth

POST `/users/token` Login For Access Token

admin

PATCH `/users/{username}/activate` Activate User

PATCH `/users/{username}/role` Update User Role

PATCH `/users/{username}/deactivate` Deactivate User

DELETE `/users/{username}` Delete User

users

GET `/users/{username}` Get User

GET `/users` List Users

POST `/users` Create User

GET `/users/me` Get Current User Me

default

POST `/users/me/avatar` Upload My Avatar

DELETE `/users/me/avatar` Delete My Avatar

GET `/users/{username}/avatar` Get User Avatar

GET `/health` Health

Team Service (port 8002)

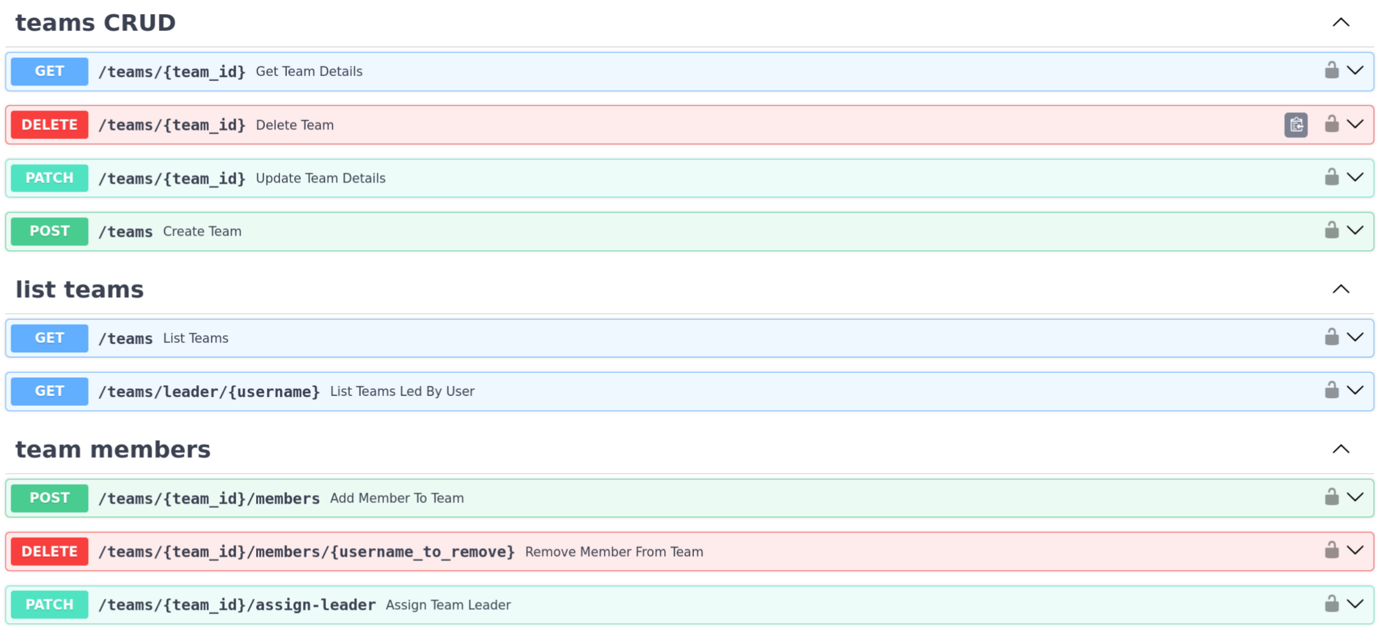
The Team service also fulfills all of the assignment's demands.
For the Team service, we opted for the MongoDB with the Motor asynchronous driver, for non-blocking and high throughput access to the database. We chose a non-relational database, since it is more suitable for team structures with flexible schemas (we'll elaborate further on these right below).
The Team service makes synchronous REST HTTP calls to the User service in many instances, such as :

- Validating a team leader. When a team is created and a team leader is assigned, the team service communicates with the user service to confirm that the username provided to the endpoint is an existing user.
- Same process is repeated when adding a member to a team.
- During user promotion/demotion, which happens automatically (will elaborate below)

There was an ambiguity in the assignment prompt concerning the handling of user roles, which was clarified, and now the logic is as such:
Admins can't change user roles freely. A user who has just been created is assigned the "MEMBER" role. This roles changes automatically.

- If the user leads 0 teams, and is assigned as a team leader to their first team, then the team service updates the user service, and their role immediately changes to "TEAM_LEADER".
- If a user leads 1 or more teams and they are assigned to lead another team, then their role simply remains "TEAM_LEADER".
- If the admin attempts to delete a user (in user service), then the user service sends a request to the team service to find out whether a user is the leader of one or more teams. In the case that they are, then the system throws a notification, informing the admin that deletion of the user is not possible until all their teams have been reassigned to other leaders. When the user's role changes back to "MEMBER", then and only then is deletion possible.
- If a team is deleted by the admin, then the team leader's status changes back to "MEMBER" if this was the only team they were leading, or remains "TEAM_LEADER" if they lead other teams as well.

The file layout is the same as the user service.
These are all the endpoints of the Team service to be found at port 8002:



teams CRUD

GET /teams/{team_id} Get Team Details
DELETE /teams/{team_id} Delete Team
PATCH /teams/{team_id} Update Team Details
POST /teams Create Team

list teams

GET /teams List Teams
GET /teams/leader/{username} List Teams Led By User

team members

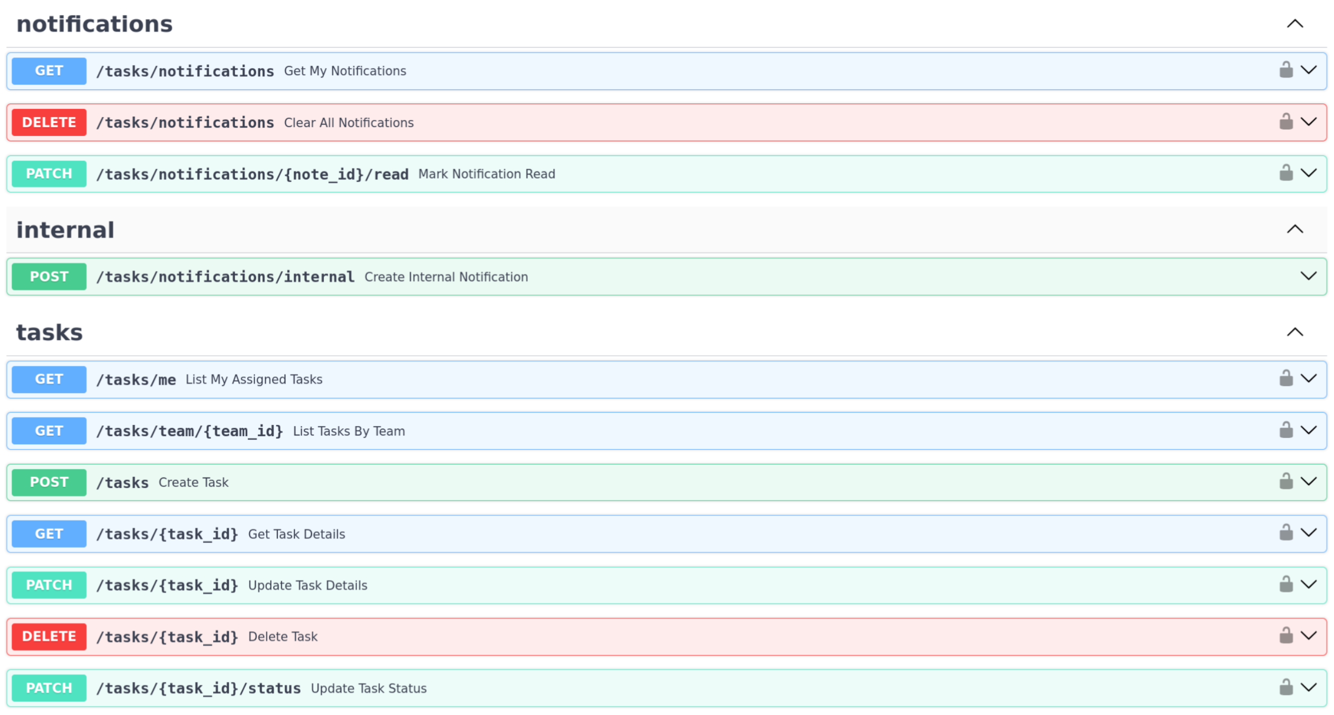
POST /teams/{team_id}/members Add Member To Team
DELETE /teams/{team_id}/members/{username_to_remove} Remove Member From Team
PATCH /teams/{team_id}/assign-leader Assign Team Leader

Task Service (port 8003)

The Task service also fulfills all of the assignment's demands.
The endpoints support task creation, deletion, updates, status tracking (TODO/IN PROGRESS/DONE), comments from anyone who belongs to the team, and files upload and deletion.
Again, the service is built with Python + FastAPI, with MongoDB, for the same reason it was used in the Team service : Tasks are complex objects, which contain lists of comments and files, and a non-relational database facilitates the process of retrieval.
The files layout is once again the same as the previous two services, with an extra directory `/task_files`, a persistent Docker volume that contains all the files uploaded to tasks within the app.
What's more, the **notification service** has also been implemented. Not as separate service, but embedded within the task service, since this service is the one that utilizes notifications the most. The internal endpoint to push notifications is not publicly visible, but is exposed to the other services so that they can use it. All notifications are stored persistently in the MongoDB, therefore even if events that concern a user occur at a time when they are logged out, they will see all the alerts in their notifications tray whenever they log back in.
The task service communicates with both the user and the team service. Some important examples that concern the implementation are:

- When a team leader creates a new task and assigns it to a member, the Task service sends a synchronous request to the User service, to verify that the user exists and is a member of the team. This however is bypassed in the front end, since the task assignment has a dropdown list of all users who are members to the team, not free user input.
- When a team is deleted, the team service calls an internal endpoint at the task service to delete all tasks within the team, to prevent leftover data.
- When a user is added to a team, the team service calls the task service to generate a system notification for the user.

The endpoints of the Task service can be accessed at port 8003:


notifications

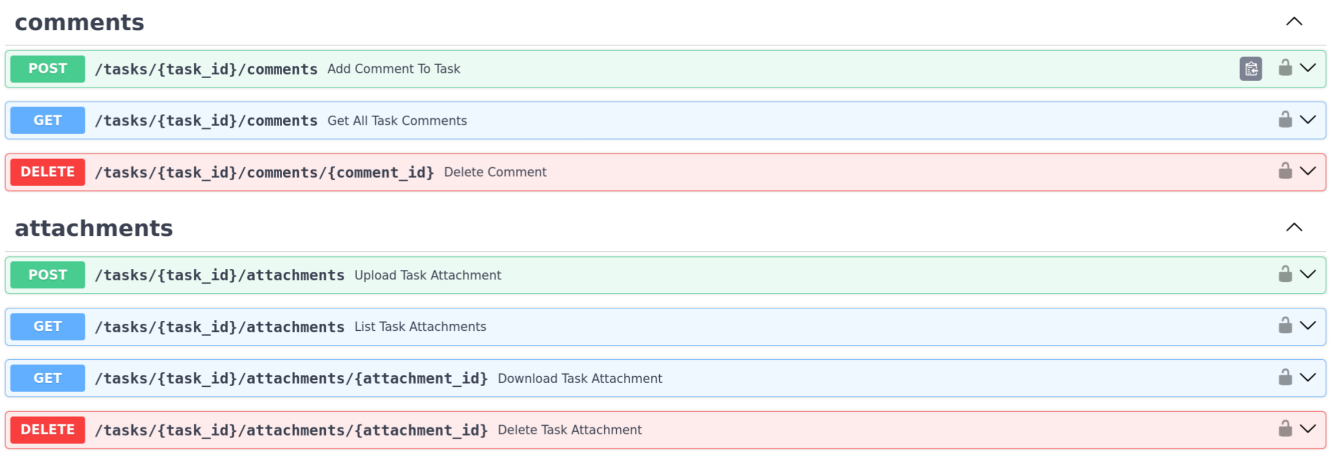
GET /tasks/notifications Get My Notifications
DELETE /tasks/notifications Clear All Notifications
PATCH /tasks/notifications/{note_id}/read Mark Notification Read

internal

POST /tasks/notifications/internal Create Internal Notification

tasks

GET /tasks/me List My Assigned Tasks
GET /tasks/team/{team_id} List Tasks By Team
POST /tasks Create Task
GET /tasks/{task_id} Get Task Details
PATCH /tasks/{task_id} Update Task Details
DELETE /tasks/{task_id} Delete Task
PATCH /tasks/{task_id}/status Update Task Status



comments

POST /tasks/{task_id}/comments Add Comment To Task

GET /tasks/{task_id}/comments Get All Task Comments

DELETE /tasks/{task_id}/comments/{comment_id} Delete Comment

attachments

POST /tasks/{task_id}/attachments Upload Task Attachment

GET /tasks/{task_id}/attachments List Task Attachments

GET /tasks/{task_id}/attachments/{attachment_id} Download Task Attachment

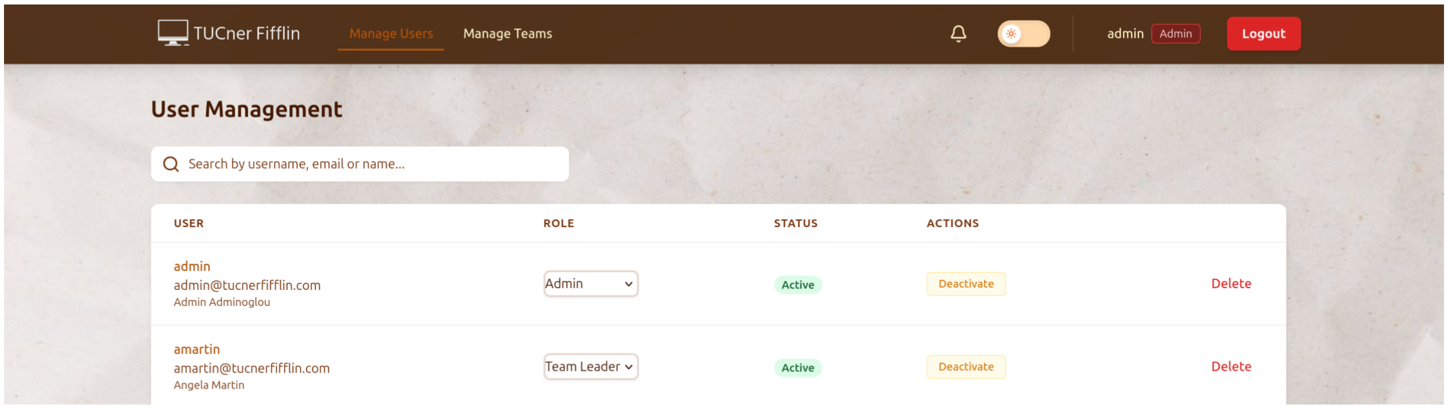
DELETE /tasks/{task_id}/attachments/{attachment_id} Delete Task Attachment

Front End Service (Port 80 or 5173)

The front-end service utilizes the JS React library, instead of vanilla JS, as well as Vite.
Also, Tailwind CSS was used instead of vanilla CSS, which facilitated building the website, and provided the opportunity for a dark/light mode switch, using a custom ThemeContext. The design choices behind the front end are references to the TV show "The Office", therefore both the light and dark mode follow an office-themed style.
Client-side routing was done using react-router-dom, which enabled instant transition between pages and hot-reloading, without needing to refresh the browser page.
What's more, we implemented an AuthContext to manage the user's log in state. Upon log in, the JWT is stored, and the user's role is broadcast to the entire app.

We have implemented two different pages as far as the user roles are concerned. The log in page is common for everybody, but the user is redirected to the admin page if they log in as an admin, or to the members page if they log in as any other role (team leader or member).

- Admin page:

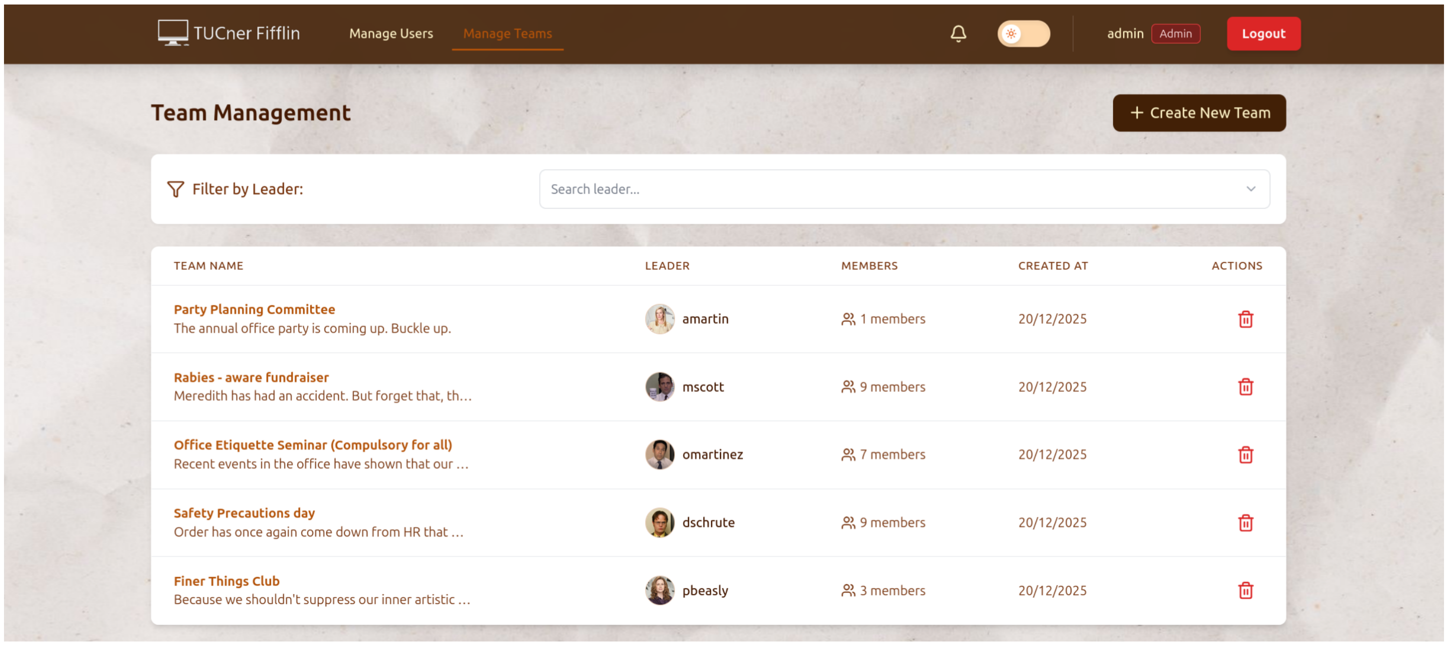


User Management

Search by username, email or name...

USER	ROLE	STATUS	ACTIONS	
admin admin@tucnerfifflin.com Admin Adminoglou	Admin	Active	Deactivate	Delete
amartin amartin@tucnerfifflin.com Angela Martin	Team Leader	Active	Deactivate	Delete

Admins have two sections : The **User Management** and the **Teams Management** pages, as well as a light/dark theme switcher, a notifications dropdown, and a logout button. From User management, they're able to view all users within the system, update their roles (to the extend where it doesn't interfere with the functionality of the system, as mentioned further above), deactivate/activate users, or delete them entirely. A search bar functionality is also provided at the top of the page, which allows the admin to search for users individually.



Team Management

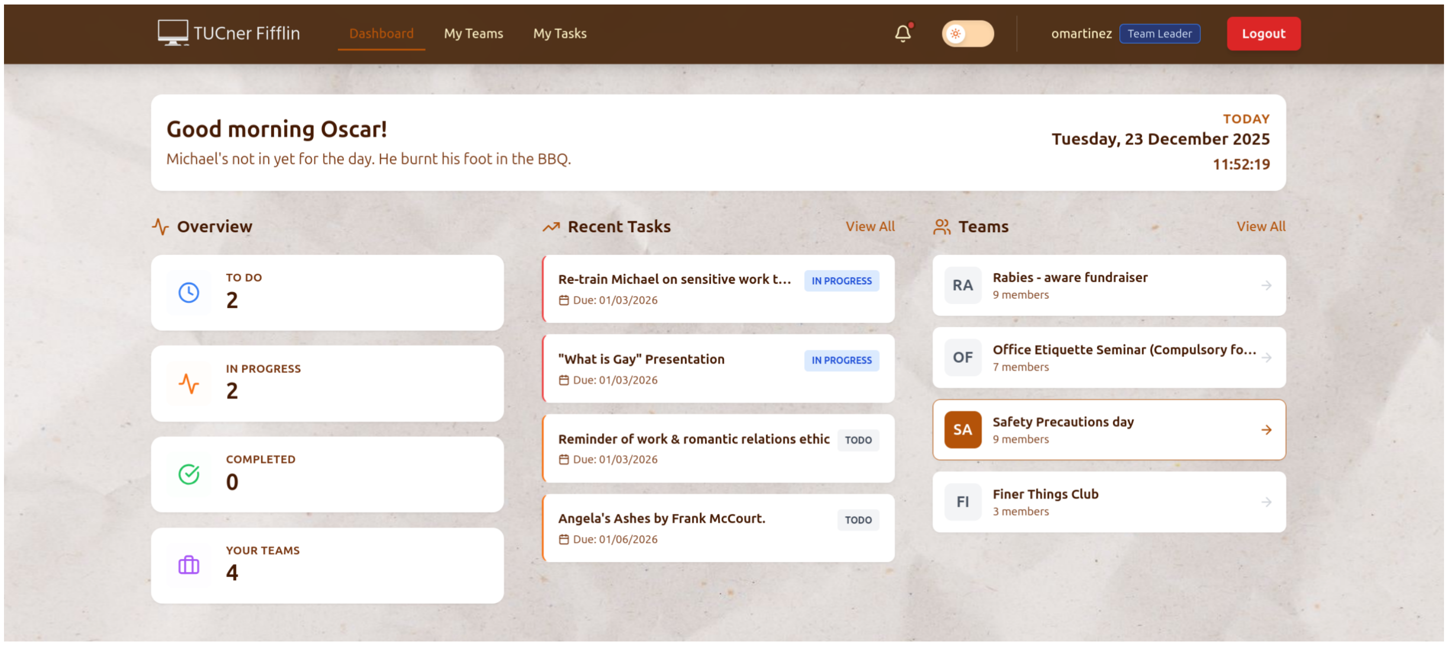
+ Create New Team

Filter by Leader: Search leader...

TEAM NAME	LEADER	MEMBERS	CREATED AT	ACTIONS
Party Planning Committee The annual office party is coming up. Buckle up.	amartin	1 members	20/12/2025	🗑
Rabies - aware fundraiser Meredith has had an accident. But forget that, th...	mscott	9 members	20/12/2025	🗑
Office Etiquette Seminar (Compulsory for all) Recent events in the office have shown that our ...	omartinez	7 members	20/12/2025	🗑
Safety Precautions day Order has once again come down from HR that ...	dschrute	9 members	20/12/2025	🗑
Finer Things Club Because we shouldn't suppress our inner artistic ...	pbeasly	3 members	20/12/2025	🗑

From the Teams Management page, the admin can view all teams that exist in the application, click on them, and view team information.
As mentioned in the report, the admin can create teams, but they cannot add/remove members. Also, they can view all of the team's tasks, however they cannot create/delete tasks within the team, as these are responsibilities that belong to the team leader. Finally, they can view files/comments inside a task, but they cannot add a file or comment, and neither can they delete a file or a comment. This can only be done by team leaders, or the individual member who added the file or created the comment on the task.

- **Members page:** The members page includes 3 sections : Their dashboard page, their teams page (teams that only THEY belong to) and their tasks page (tasks that have been assigned to THEM only). The navigation bar at the top also contains the theme switch, the notification dropdown and the logout button.



Good morning Oscar!
Michael's not in yet for the day. He burnt his foot in the BBQ.

TODAY
Tuesday, 23 December 2025
11:52:19

Overview

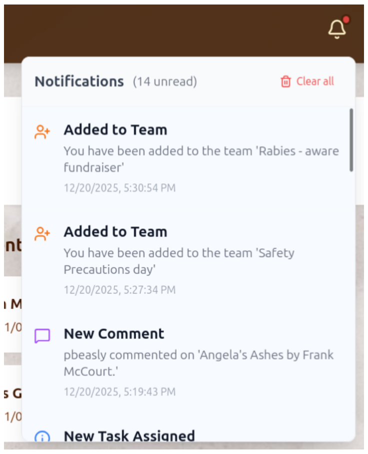
TO DO 2
IN PROGRESS 2
COMPLETED 0
YOUR TEAMS 4

Recent Tasks View All

Re-train Michael on sensitive work t... — IN PROGRESS — Due: 01/03/2026
"What is Gay" Presentation — IN PROGRESS — Due: 01/03/2026
Reminder of work & romantic relations ethic — TODO — Due: 01/03/2026
Angela's Ashes by Frank McCourt. — TODO — Due: 01/06/2026

Teams View All

RA Rabies - aware fundraiser — 9 members
OF Office Etiquette Seminar (Compulsory fo... — 7 members
SA Safety Precautions day — 9 members
FI Finer Things Club — 3 members


Notifications (14 unread)
Clear all

Added to Team
You have been added to the team 'Rabies - aware fundraiser'
12/20/2025, 5:30:54 PM

Added to Team
You have been added to the team 'Safety Precautions day'
12/20/2025, 5:27:34 PM

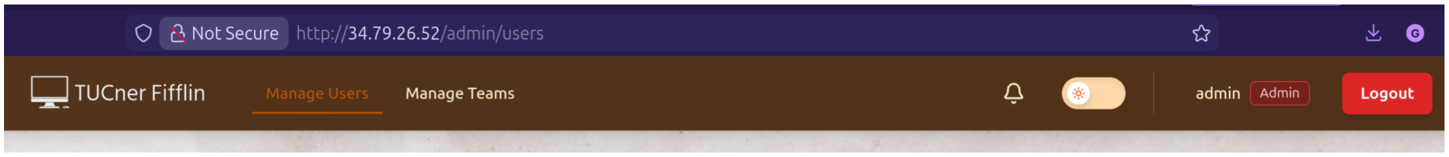
New Comment
pbeasly commented on 'Angela's Ashes by Frank McCourt.'
12/20/2025, 5:19:43 PM

New Task Assigned

The dashboard contains a quick recap of their activity on the application : their teams, their tasks and their status, as well as a welcome bar with the time and date. Apart from the Dashboard, the other two sections are My Teams and My Tasks. A member of a team can view all of the team's members, as well as the tasks within the team. They can comment or add files to any task. If a task is assigned to them, they can edit the task's status to let the other members know about their progress. They cannot edit other task info such as the name or description, or even delete the task entirely, since this falls under the team leader's responsibilities

Migration to the GCP

Last part of the assignment was the migration to the GCP.
To host the project on the GCP, we created our account and headed to Compute engine VM instances, and created the host VM running Ubuntu. We chose the e2-medium VM type, which was sufficient for our project to run seamlessly, and we promoted the external IP address from ephemeral to static, so that we wouldn't have to keep changing the IP address in the project's files.
We configured the firewall to allow traffic on port 80 (HTTP) (HTTPS wasn't required, therefore port 443 wasn't needed), to expose the application's front end to the Internet. We intentionally didn't expose ports 8001,8002,8003, since this would simply allow any external user to access `http://34.79.26.52:8001/docs` and view our internal endpoints.
Since we did not expose ports 8001-8003, we needed to modify the `nginx.conf` file to route traffic internally. For example, a request to `http://34.79.26.52:8001/login` is proxied to `http://34.79.26.52/api/users/login`. Any request is forwarded internally to the proper service container, on the proper port, within the Docker network.
Other files such as `axios.js, endpoints.js`, as well as the .env files in the root and the front end directories were modified to become compatible with this change.
These changes are now shown in the deliverables, since the deliverables contain the local development project, not the production project.
We uploaded the .zip file to the VM's files, unzipped the project, cd'd into the project directory, and ran docker compose up. The app became accessible at `http://34.79.26.52`.


Not Secure
http://34.79.26.52/admin/users
TUCner Fifflin
Manage Users
Manage Teams
admin
Admin
Logout