User Management (Part 1) :

Registration :

Features of Registration Service

- **User Account Creation: Allows new students to create their secure login credentials (email/password) for the virtual lab system.**

- **Detailed Profile Collection & Storage: Gathers and saves essential personal, educational, and course enrollment information during the initial signup process.**

- **Data Validation: Performs validation on submitted data, including required fields checks on the backend and leveraging Supabase Auth's built-in email format and password policy checks.**

- **Post-Registration Event Triggering: Designed to automatically initiate an event upon successful creation of a user's profile in the database, enabling downstream actions by other microservices (like Notifications).**

- **Collected Information Fields: Includes Email, Password, First Name, Last Name, Date of Birth, Phone Number, Address, Major, Enrolled Courses, Academic Year, and Previous Education.**

Implementation Details

Technology Stack

- **Frontend:**
    - **React (bootstrapped with Vite)**
    - **MUI (Material UI) for UI components and styling**
    - **Axios for API communication**
- **Backend:**
    - **Node.js with the Express.js framework**
    - **Supabase client library (@supabase/supabase-js)**
    - **CORS for enabling Cross-Origin Resource Sharing**
    - **dotenv for environment variable management**

- **Database:**
    - **Supabase (utilizing managed PostgreSQL for a custom profiles table and Supabase Auth for core user authentication)**
- **Eventing:**
    - **Supabase Database Webhooks listening to profiles table inserts**
    - **Supabase Edge Functions (Deno/TypeScript) for event forwarding**
- **Containerization:**
    - **Docker (using Dockerfiles for frontend and backend)**
    - **Docker Compose for local multi-container orchestration**

API Design

RESTful API approach centered around a single primary endpoint exposed by the backend service:

- **POST /register: Accepts a JSON payload containing the complete set of user details (authentication credentials and profile information).**
    - **This endpoint handles the two-stage process:**
        1. **Creates the user securely via Supabase Auth (supabase.auth.signUp)**
        2. **Inserts the associated detailed profile information into the profiles table using elevated privileges (service_role key) to link it correctly immediately after signup**
    - **Returns appropriate success (201) or error (400, 500) responses**
    - **The database insert indirectly triggers the post-registration event**

Data Storage

Leverages Supabase for persistence:

- **Core authentication data (User ID, email, hashed password) is stored and managed within the secure Supabase Auth system (auth.users table)**
- **Extended profile details (First Name, Major, Courses, etc.) are stored in a dedicated profiles table within the Supabase PostgreSQL database, linked via a foreign key (id) to the auth.users table**
- **Row Level Security (RLS) policies are configured on the profiles table, although the initial insert by the backend currently bypasses these using the service_role key**

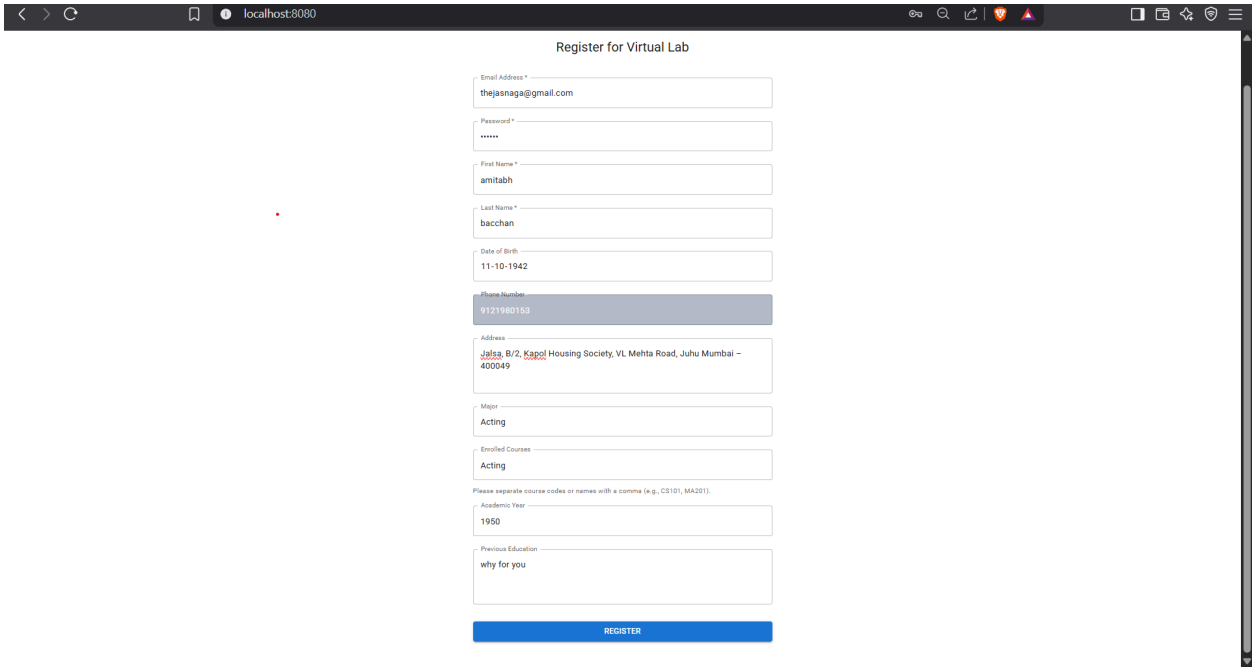
Frontend Integration

A React single-page application (frontend/) provides the user interface:

- **Includes a dedicated form component (RegisterForm.jsx) built with MUI components (TextField, Button, Alert, etc.) to capture all required user details**
- **Client-side state is managed using React hooks (useState)**
- **On submission, it formats the data (e.g., parsing comma-separated courses) and sends it via an asynchronous POST request (using Axios) to the backend's /register endpoint**
- **Displays success or error feedback messages to the user based on the API response**

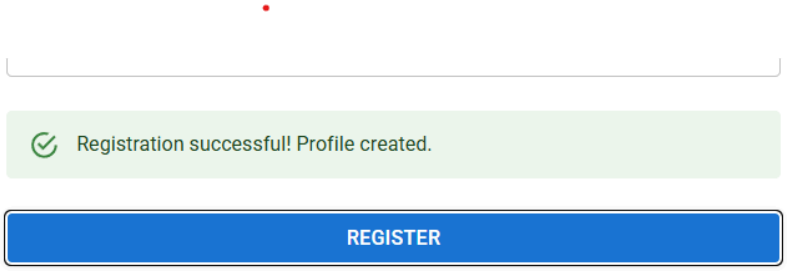
Screen shot 1:

This is the main registration page with filled values



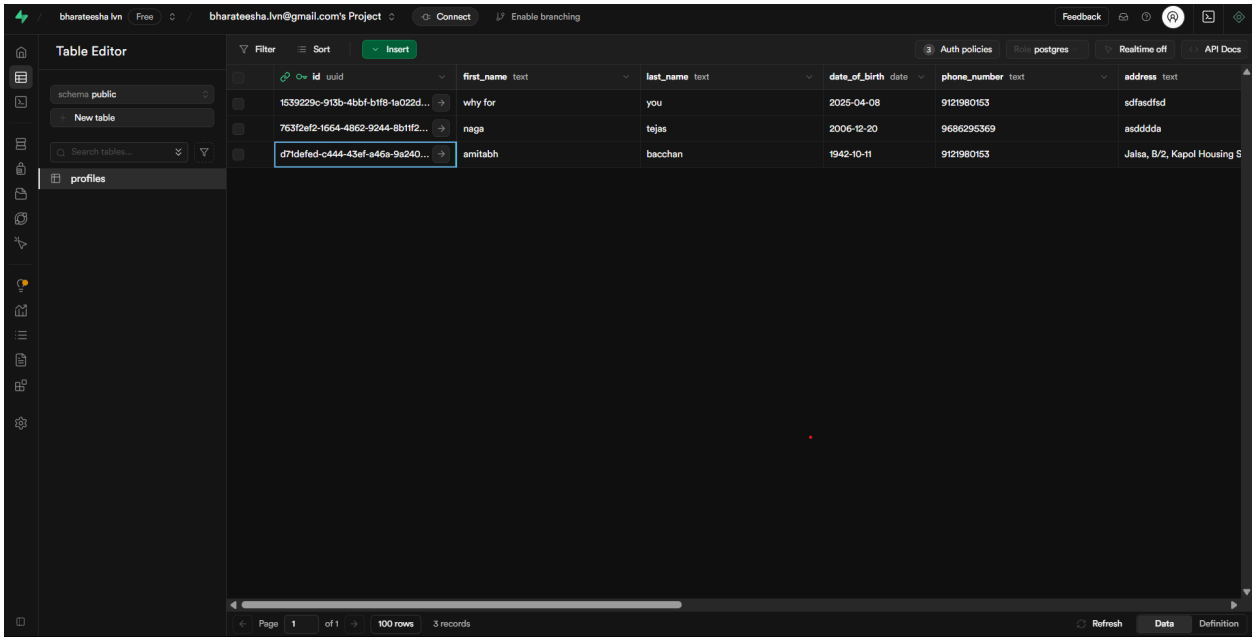
Screen shot 2:

This shows that registration has been succesfull



Screen shot 3:

The newly registered entry has been added in the supabase database



Profile Management :

1. Features of Profile Management:

- User Profile Creation: Allows students to create their profiles by providing essential information.
- User Profile Retrieval: Enables students to view their profile details.
- User Profile Update: Allows students to modify their existing profile information.
- Basic Personal Details: Includes fields for First Name, Last Name, Phone Number, and Address.
- Educational Information: Captures details such as Major and Previous Education.
- Course Enrollment: Students can select the courses they are currently enrolled in (Operating Systems, Data Structures and Algorithms, Software Engineering).

2. Implementation Idea:

- **Technology Stack:** Python (Flask), Flask-WTF for form handling, Flask-CORS for enabling Cross-Origin Resource Sharing, and an in-memory dictionary for data storage (for this example). Docker for containerization.
- **API Design:** RESTful API with endpoints for creating, retrieving, and updating user profiles.
  - `POST /api/profiles`: Creates a new user profile.
  - `GET /api/profiles/<user_id>`: Retrieves the profile for a given user ID.
  - `PUT /api/profiles/<user_id>`: Updates the profile for a given user ID.
- **Data Storage:** Using an in-memory Python dictionary (`profiles`) where the key is the `user_id` and the value is a dictionary containing the user's profile data. **Note:** For a production environment, a persistent database (like PostgreSQL or MySQL) would be used.
- **Frontend Integration:** A simple HTML form with JavaScript to interact with the backend API.

3. Implementation Source Code:

app.py:

Contains the Flask backend logic for handling API requests and managing profile data.

index.html:

Defines the structure and elements of the Profile Management web page's user interface.

script.js:

Implements the frontend JavaScript logic for user interactions and communication with the backend API.

style.css:

Provides the styling and visual appearance for the Profile Management web page.

4. Use of Docker:

Docker is used to containerize the Profile Management microservice. This means:

- We create a `Dockerfile` that specifies the environment and dependencies needed to run the microservice (e.g., the base Python image, installing required libraries from `requirements.txt`, copying the application code).

- Docker builds an image from this `Dockerfile`, which is a lightweight, standalone, and executable package containing everything needed to run the application.
- We then run a container from this image. The container is an isolated process that runs the Profile Management microservice.
- Benefits of using Docker:
    - Consistency: Ensures the microservice runs the same way across different environments (development, testing, production).
    - Isolation: Provides isolation from other applications running on the same machine, preventing dependency conflicts.
    - Portability: The Docker image can be easily shared and run on any system with Docker installed.
    - Simplified Deployment: Makes it easier to deploy and manage the microservice.

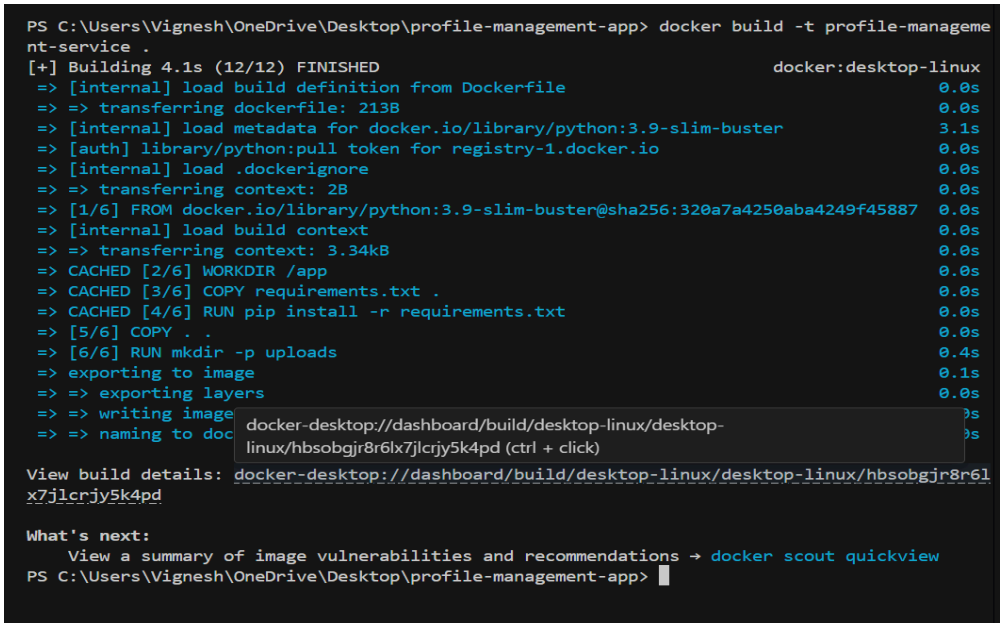
Dockerfile:

```
FROM python:3.9-slim-buster
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 5001
CMD ["python", "app.py"]
```

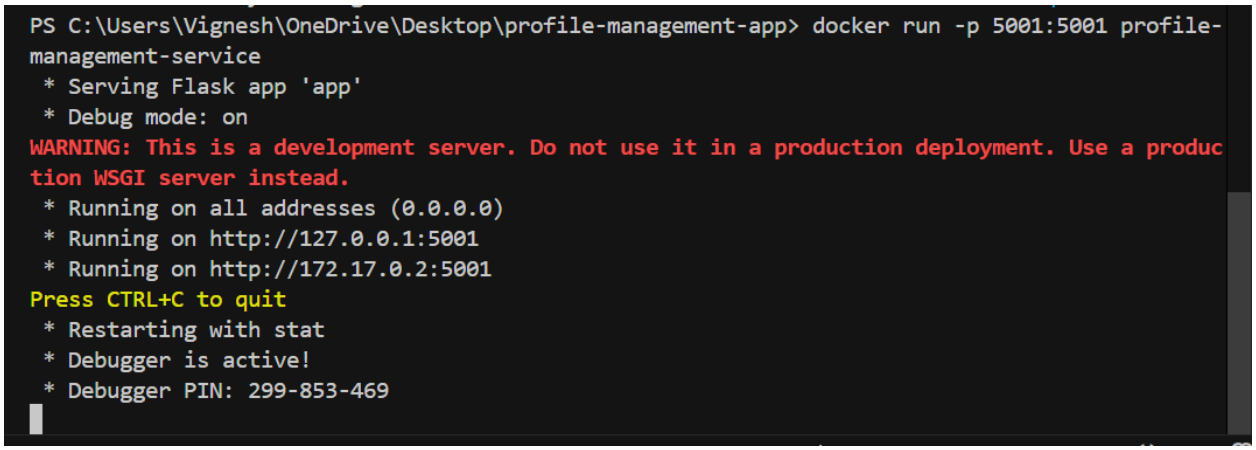
Running with Docker:

```
docker build -t profile-management-service .
docker run -p 5001:5001 profile-management-service
```

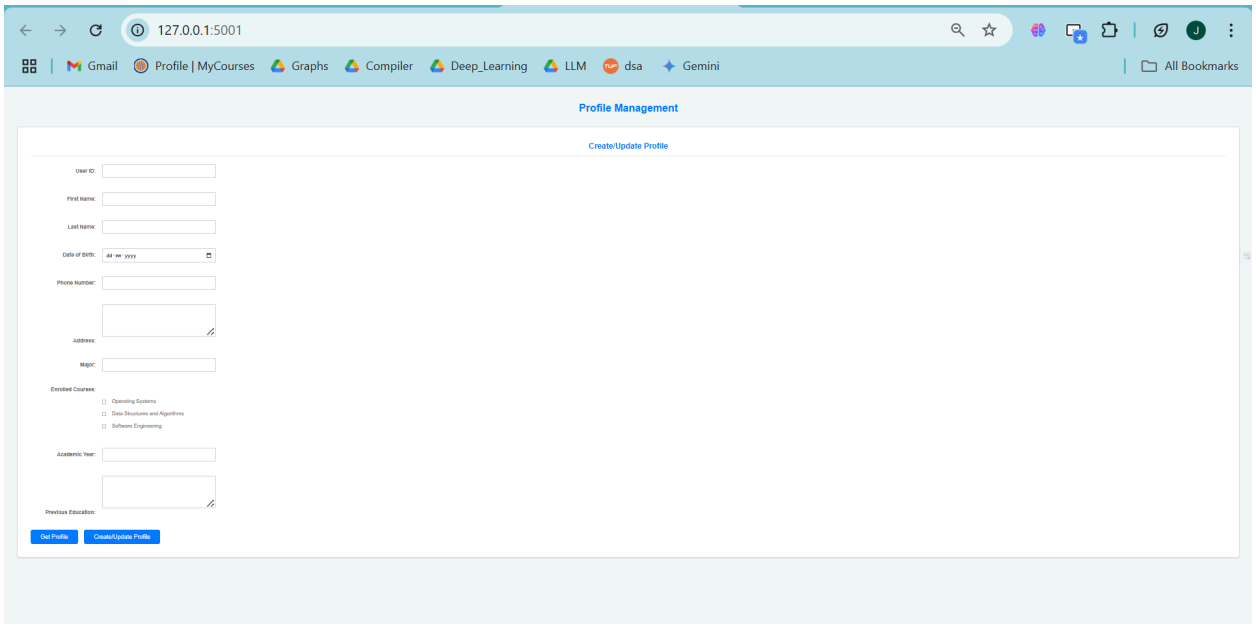
Building The Container



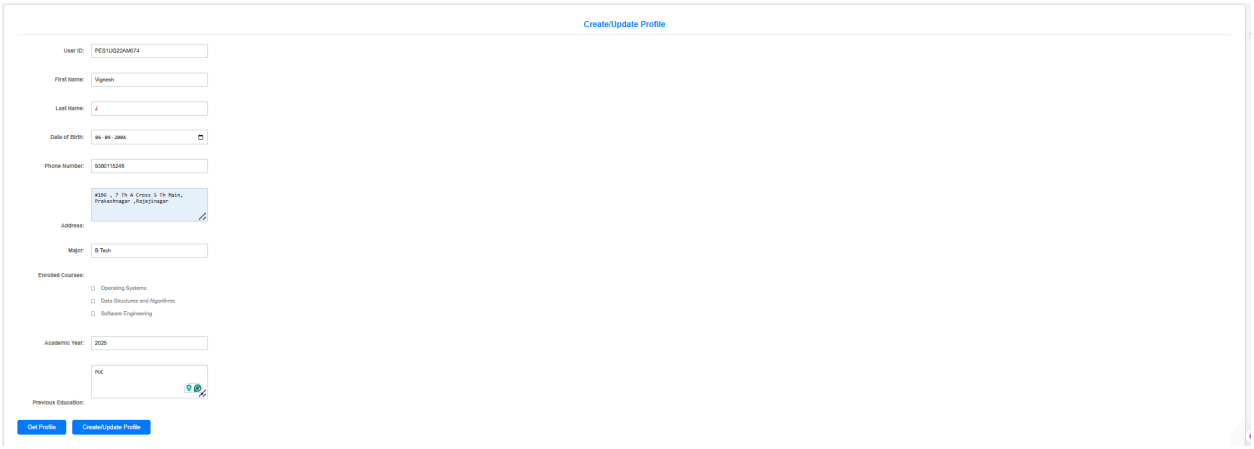
Container Running:

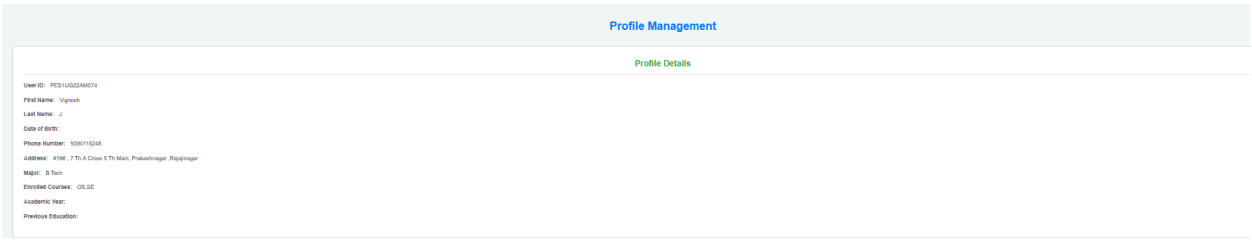


5.Output:

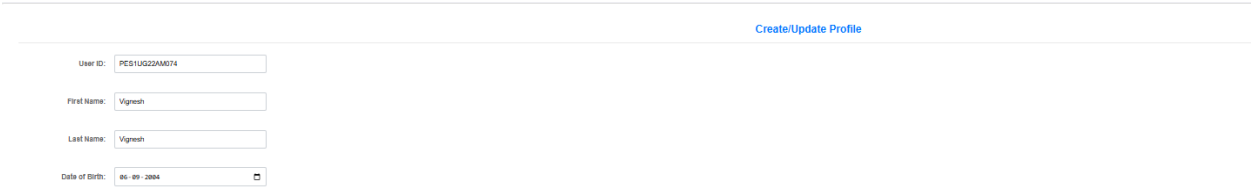


Entering Details :



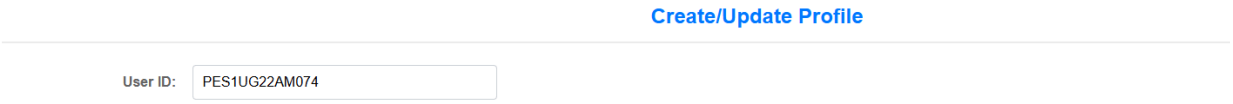


Updating the details : Updating Name

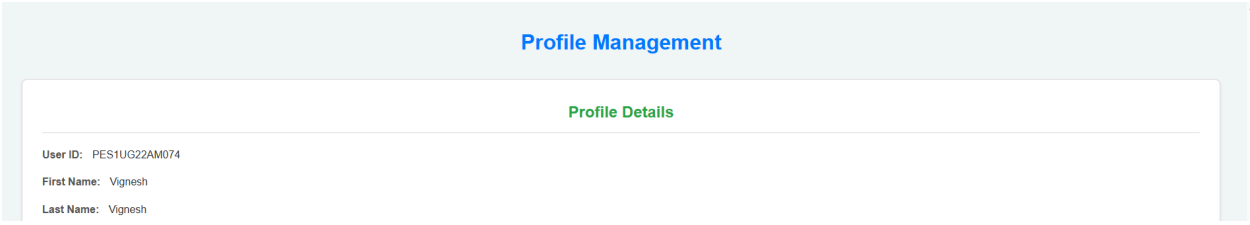


Details Updated :

Can Fetch The details By User ID :



Details Based On User Id:



6. How to Integrate with Other Microservices:

The Profile Management microservice can be integrated with other microservices in the virtual lab platform using various methods:

- **RESTful APIs:** Other microservices can communicate with the Profile Management microservice by making HTTP requests to its defined API endpoints (e.g., to retrieve user profile information). This is a synchronous communication method.
- **Message Broker (Asynchronous Communication):** If you are using a message broker like RabbitMQ or Kafka, the Profile Management microservice can publish events when a profile is created or updated. Other interested microservices (e.g., the Authentication or Notification service) can subscribe to these events and react accordingly.
- **Database Sharing:** While generally not recommended for strict decoupling, in some cases, microservices might need to access the same underlying database (e.g., if the Registration and Profile Management services share a user database). This approach requires careful management to avoid conflicts.

Authentication

Features of Authentication Service

- **User Authentication**: Provides secure login functionality for users to access the virtual lab system using email/password credentials.
- **Token Management**: Generates and validates JSON Web Tokens (JWT) for maintaining authenticated sessions.
- **Session Control**: Enables user logout and token validation capabilities.
- **Inter-service Authentication**: Offers dedicated endpoints for other microservices to verify user tokens and retrieve role information.
- **Role-based Authorization**: Verifies user permissions based on assigned roles in the system.

Implementation Details

Technology Stack

- **Frontend:**
    - **React (create-react-app)** for building the user interface
    - **React Router** for client-side routing and protected route implementation
    - **Axios** for API communication
    - **JWT-decode** for token parsing and validation
- **Backend:**
    - **Node.js with Express.js framework**

    - **bcryptjs** for password hashing and comparison
    - **jsonwebtoken** for JWT generation and verification
    - **Supabase client (@supabase/supabase-js)** for database operations
    - **CORS** for enabling Cross-Origin Resource Sharing
    - **dotenv** for environment variable management
- **Database:**
    - **Supabase** (PostgreSQL-based cloud database) for storing user credentials and role information
- **Containerization:**
    - **Docker** with multi-stage build process
    - **Docker Compose** for orchestrating the frontend and backend containers

API Design

RESTful API endpoints exposed by the authentication service:

- **POST /api/auth/login**: Accepts email and password, validates credentials against the database, and returns a JWT token upon successful authentication.
- **POST /api/auth/logout**: Handles user session termination on the client side.
- **GET /api/auth/validate**: Protected endpoint that verifies if a provided token is valid and returns user information.
- **POST /api/service/validate-token**: Internal endpoint for other microservices to verify user tokens.
- **POST /api/service/get-user-role**: Internal endpoint for role-based access control verification across microservices.
- **GET /health**: Provides service health check status.

Data Storage

Leverages Supabase for user data persistence:

- **User information**: Stores email, securely hashed password, and role assignments in a users table.
- **Security**: Implements Row Level Security (RLS) policies to protect user data.
- **Schema Design**: Structured to support seamless integration with other microservices like Registration and Profile Management.

Frontend Integration

React-based single-page application providing:

- **Login Form**: Clean, user-friendly interface for submitting credentials.
- **Protected Routes**: Components that restrict access based on authentication state.
- **Dashboard**: Displays user information and connected microservices upon successful login.

- **Context-based State Management**: Uses React Context API to maintain authentication state across the application.
- **Token Storage**: Securely stores authentication tokens in browser localStorage with automatic expiration handling.

Inter-service Communication

- **Service Registry**: Maintains configuration for connecting to other microservices.
- **Token Verification**: Provides a secure method for other services to validate user authentication.
- **Service-to-Service Authentication**: Implements a dedicated token system for microservice-to-microservice communication.

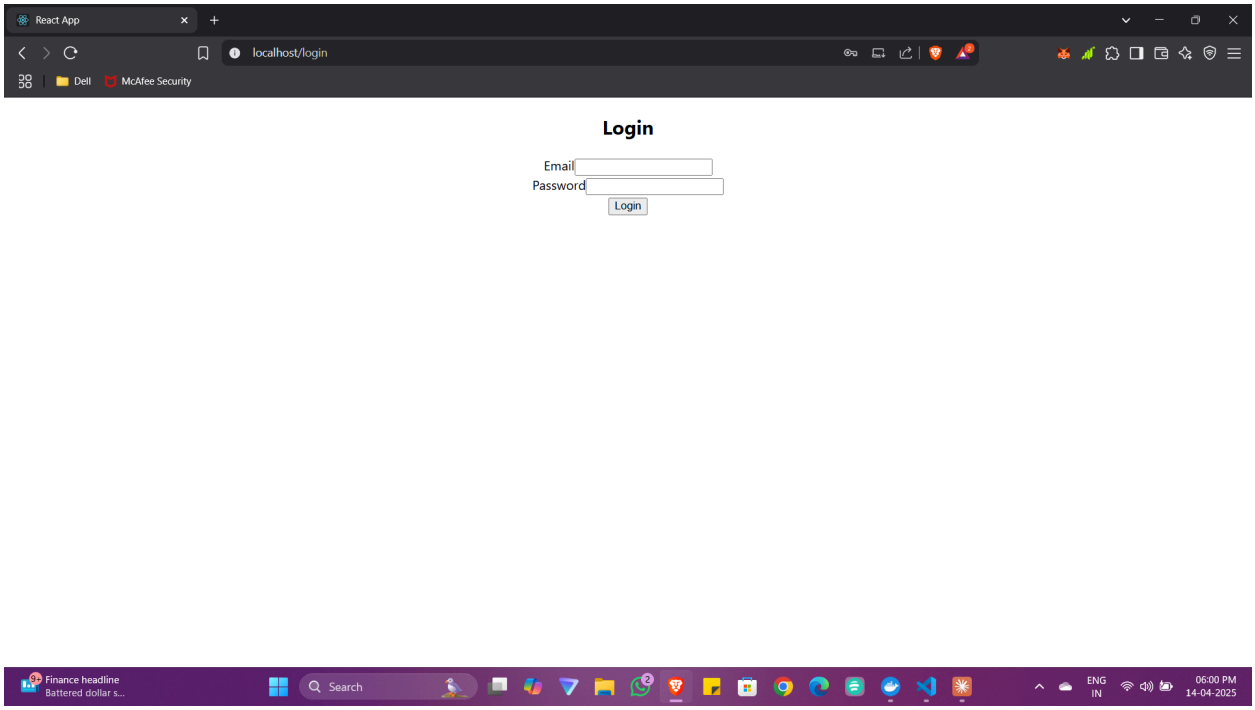
Security Features

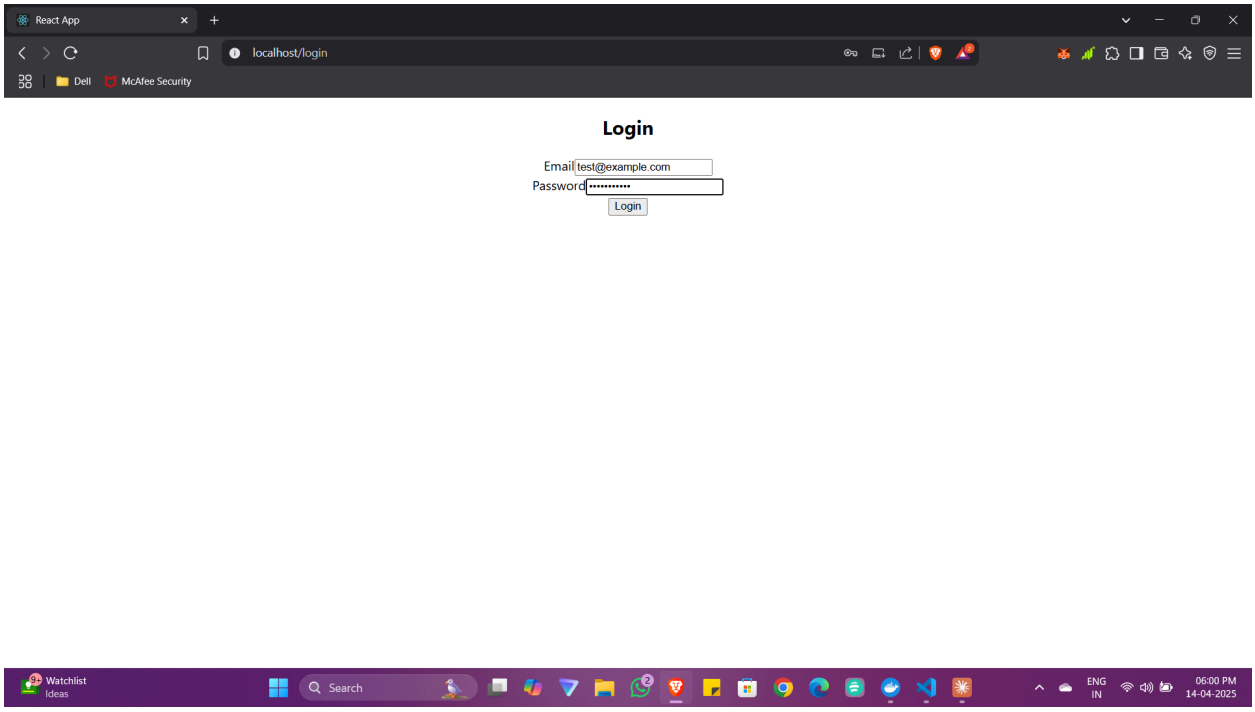
- **Password Hashing**: Uses bcryptjs to securely store user passwords.
- **JWT-based Authentication**: Implements stateless authentication using signed tokens.
- **Token Expiration**: Automatically invalidates tokens after a configurable period.
- **Cross-Origin Protection**: Implements proper CORS settings to prevent unauthorized access.
- **Service-level Authorization**: Verifies service identity for internal API calls between microservices.

This authentication microservice is designed to be lightweight, secure, and easily integrable with other components of the cloud-based virtual lab system, providing centralized authentication services across the entire application ecosystem.

Screenshots -

Frontend -


React App
localhost/login
Dell
McAfee Security
Login
Email
Password
Login
Finance headline
Battered dollar s...
Search
ENG
IN
06:00 PM
14-04-2025


React App
localhost/login
Dell
McAfee Security
Login
Email
test@example.com
Password
Login
Watchlist
Ideas
Search
ENG
IN
06:00 PM
14-04-2025

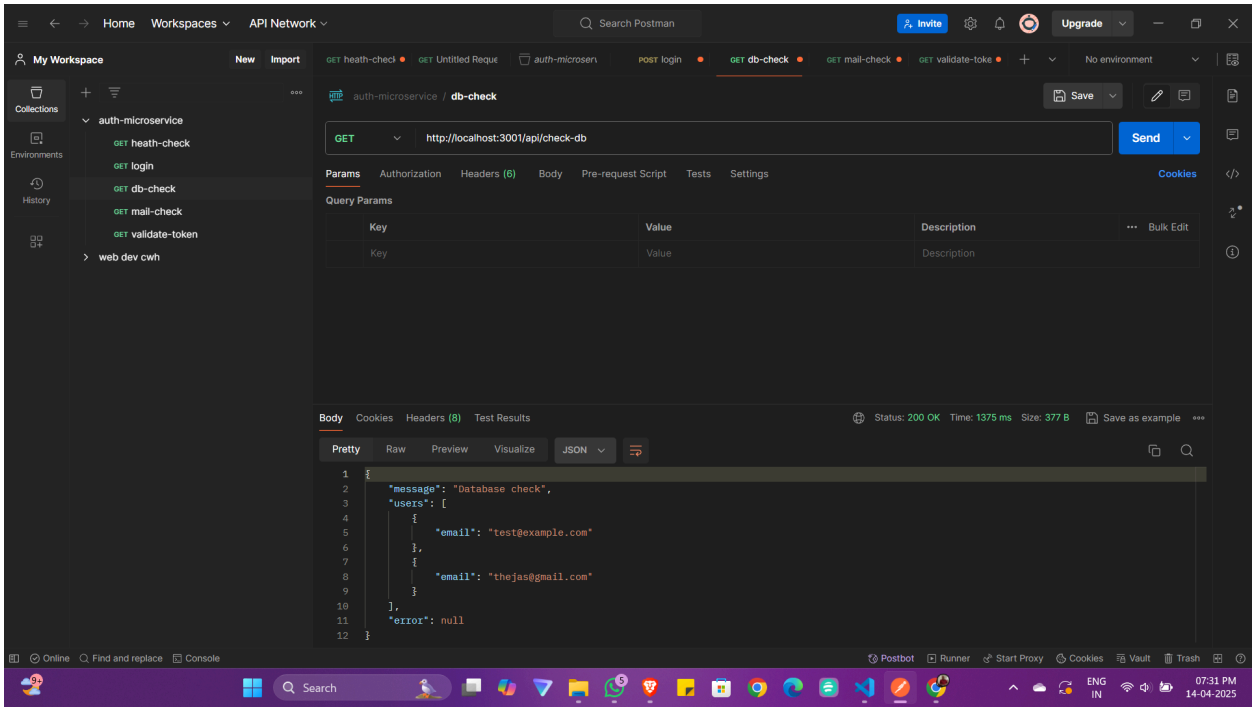
Dashboard

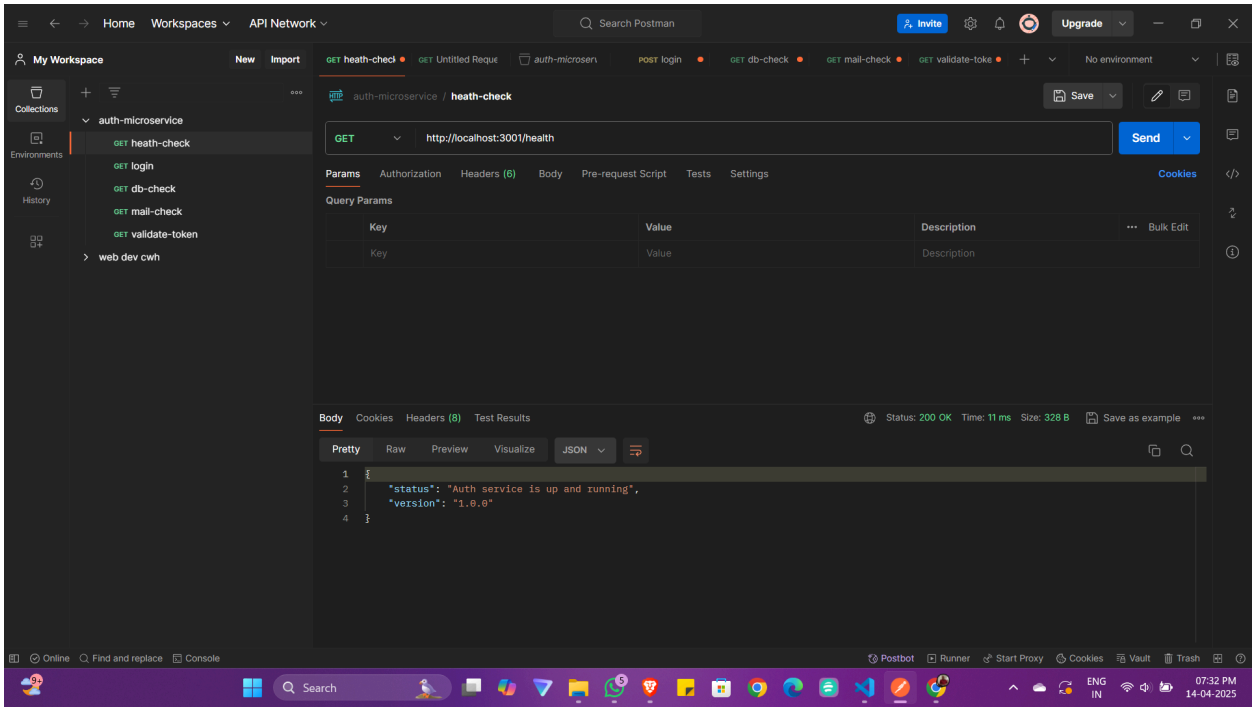
Welcome, test@example.com!

You are logged in as: **user**

Logout

Backend -





Role Based Access Control (RBAC)

This microservice provides centralized management of user roles, permissions, and the logic to determine if a user has the necessary authorization to perform specific actions within the virtual lab system.

1. Features of RBAC Service:

- **Role Management:** Allows creation and listing of roles (e.g., "admin", "editor", "viewer") with descriptions. Ensures role names are unique.
- **Permission Management:** Allows creation and listing of permissions (e.g., "documents:edit", "users:manage") with descriptions. Ensures permission names are unique.
- **Role-Permission Assignment:** Enables assigning specific permissions to roles and removing those assignments.

- **User-Role Assignment:** Enables assigning roles to specific users (identified by user ID) and removing those assignments.
- **Access Control Check:** Provides a core API endpoint (`/check`) that determines if a given user ID has a specific permission based on the roles assigned to them and the permissions assigned to those roles.

2. Implementation Details:

- **Technology Stack:**
    - Language: Python (Version 3.11)
    - Web Framework: FastAPI
    - ASGI Server: Uvicorn (with hot-reloading enabled via `--reload`)
    - Database: PostgreSQL (Version 15, via official Docker image)
    - ORM & Migrations: SQLAlchemy (Version 2.x style), Alembic
    - Database Driver: psycopg2-binary
    - Data Validation/Serialization: Pydantic (Version 2.x)
    - Containerization: Docker, Docker Compose
    - Environment Variables: python-dotenv (used by Alembic config), direct environment variable injection via Docker Compose for `DATABASE_URL`.
- **API Design:**
    - Follows a RESTful approach. Key endpoints provided under the `/api/v1/` prefix:
        - `POST /roles`, `GET /roles`, `GET /roles/{role_id}`

        - `POST /permissions`, `GET /permissions`, `GET /permissions/{permission_id}`
        - `POST /roles/{role_id}/permissions`, `DELETE /roles/{role_id}/permissions/{permission_id}`
        - `POST /users/{user_id}/roles`, `GET /users/{user_id}/roles`, `DELETE /users/{user_id}/roles/{role_id}`
        - `POST /check` (Core access verification endpoint)
        - `GET /` (Basic health check)
    - Uses Pydantic schemas for request validation and response serialization (JSON).
    - Handles potential errors (e.g., duplicate names, item not found) using appropriate HTTP status codes (400, 404) and error details via FastAPI's `HTTPException`.
- **Data Storage:**
    - Uses a PostgreSQL database, managed within a Docker container via Docker Compose.
    - Data persistence is handled using a named Docker volume (`postgres_data`).
    - The database schema includes tables: `roles`, `permissions`, `role_permissions` (many-to-many), `user_roles` (many-to-many).
    - Schema is defined using SQLAlchemy ORM models (`app/models/rbac.py`).
    - Primary keys primarily use UUIDs for global uniqueness.
    - Database schema creation and updates are managed by Alembic migrations.

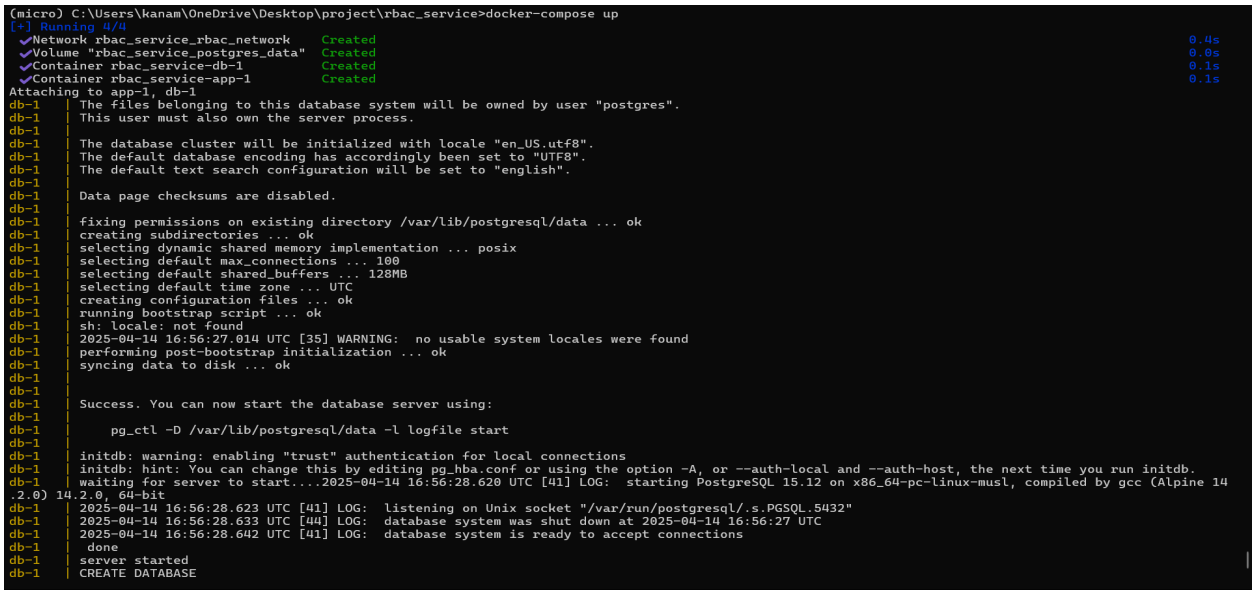
- **Core Logic:**
    - Database interactions are encapsulated in CRUD functions (`app/crud/rbac.py`).
    - The core permission check logic (`app/core/security.check_user_permission`) uses efficient SQLAlchemy queries involving subqueries and `exists()` to check relationships across association tables without loading unnecessary data.

3. Use of Docker:

- The RBAC microservice is fully containerized using Docker for consistency and portability.
-
- A `Dockerfile` defines the image:
    - Starts from a `python:3.11-slim` base image.
    - Sets up the working directory.
    - Copies `requirements.txt` and installs dependencies using `pip`.
    - Copies the entire project context (`.`) including application code (`app/`) and configuration files (`alembic.ini`) into the image.
    - Exposes port 8000.
    - Specifies the default command to run Uvicorn with `--reload` enabled for development.
- `docker-compose.yml` orchestrates the RBAC service (`app`) and the PostgreSQL database (`db`) service:
    - Builds the `app` image from the Dockerfile.
    - Mounts the local `./app` directory into the container for hot-reloading.

- Sets up a dedicated network (`rbac_network`) for inter-service communication.
- Configures the `db` service using the official `postgres:15-alpine` image, environment variables for setup, and a named volume for data persistence.
- Includes a `healthcheck` for the `db` service to ensure it's ready before the `app` service starts.
- Configures the `app` service to depend on the `db` service being healthy.
- Provides the `DATABASE_URL` (using the `db` service name) to the `app` container.
- *(Initial setup included automatic migration via `command:`, but final version relies on manual `docker-compose exec app alembic upgrade head` after startup for robustness during troubleshooting).*

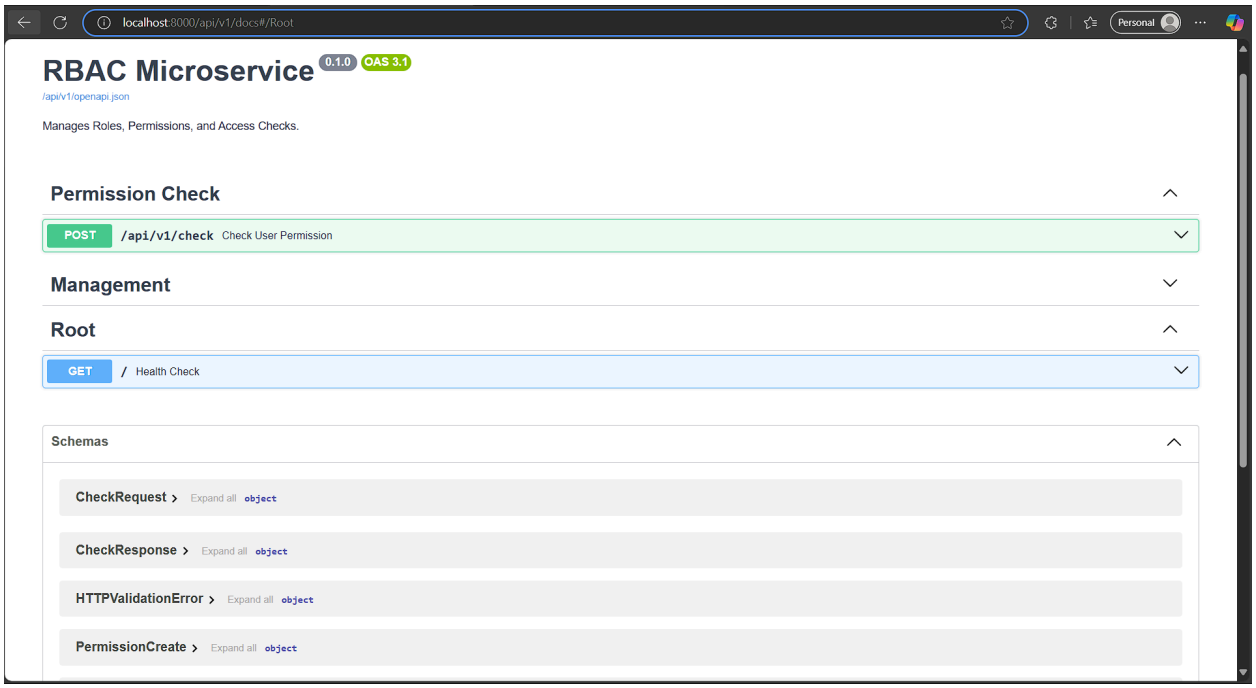
4. Screenshots:

- 

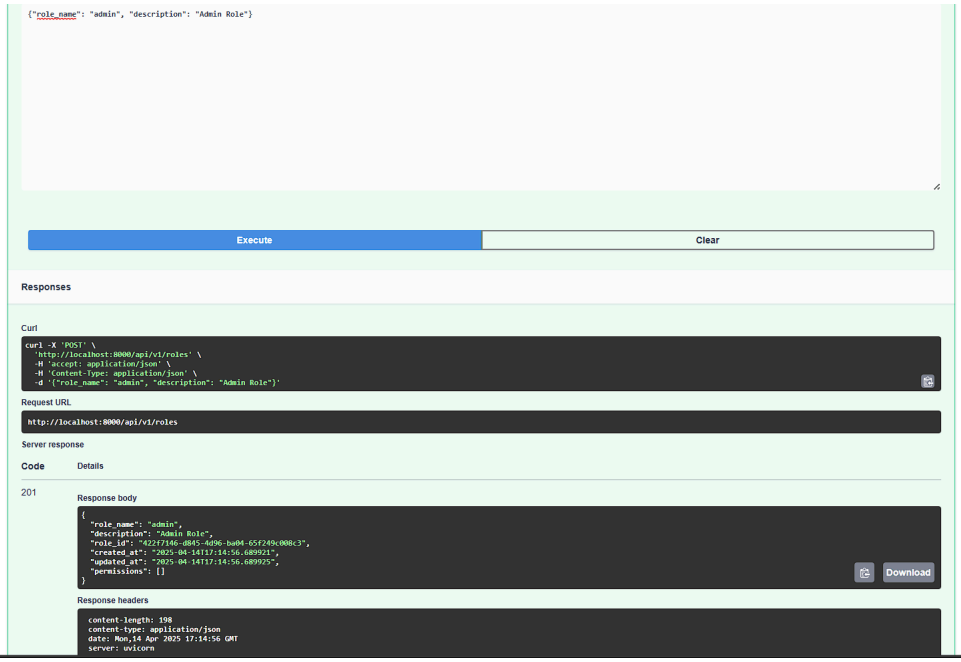
```
db-1    | CREATE DATABASE
db-1    |
db-1    |
db-1    | /usr/local/bin/docker-entrypoint.sh: ignoring /docker-entrypoint-initdb.d/*
db-1    |
db-1    | waiting for server to shut down....2025-04-14 16:56:28.856 UTC [41] LOG:  received fast shutdown request
db-1    | 2025-04-14 16:56:28.859 UTC [41] LOG:  aborting any active transactions
db-1    | 2025-04-14 16:56:28.864 UTC [41] LOG:  background worker "logical replication launcher" (PID 47) exited with exit code 1
db-1    | 2025-04-14 16:56:28.865 UTC [42] LOG:  shutting down
db-1    | 2025-04-14 16:56:28.868 UTC [42] LOG:  checkpoint starting: shutdown immediate
db-1    | 2025-04-14 16:56:29.049 UTC [42] LOG:  checkpoint complete: wrote 920 buffers (5.6%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.048 s, sy
nc=0.123 s, total=0.184 s; sync files=301, longest=0.003 s, average=0.001 s; distance=4233 kB, estimate=4233 kB
db-1    | 2025-04-14 16:56:29.065 UTC [41] LOG:  database system is shut down
db-1    |  done
db-1    | server stopped
db-1    |
db-1    | PostgreSQL init process complete; ready for start up.
db-1    |
db-1    | 2025-04-14 16:56:29.204 UTC [1] LOG:  starting PostgreSQL 15.12 on x86_64-pc-linux-musl, compiled by gcc (Alpine 14.2.0) 14.2.0, 64-bit
db-1    | 2025-04-14 16:56:29.204 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
db-1    | 2025-04-14 16:56:29.204 UTC [1] LOG:  listening on IPv6 address "::", port 5432
db-1    | 2025-04-14 16:56:29.210 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
db-1    | 2025-04-14 16:56:29.219 UTC [57] LOG:  database system was shut down at 2025-04-14 16:56:29 UTC
db-1    | 2025-04-14 16:56:29.229 UTC [1] LOG:  database system is ready to accept connections
app-1   | INFO:     Will watch for changes in these directories: ['/app']
app-1   | INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
app-1   | INFO:     Started reloader process [1] using WatchFiles
app-1   | INFO:     Started server process [8]
app-1   | INFO:     Waiting for application startup.
app-1   | INFO:     Application startup complete.

v View in Docker Desktop   o View Config   w Enable Watch
```

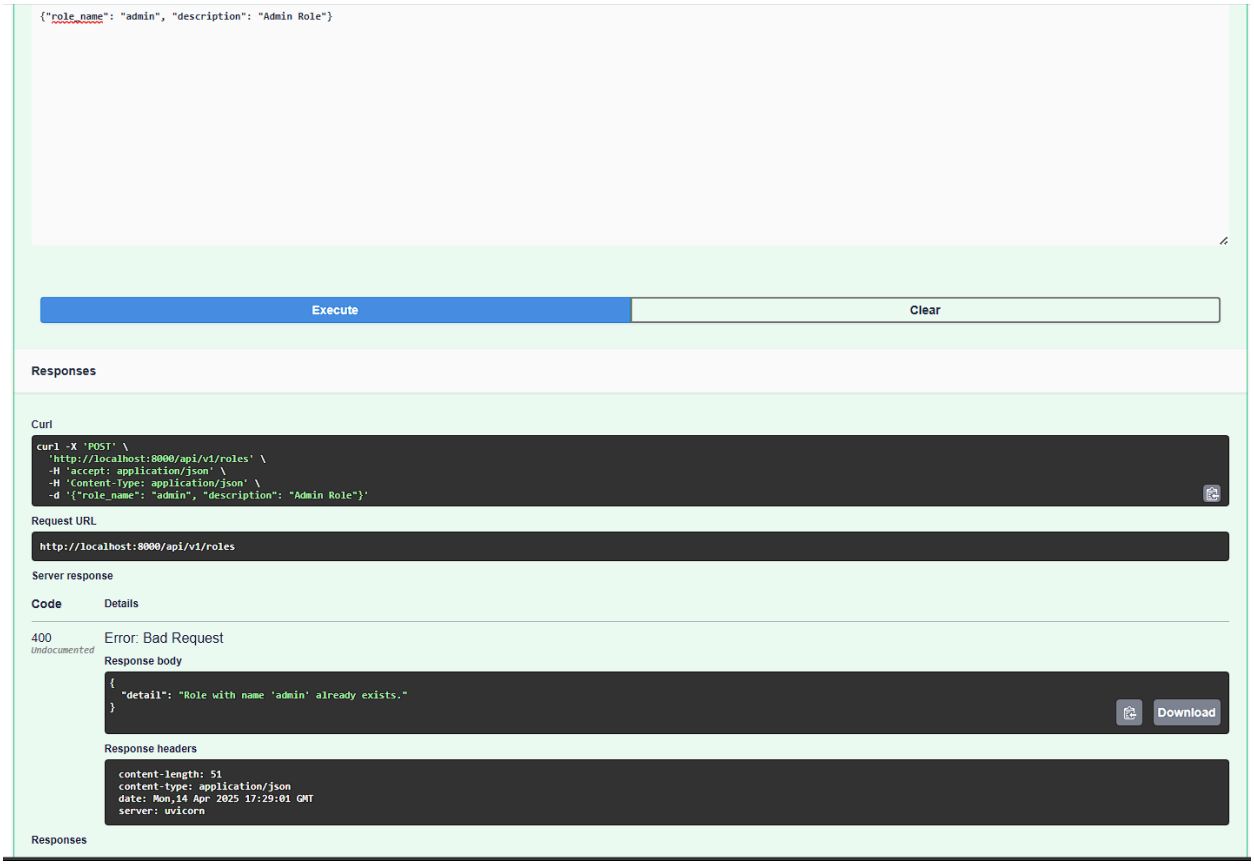
- RBAC Service startup via Docker Compose, showing successful database migration (or DB up-to-date) and Uvicorn server start.



- *Caption:* RBAC Service API Documentation (Swagger UI) at /api/v1/docs.

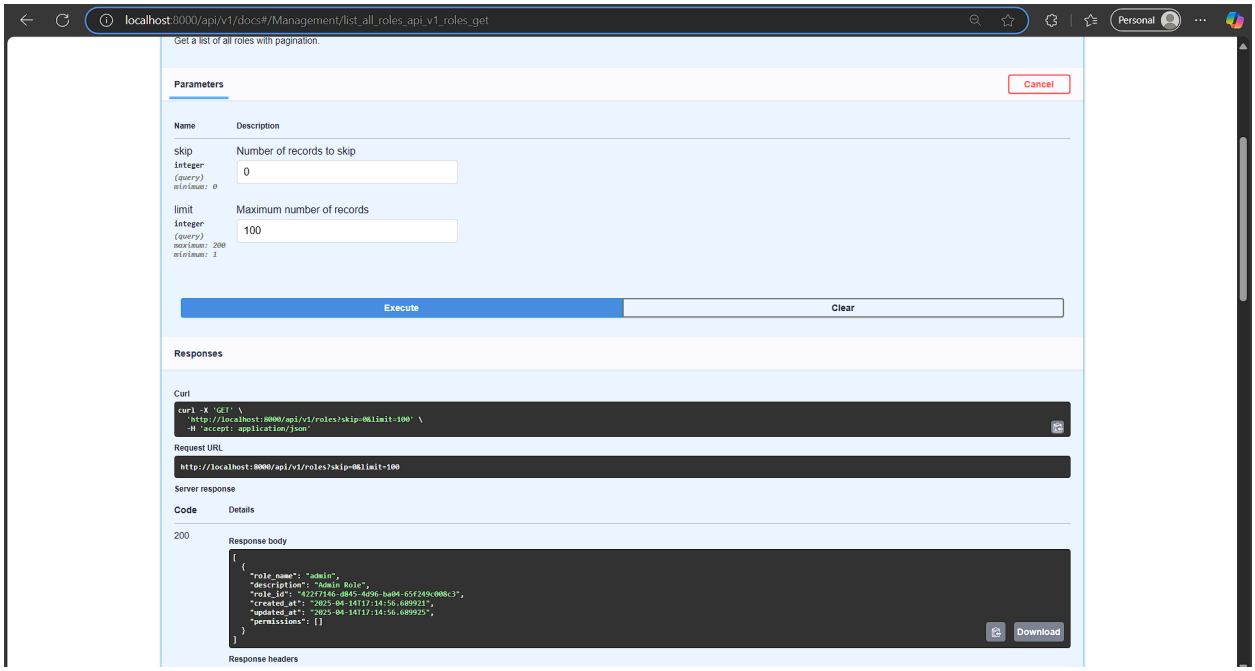


- *Caption:* Successful creation of 'admin' role via POST /api/v1/roles.

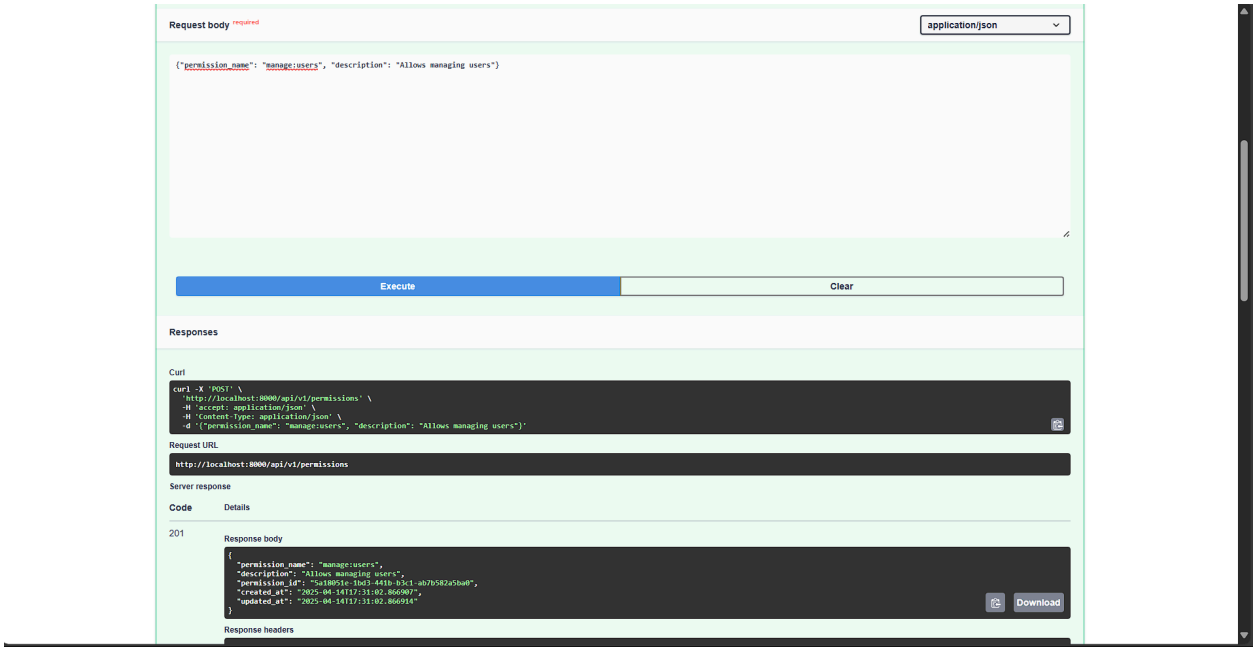


-

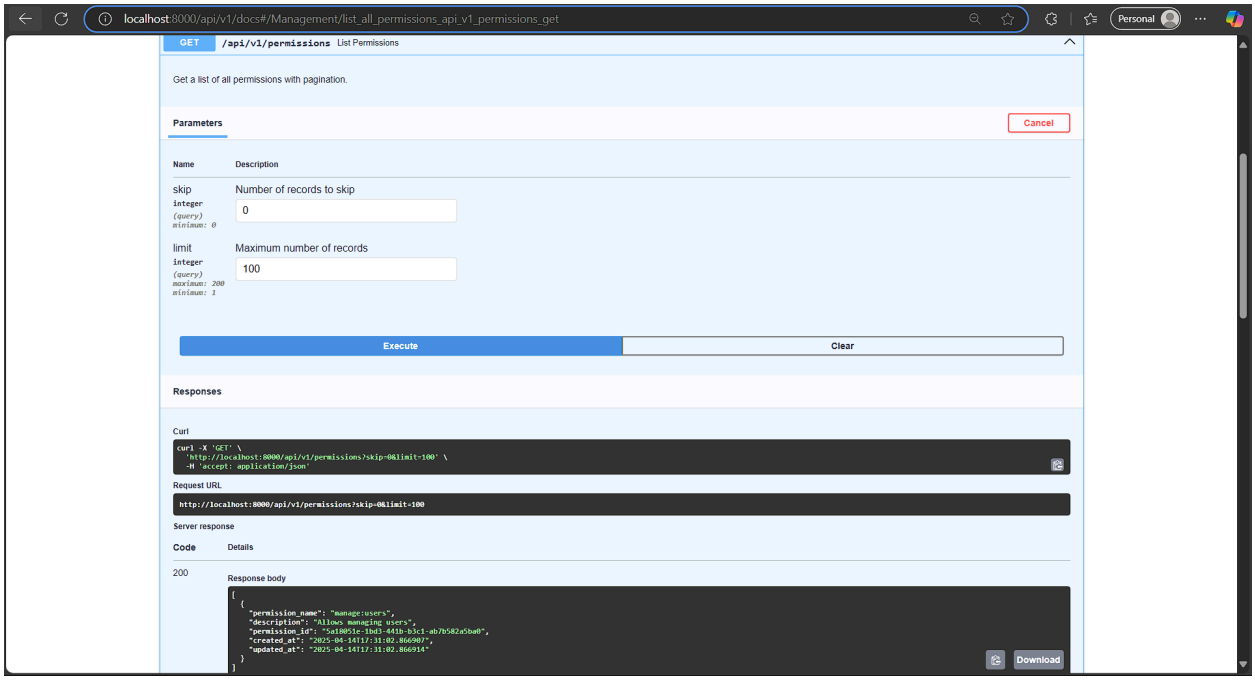
- *Caption:* Correct error response (400 Bad Request) when attempting to create a duplicate role name.

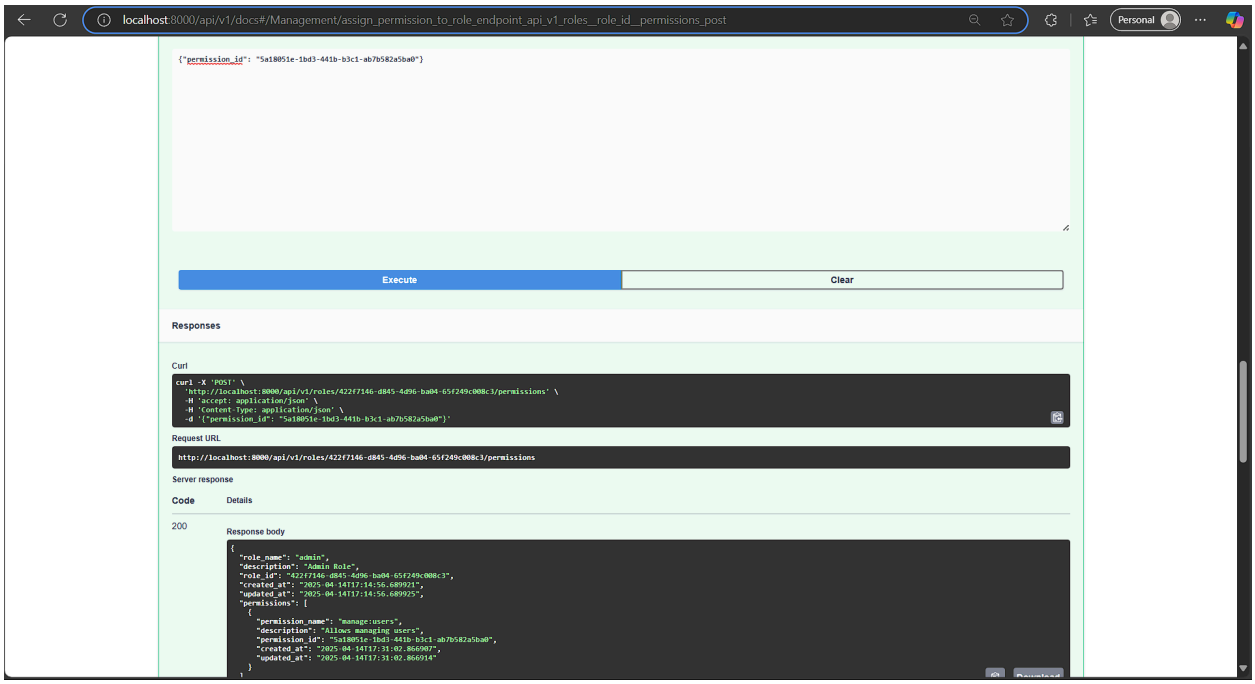


- *Caption:* Successful retrieval of created roles via GET /api/v1/roles.

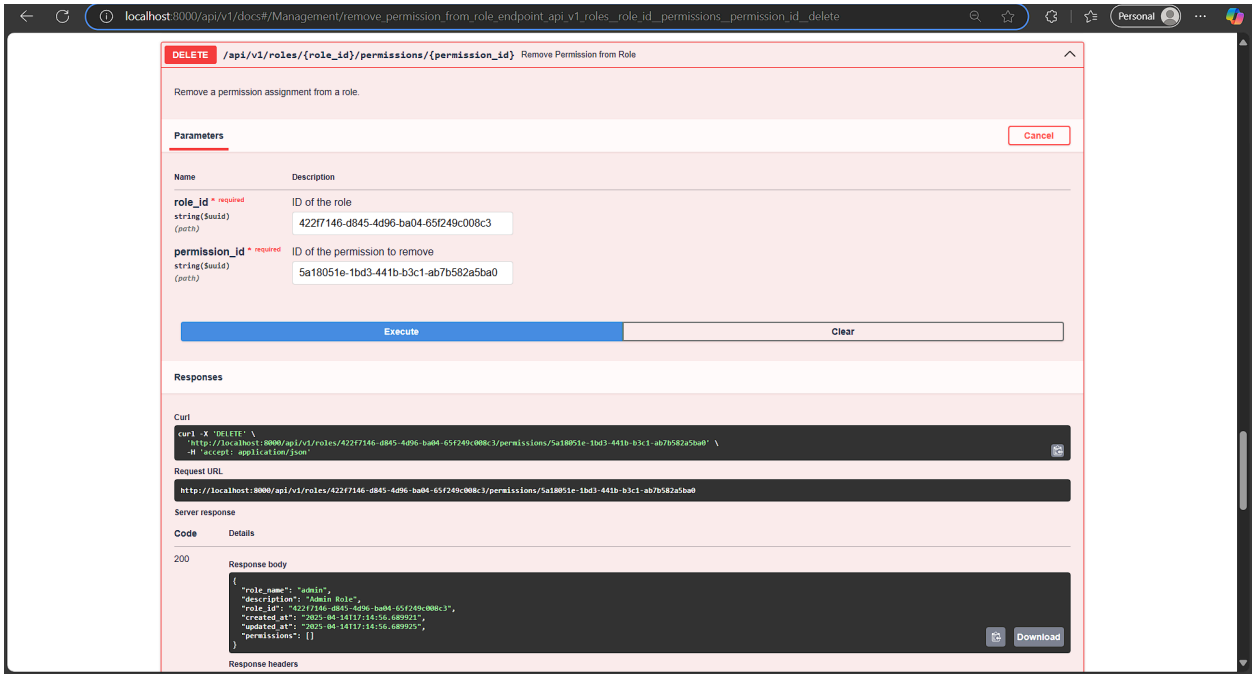


- *Caption:* Successful creation of 'manage:users' permission via POST /api/v1/permissions.

- 

  - *Caption:* Successful retrieval of created permissions via GET /api/v1/permissions.
- 

  - *Caption:* Successfully assigning 'manage:users' permission to the 'admin' role via POST /roles/{role_id}/permissions. Response shows updated role.



- *Caption:* Successfully removing 'manage:users' permission from the 'admin' role via DELETE /roles/{role_id}/permissions/{permission_id}. Response shows updated role.

Notification -Event Triggered , Push

1. Features of Notification Microservice (Implemented so far):

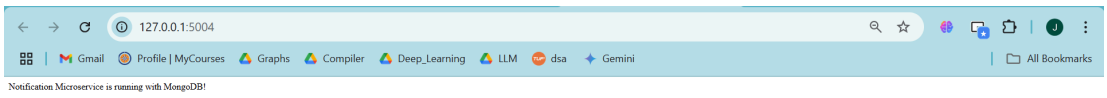
- **In-App Notification Creation:** The microservice provides the functionality to create and store in-app notifications for users. These notifications can be triggered by other events in the virtual lab platform.
- **Test Endpoint:** A temporary API endpoint (`/test-in-app-notification/<user_id>`) has been implemented to simulate the creation of an in-app notification for a specific user ID.

2. Implementation Idea:

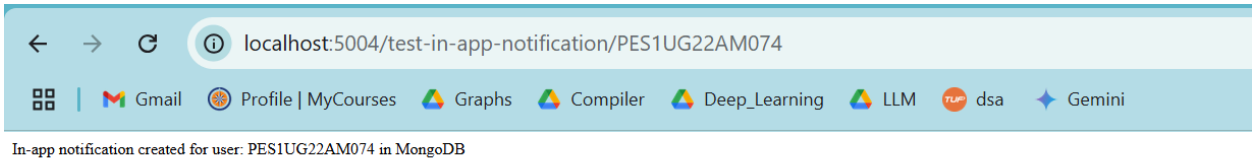
- Technology Stack: Python (Flask), `pymongo` for MongoDB interaction. Docker for containerization.
- API Design: For now, we have an internal test endpoint (`/test-in-app-notification/<user_id>`) to trigger notification creation. In the future, this service will subscribe to events from a message broker.
- Data Storage: MongoDB is used to store the in-app notifications. A database named `notification_db` and a collection named `in_app_notifications` are used. Each notification is stored as a document with fields like `user_id`, `notification_type`, `message`, `created_at`, and `is_read`.
- Dockerization: The microservice is containerized using Docker for consistent execution and simplified deployment.

3. Output of Implementation:

Accessing the `/test-in-app-notification/<user_id>` endpoint in a web browser (e.g., `http://localhost:5004/test-in-app-notification/PES1UG22AM074`) will return the message: `In-app notification created for user: PES1UG22AM074 in MongoDB`.



http://localhost:5004/test-in-app-notification/PES1UG22AM074



4.Use of Docker:

- Docker is used to containerize both the Notification microservice and the MongoDB database.
- The `Dockerfile` for the Notification microservice specifies the Python environment and dependencies needed to run the Flask application.
- MongoDB is run in a separate Docker container using the official `mongo` image.
- The Notification microservice container is linked to the MongoDB container, allowing it to connect to the database using the container name as the hostname (`notification-mongo`).

5.How to Integrate with Other Microservices:

Currently, the Notification microservice has an internal test endpoint. For event-triggered notifications, the next step would be to:

- Integrate with a Message Broker (e.g., RabbitMQ): The Notification microservice will subscribe to relevant events published by other microservices (like the Registration service when a new user registers, or a Lab Management service when a lab session starts).
- Implement Logic to Handle Events: When an event is received, the Notification service will determine the appropriate notification type (email, push, in-app) and the target user(s) based on the event data and user preferences.
- Use the `create_in_app_notification` function to store in-app notifications.

Source Code For each Microservices Have been provided in the google drive :