**CS673 Software Engineering** 

**Team 1 - Spring Alert**

**Software Design Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Jim Penner | Team Leader | *Jim Penner* | 05/22/2024 |
| Shenxiao (Sherry) Li | Requirement Leader | *Sherry Li* | 05/27/2024 |
| Chaozheng Zhu | Configuration Leader | *Chaozheng zhu* | 05/28/2024 |
| Jordany Arnaud | Design & Implementation & Security leader | *Jordany A.* | 05/27/2024 |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **1** | **Team 1:Spring\_Alert** | **06/10/2024** | **v1.0** |
| **2** | **Team 1:Spring\_Alert** | **06/18/2024** | **v1.1** |

[Introduction](#_heading=h.gjdgxs)

[Software Architecture](#_heading=h.30j0zll)

[Class Diagram](#_heading=h.1fob9te)

[UI Design (if applicable)](#_heading=h.3znysh7)

[Database Design (if applicable)](#_heading=h.2et92p0)

[Security Design](#_heading=h.tyjcwt)

[Business Logic and/or Key Algorithms](#_heading=h.3dy6vkm)

[Design Patterns](#_heading=h.1t3h5sf)

[Any Additional Topics you would like to include.](#_heading=h.4d34og8)

[References](#_heading=h.2s8eyo1)

[Glossary](#_heading=h.17dp8vu)

# Introduction

This document outlines the design for the implementation of a user management service and a notification service. The user management service is responsible for handling user-related operations, such as user registration, user profile management, and group management. The notification service handles sending notifications to users based on various triggers and events within the system.

The document focuses on the following aspects:

* Design architecture
* Class diagrams
* Database schema
* Security and authorization mechanisms

This document provides a detailed description of how OAuth 2.0 and Auth0 are used to secure backend APIs, ensuring proper access control through token validation and permission-based access.

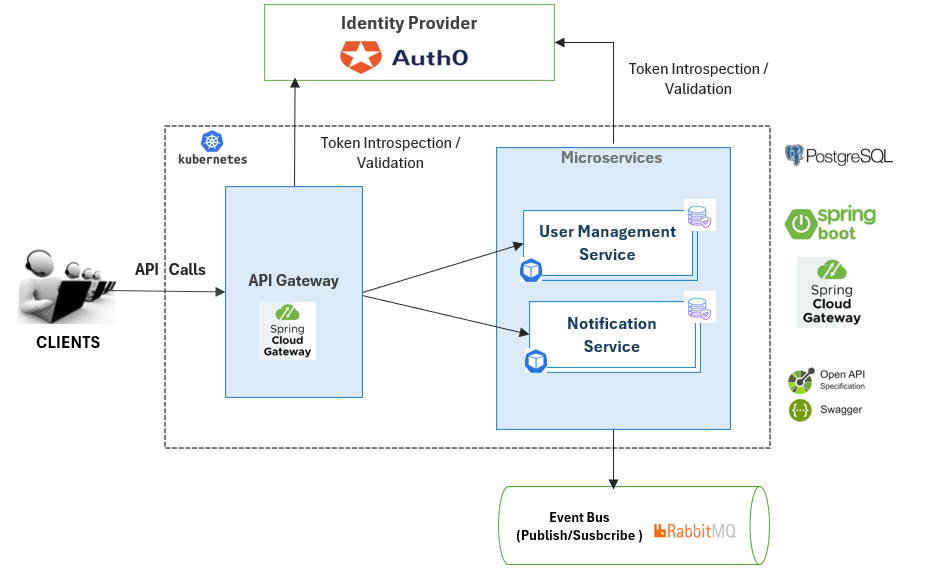
# Software Architecture

This architecture utilizes microservices for user management and notifications, leveraging Auth0 for authentication. In this section, we cover:

* Components & High-Level Architecture
* Package Structure
* Exception & Error Handling
* Framework and Tools
* API Specification Via Swagger documentation

Components & High-Level Architecture

* *API Gateway*: Receives all requests and routes them based on the path. It will be implemented using Spring Cloud Gateway.
* *User Management Service*: Handles user registration, profile management, and group creation.
* *Notification Service*: Manages notifications for users and groups.
* *RabbitMQ*: Used for publishing notification "Send Request".
* *PostgreSQL*: Serves as the database system.
* *Auth0*: Provides user authentication and authorization services.

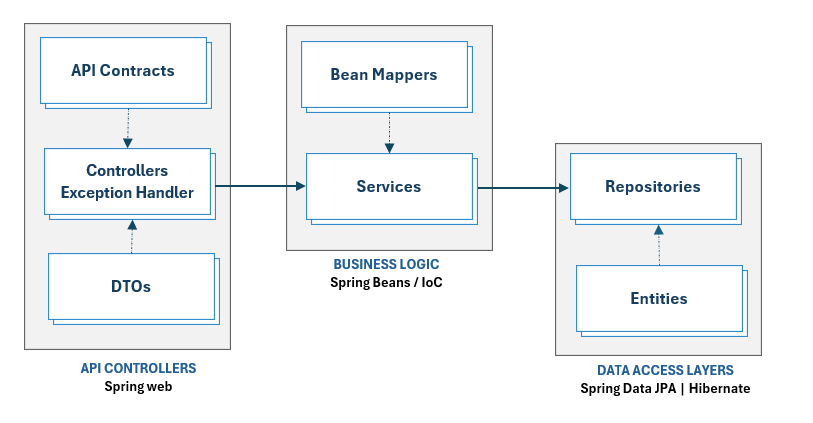


Service Naming Convention & Package structure

The standard naming convention for the services is “metscs673-<name>-service” and the basic package structure is:

* “edu.bu.metcs673.<service\_name>.api.models”: Data Transfer Objects classes.
* “edu.bu.metcs673.<service\_name>.api.controllers”: API Controllers.
* “edu.bu.metcs673.<service\_name>.api.exception.handler”: Global handler.
* “edu.bu.metcs673.<service\_name>.api.configs”: Configuration classes
* “edu.bu.metcs673.<service\_name>.entities”: Entity classes.
* “edu.bu.metcs673.<service\_name>.repositories”: Repository interfaces.
* “edu.bu.metcs673.<service\_name>.services”: Service interfaces.
* “edu.bu.metcs673.<service\_name>.services.impl”:Service Impl.

Overview of the main class components.



Exception & Error handling

The service interfaces will be responsible for catching possible code exceptions using try-catch blocks and converting those exceptions into two possible types of runtime exceptions:

* *Specific Exception Handling*: Specific exceptions will be defined for different error scenarios (e.g., UserNotFoundException).
* *Spring Boot Global Exception Handler*: A global exception handler will be used to capture unhandled exceptions that propagate through the service layer. Custom Exceptions: Specific error messages will be extracted from custom exceptions to provide informative responses to the client.

*Unforeseen Exceptions*: These exceptions will be logged for debugging purposes, and generic error messages will be returned to the client, indicating an internal server error.

*User Management Service*

| **Exception** | **When/ Description** |
| --- | --- |
| InvalidUserAccountException | when an invalid user account is encountered, or doesn’t exist in Auth0. |
| UserProfileAlreadyExistsException | when attempting to register a user that already exists. |
| ResourceNotFoundException | When a the requested resource is not found |

*Notification Service*

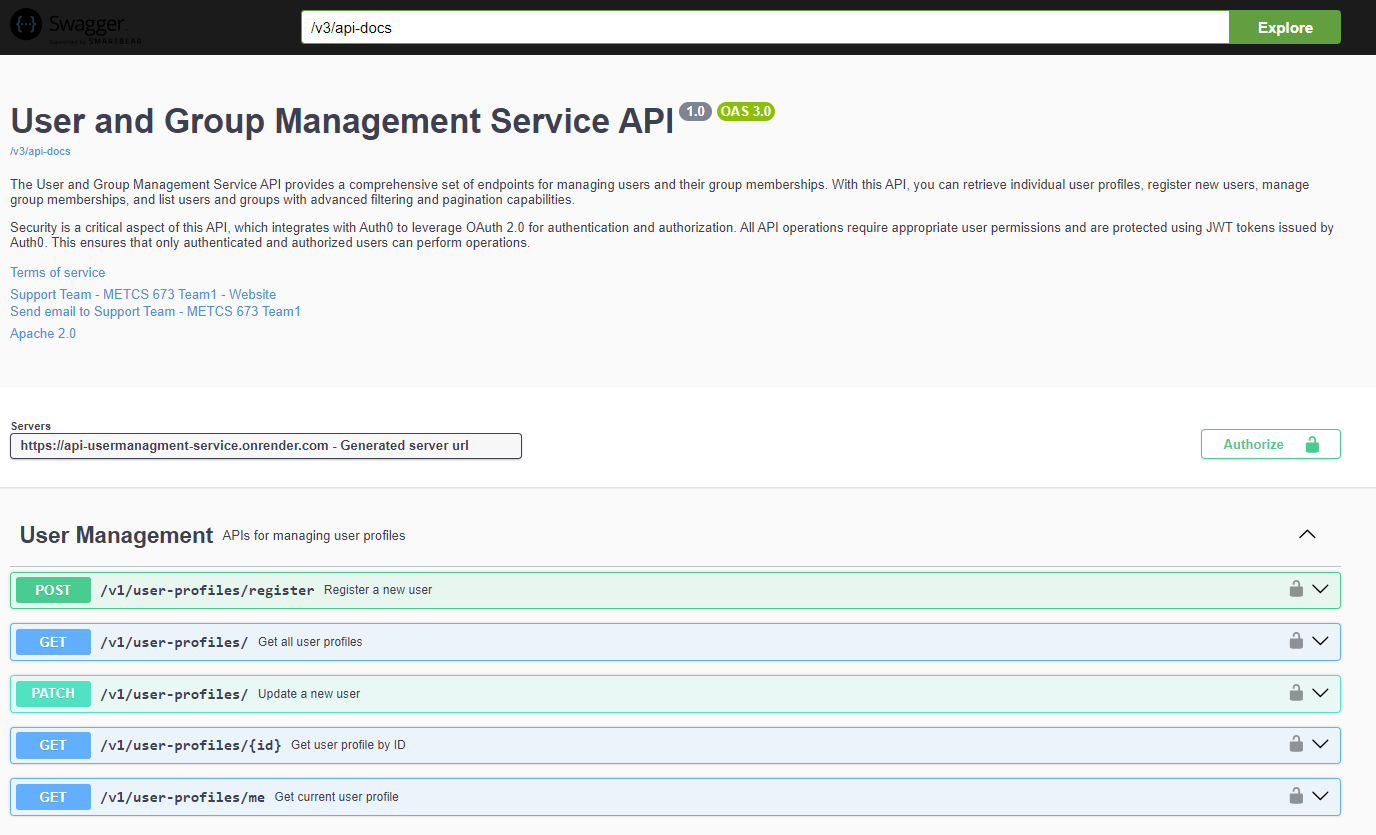
| **Exception** | **When** |
| --- | --- |
| UserNotFoundException | When trying to retrieve information about a user that doesn’t exist in the database |
| Custom exemption | If for instance an observer array is empty, you may simply run a graceful, custom exception.  It may be a system.out print that says: No observers registered, let’s say if all observers were removed. |
|  |  |

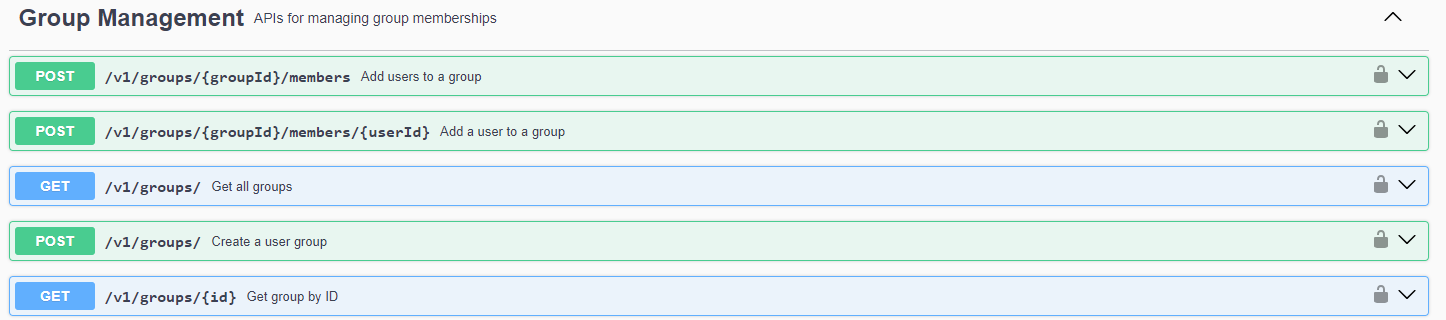
Frameworks used

Spring/Spring Boot is the primary framework that will be used to implement the microservices. Hibernate/JPA will also be utilized for the data access layer.

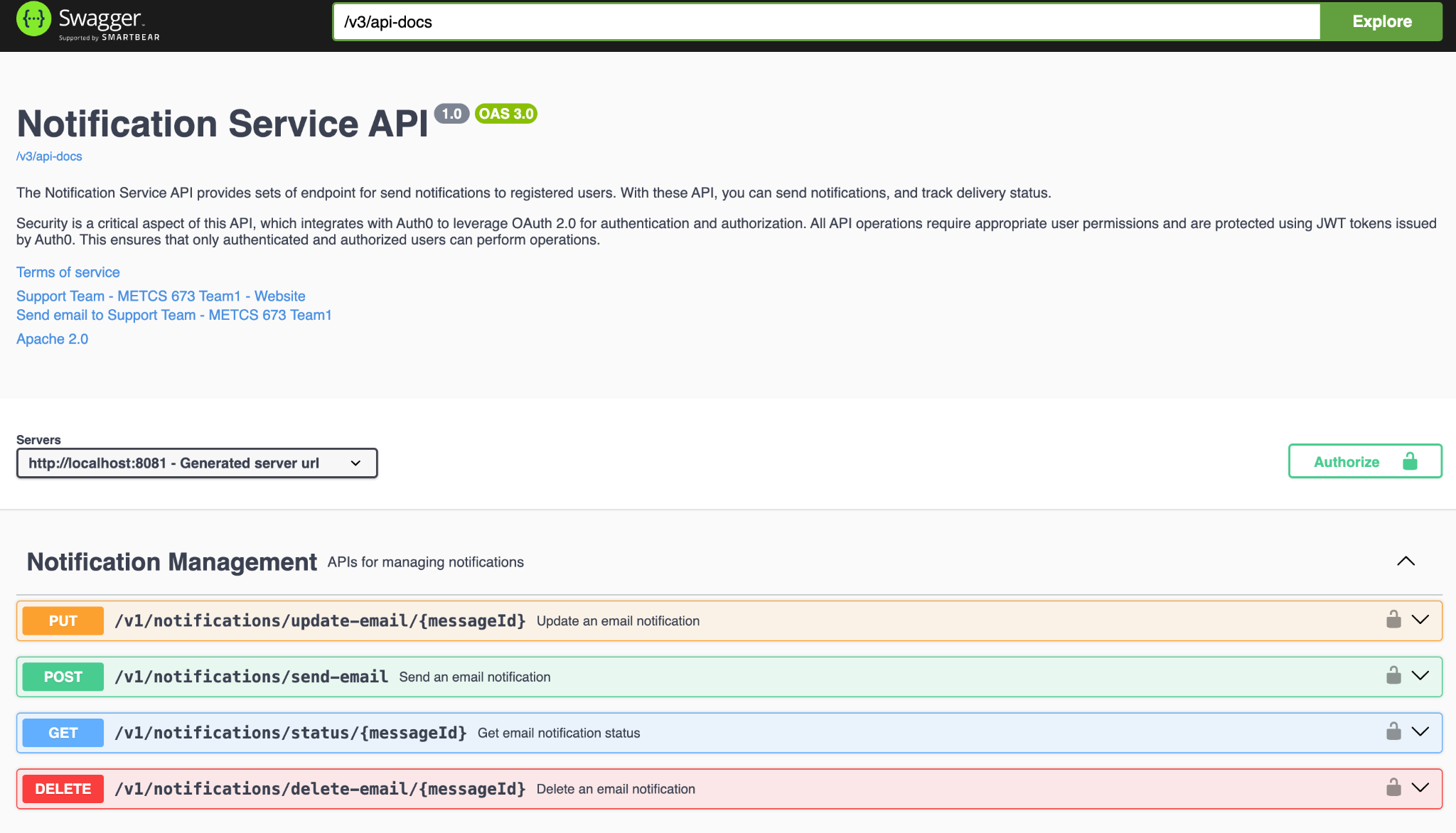
API Specification Via Swagger documentation

User management service - [Swagger Documentation](https://api-usermanagment-service.onrender.com/swagger-ui/index.html#/)

OpenAPI/Swagger is employed for designing the API specifications and providing documentation [here](https://api-usermanagment-service.onrender.com/swagger-ui/index.html#/)

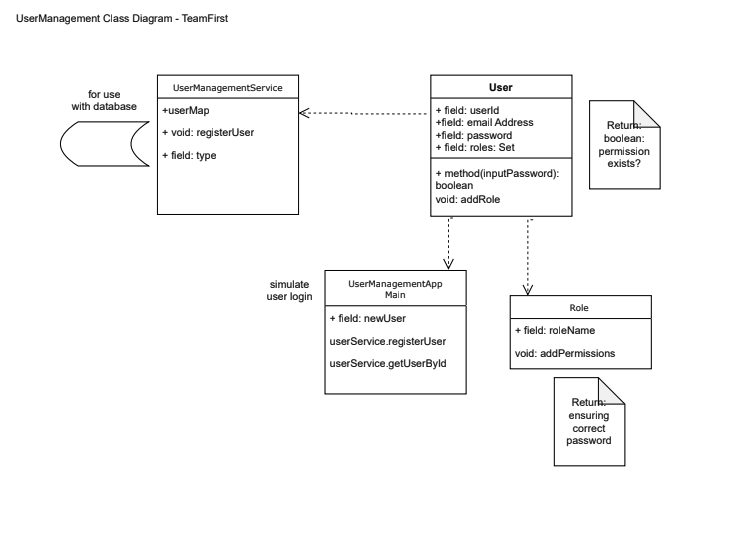


Notification Service - Swagger documentation

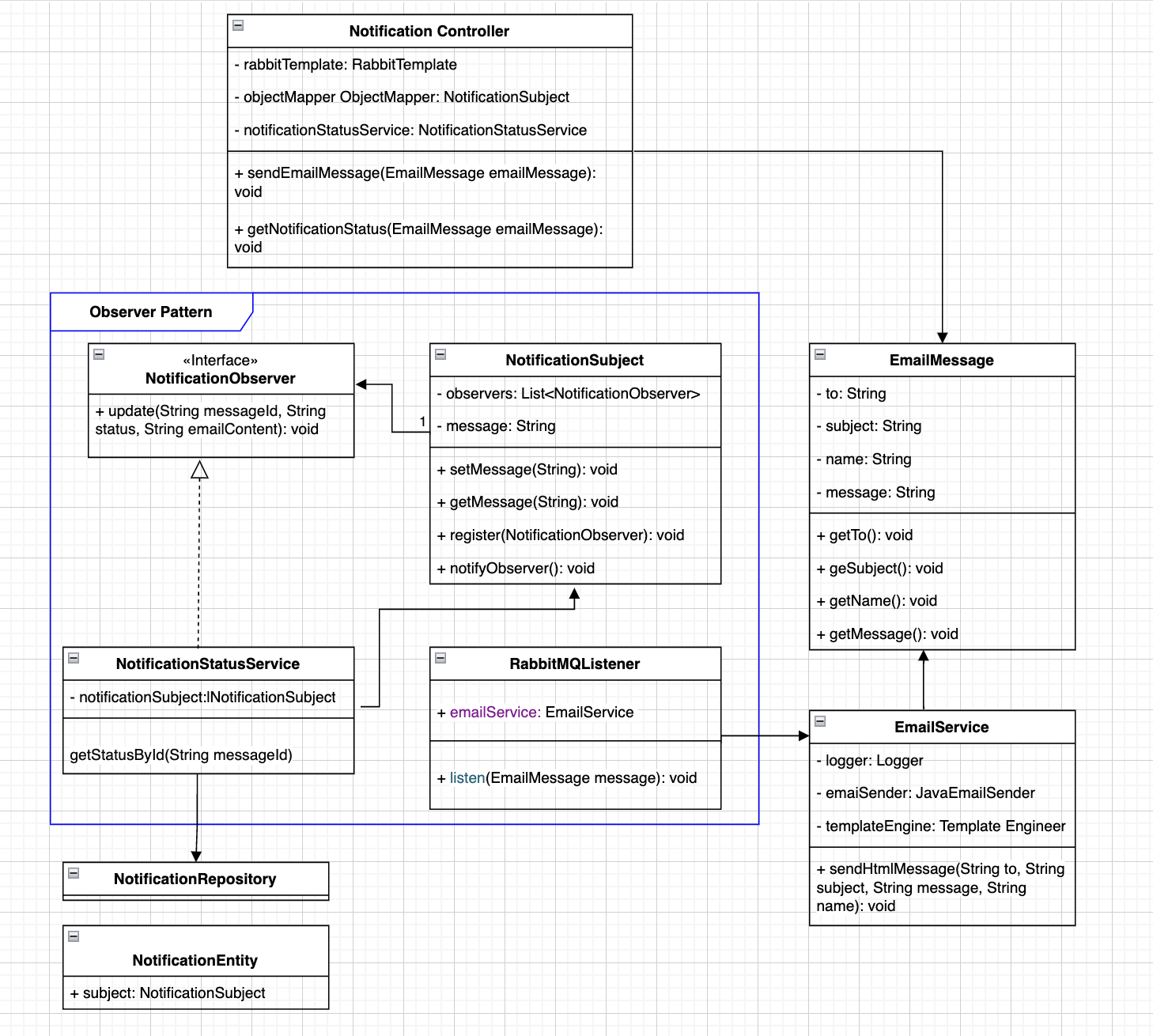


# Class Diagram

In this section, you will provide a detailed description of each component (or package) and use one or multiple class diagrams to show the main classes and their relationships in each component.

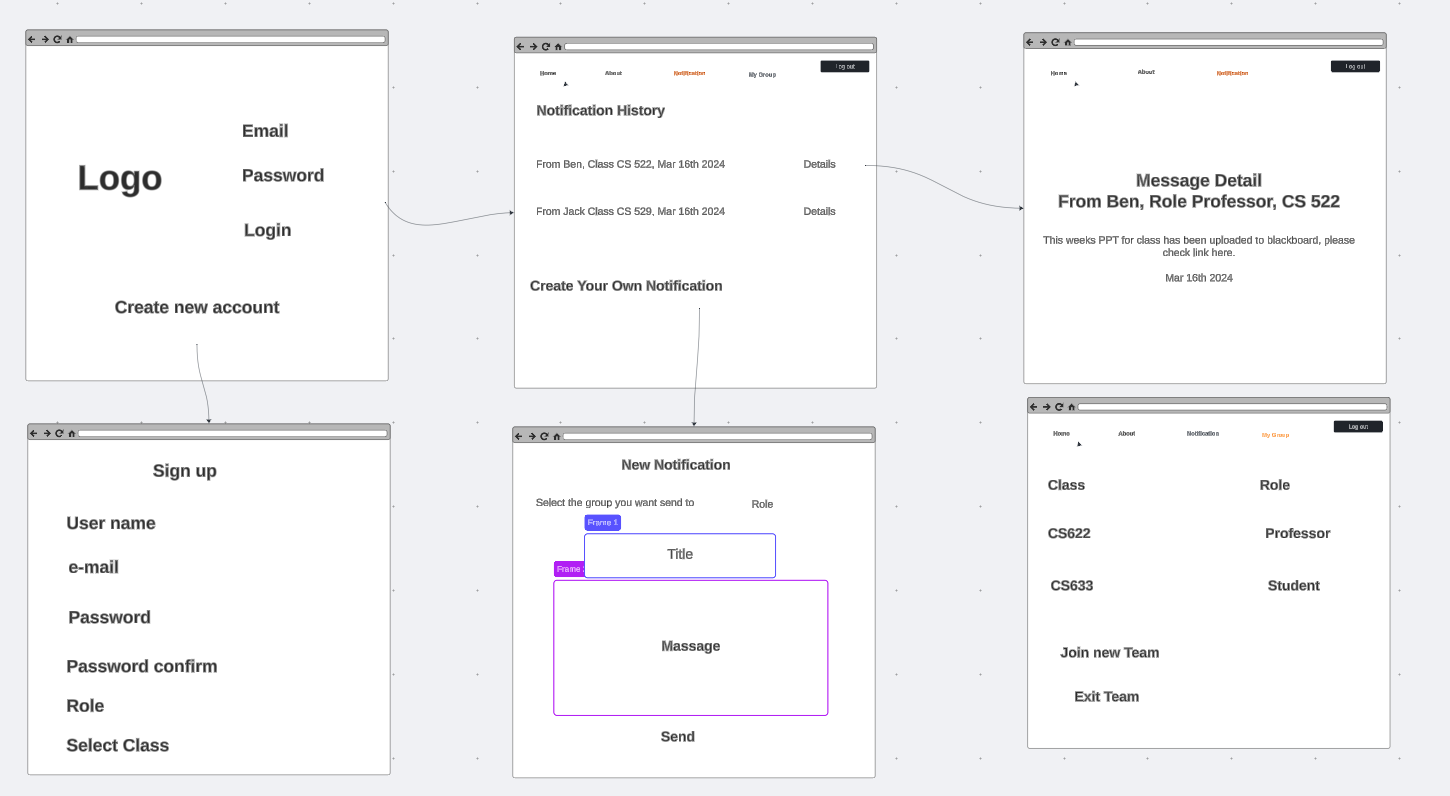
User management service 

Notification service



# UI Design (if applicable)

As we are implementing backend APIs, the UI is not applicable to this project. However, we have provided wireframes for the Login and Sign Up processes as part of our user management service.

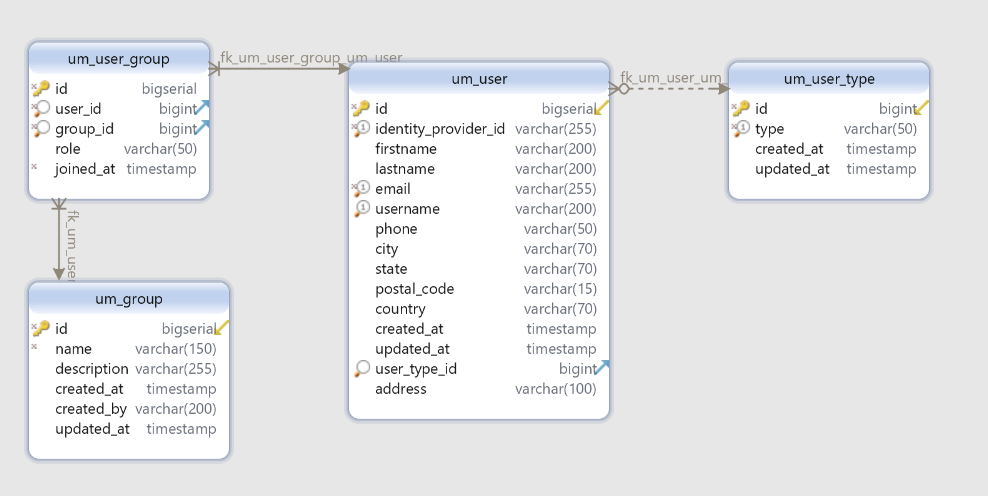


# Database Design

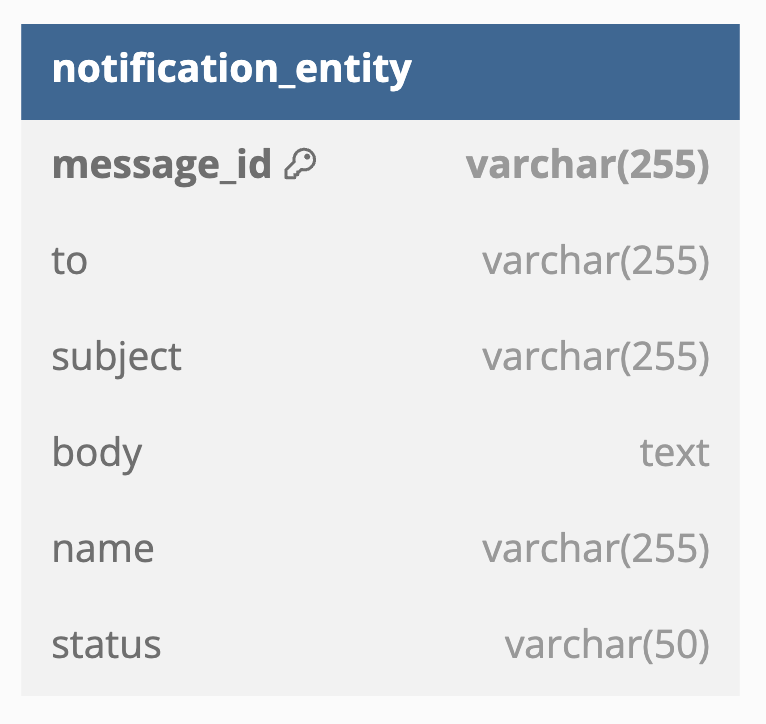
We have designed two distinct database schemas: one dedicated to the notification service and the other to the user management service. This design aligns with the microservice principle, which advocates for each microservice having its own database or schema.

Our database schemas are relational in nature, and we have opted to utilize PostgreSQL as our database management system. We operate PostgreSQL as an online database service hosted on CockroachDB, which offers scalability and resilience suited to our microservices architecture.

*User Management Schema*



*Notification Service Schema*

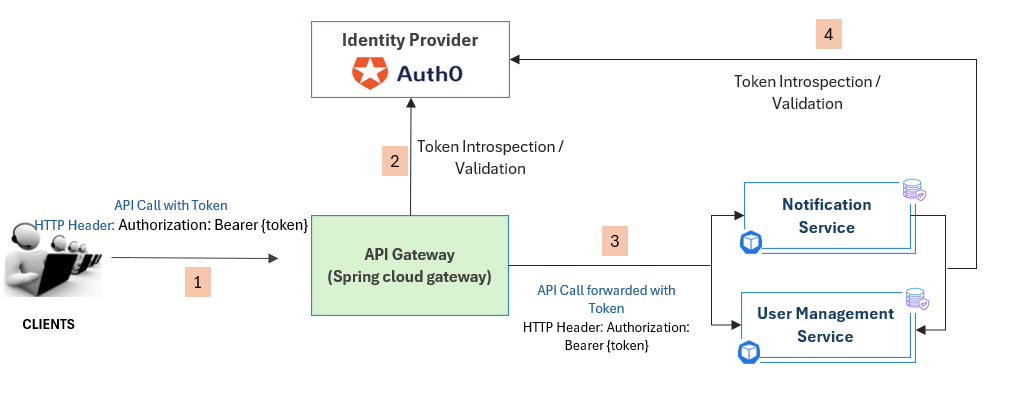


# Security Design

OAuth 2.0 will be used to secure the backend APIs, with Auth0.com serving as the authorization server. Auth0 provides a flexible, drop-in solution for adding authentication and authorization services to applications.

We have chosen to implement permission-based access control, where each resource is associated with a specific permission. These permissions will be configured in Auth0 for each microservice. When a user requests a token from Auth0 to access the backend API, the token will include the relevant permissions and roles. This information will be used to verify whether the client has the appropriate authorization to access the requested resource.

Upon registration, the user, based on the type, will be granted default access for accessing the resources.



The API client could be a Single Page Application (React Native) or another microservice. In the first case, the SPA will use the Authorization Code Flow with Proof Key for Code Exchange (PKCE) to obtain the access token as described [here](https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce). In the second case, the microservice will use the Client Credentials Flow to obtain the token and access the APIs as described [here](https://auth0.com/docs/get-started/authentication-and-authorization-flow/client-credentials-flow).

1. The client obtains the token using the appropriate flow from Auth0, the authorization server.
2. The client calls the APIs with the access token set in the HTTP header.
3. The API gateway validates the token by calling Auth0 to ensure it is legitimate and checks the caller's permissions to verify access to the requested resource. If the token is valid and the caller is authorized, the API gateway forwards the request to the appropriate backend service.
4. The backend service checks if the request came through the API gateway. If the request did not pass through the API gateway, the service independently validates the token using Auth0.

*Permissions* : Below table provides the different permissions for each microservice.

*User Management Service*

| **Permission** | **Description** | **Resource** | **User type: DefaultAccess** |
| --- | --- | --- | --- |
| create:register\_profile | Register a new user | POST v1/user-profiles/register | Anyone |
| create: update\_profile | Update one’s profile | POST v1/user-profiles/{id} | Any user |
| read: view\_profile | View one’s profile | GET v1/user-profiles/me | Any user |
| create: create\_group | Create group | POST v1/groups | Instructor/Administrator |
| create: manage\_members | Add members to group | POST v1/groups/{groupId}/members{userId} - POST v1/groups/{id}/members  PATCH v1/groups/{groupId}/members{userId} | Instructor/Administrator,  Group’s owner |
| read: view\_goup | View one’s group | GET v1/groups - v1/groups/{id} | Student/ Instructor/Administrator |
| delete: delete\_group | View group members | DELETE v1/groups/{id}/ | Student/ Instructor/Administrator |

*Notification Service*

| **Permission** | **Description** | **Resource** | **User type: DefaultAccess** |
| --- | --- | --- | --- |
| create:notifications | create notification, fill in placeholder parts for notification body | POST /v1/notifications/send-email | Instructor/Administrator |
| read: delivery\_status | Retrieves the status of a specific email notification using the messageId. This can include information about whether the email was sent, delivered, opened, etc | GET /v1/notifications/status/{messageId} | Instructor/Administrator |
| update: notifications | Updates an existing email notification identified by the messageId. This can include changes to the recipient, subject, body, or other metadata | PUT /v1/notifications/update-email/{messageId} | Instructor/Administrator |
| delete: notifications | Deletes an email notification identified by the messageId. This removes the notification from the system | DELETE /v1/notifications/delete-email/{messageId} | Instructor/Administrator |

# Business Logic and/or Key Algorithms

In this section, you shall describe any key algorithms used in your software system, either in terms of pseudocode or flowchart, or sequence diagrams.

# Design Patterns

At this stage, we decide to apply the observer pattern and template pattern in notification service.

**Observer pattern:**

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

*Subjects and Observers*:

* The Subject in this case is the notification service.
* The Observers are the email clients or any other components that need to be notified when an email is sent(group members or related party)

*Implementation*:

* The notification service maintains a list of observers (group members).
* When a new notification (email) needs to be broadcasted, the service notifies all registered observers(e.g. group members).

**Template pattern:**

Notification templates are introduced to avoid building every notification from scratch. Also, a notification template allows for unique notification by customizing parameters, styling etc.

Template Method pattern is an excellent choice due to its ability to define the overall structure of the email creation process while allowing specific details to be customized by subclasses.

Going forward, we may apply more design patterns e.g. strategy patterns as we discuss.

# References

* Spring guidance via: <https://start.spring.io/>
* Docker guidance via: <https://www.docker.com/support/>
* RabbitMQ guidance via: youtube and <https://discord.com/channels/1092487794984755311/1092487794984755314>

# Glossary

# UML - Unified Modelling Language,

* API: Application Programming Interface
* HTTP: Hypertext Transfer Protocol
* UI: User Interface