SOA Generation System – Architecture Document

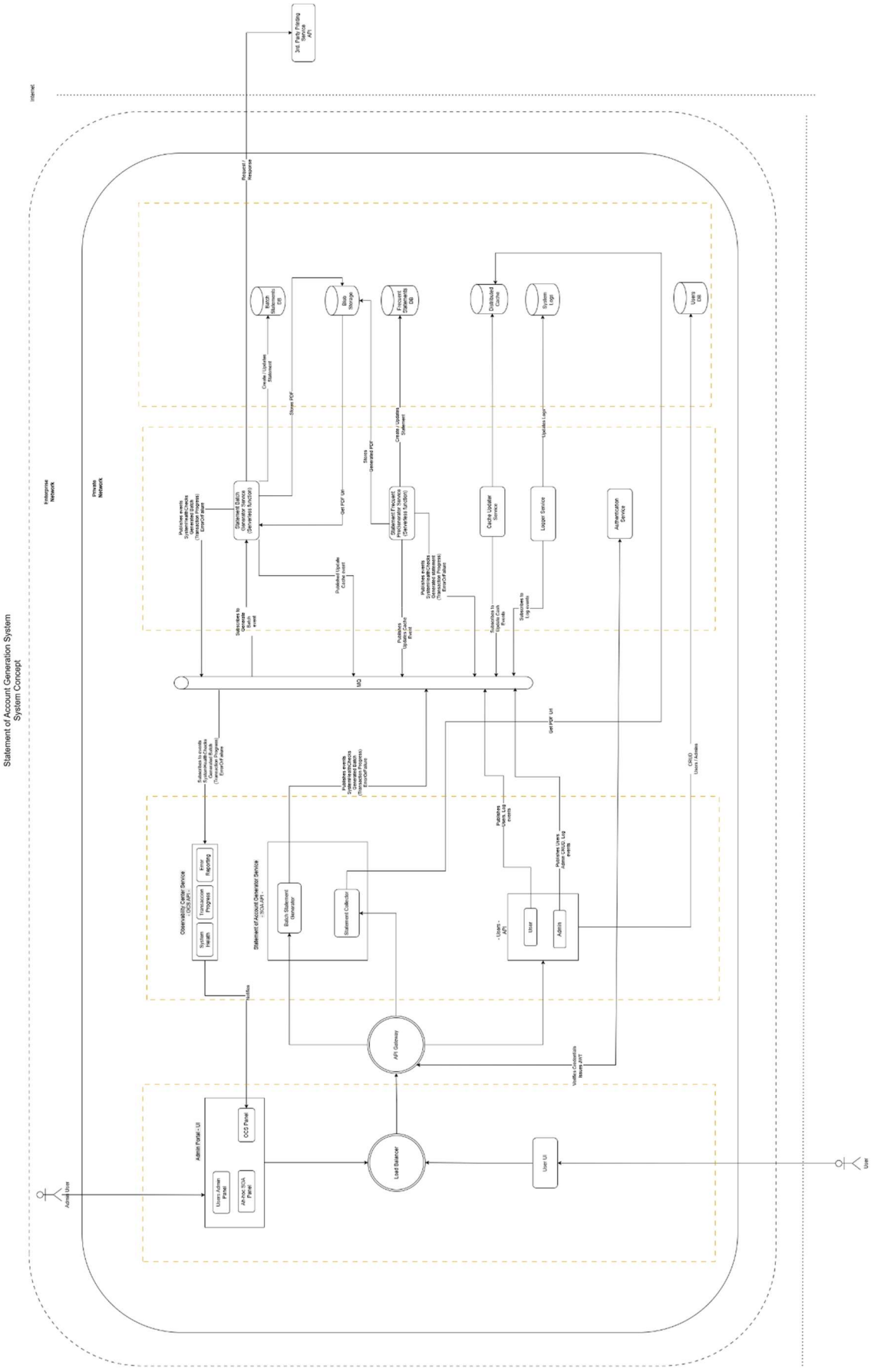
Table of Contents

1. System Design

1.1 Overview

The **Statement of Account (SOA) Generation System** automates the generation and distribution of financial statements for a user base of up to **700,000 users**. It supports scheduled and ad-hoc generation, real-time monitoring, offline batch printing, and secure cloud storage of statement data.

(For better resolution get the attached SOA-ConceptualDiagram.png file)


Statement of Account Generation System
System Concept
Internet
Enterprise Network
Private Network
3rd Party Payment API
Request / Response
Batch Statements DB
Blob Storage
Frecuent Statements DB
Distributed Cache
System Logs
Users DB
Create / Update Statement
Store PDF
Store Generated PDF
Get PDF Url
Statement Batch Generator Service (Quartz Job)
Statement Frecuent Requested Generator Service (Serverless function)
Cache Updater Service
Logger Service
Authentication Service
Publishes events SystemHealthCheck (Transactional Process)
Error/Failure
Subscribes to Generate Batch event
Publishes Update Cache event
Publishes events SystemHealthCheck (Transactional Process)
Error/Failure
Subscribes to Update Cache Events
Subscribes to Log events
Update Logs
MQ
Get PDF Url
CRUD Users / Admins
Subscribes to events SystemHealthCheck (Transactional Process)
Error/Failure
Publishes events Statement Process Generated Batch (Transactional Process)
Error/Failure
Publishes Users Log events
Publishes Users Admins Log events
Observability Center Service (OCS API)
System Health
Transaction Progress
Error Reporting
Statement of Account Generator Service (SOA API)
Batch Statement Generator
Statement Collector
Users - API
User
Admin
Traffic
API Gateway
Traffic Controller Main API
Admin Portal - UI
Users Admin Panel
API for SOA Rule
OCS Panel
Load Balancer
User UI
Admin Users
User

1.2 Per Service Application Architecture Layers

Asynchronous Microservices with an application Architecture in Layers and Dependency Injection Throughout, following SOLID principles and Design Patterns like Repository, Factory, UnitOfWork ans so forth. Unit testing and Integration testing is part of the development cycle.

Layers

- **Domain Layer**: Business rules, entities and ValueObjects.
- **Application Layer**: CQRS pattern with MediatR; handles coordination, validation, and transformation
- **Infrastructure Layer**: PostgreSQL access, Redis cache, external APIs (e.g., printing services). Repository patters and Unit of Work Pattern.
- **API Layer**: Exposes RESTful endpoints via Rest APIs
- **UI Layer**: Admin and monitoring portals.
- **Testing**: Unit testing (Moq, NUnit, EFCore.InMemory), integration testing (WebApplicationFactory, FluentAssertions)

1.3 Core Services

- **APIs**: Handle for Observability, Statement Generation and Users Management.
- **Message Broker:** The services communicate with each other via events, that are transported in a message broker with a certain topic in it. Events will be like: UserRegistered, UserCreatedByAdmin, UserDeleted, UserUpdated, UserDeactivated
- **Observability Center Service (OCS)**: Monitors system metrics, health, and errors
- **Admin Portal UI**: Enables ad-hoc statement generation and real-time observability, as well as handling Users.
- **User UI**: Used by customers who can register a user for themselves.
- **Statement Batch Generator Service**: A serverless function that works as a background job, which creates and stores big files containing several statements.
- **Statement Frequent PreGenerated Service**: A serverless function that works as a background job, which creates and stores individual files containing statements.
- **Cache Udater Service**: Intended to have a Cache in Sync.
- **Logger Service**: It will maintain a log in a regular basis.
- **The Authentication Service**: It seats close to the API Gateway. The plan is as follows: [Client] —▶ [API Gateway] —▶ [Authentication Service] —▶ [JWT Issued] —▶ [Forward to downstream APIs]

1.4 Offline Support

Approximately 5% of clients receive printed statements. Offline flow includes. We will have a flow were the pdfs are produced asynchronously and set ready via Cache and also stored as registries in DB and Blob in the Cloud. This will allow the users to query on the Cache first and if the statement exists, then they will have it, otherwise, it might need to be produced manually.

1.5 Database Design

Database is implemented in **PostgreSQL**, using a multi-tenant design. More on this in the PoC Section.

---

2. Security Architecture

2.1 Authentication

- **JWT tokens** are issued after login and include claims like UserId, TenantId, and Role
- [Authorize(Roles = "Admin")] is used to secure admin endpoints
- Plans to add **OAuth 2.0** support.

2.2 Authorization

- **Role-Based Access Control (RBAC)** defines permissions

- Future: Identity Provider Integration

2.3 Data Protection

- **Encryption** in transit (HTTPS) and at rest (PostgreSQL and Redis)
- JWTs are signed and securely stored on the client
- Redis caches only non-sensitive statement metadata with strict TTL policies

2.4 Transport Security (SSL/TLS)

All communication across the system—both internal and external—is secured using Transport Layer Security (TLS), ensuring the confidentiality and integrity of data in transit.

HTTPS Enforcement

- All public-facing endpoints (e.g., SOA.UsersAPI, Admin Portal, and Auth Service) are only accessible via HTTPS.
- TLS 1.2 or higher is required for all connections.
- Certificates are issued and managed by Azure-managed certificates or retrieved securely from Azure Key Vault, depending on the environment.

Internal Service Communication

- Internal API communication between services (e.g., API Gateway → Authentication Service → UsersAPI) is also protected using HTTPS.
- Communication within Azure Virtual Network (VNet) may use private endpoints, further reducing attack surfaces.

Additional Notes

- Azure App Services and Azure API Management enforce HTTPS by default.
- Custom domains are supported and secured using domain-validated certificates.

This ensures full end-to-end encryption, aligning with best practices for secure web service architecture.

2.5 Compliance

The design in the SOA System is though to comply with the following:

- Compliant with GDPR.
- Data Encryption and Standards
- Data Classification and Retention
- Cross Border Data Transfer
- Audit and Logging Integrity
- Multi-Tenant Data Isolation

---

3. Sizing and Scalability Guidelines

3.1 User Load

- Target support: **700,000 users**
- Average transaction count: **300/user/year**
- Statement generation frequencies:
    - Monthly
    - Quarterly
    - Semi-Annually
    - Annually

3.2 Performance Model

- **Lazy loading** strategy:
    - Statements are pre-generated by background jobs the stores in DB, Cache and Blob.
    - **Redis** cache stores URLs to PDF files
    - If missing, statements are generated on-demand

3.3 Scalability

- All services are **containerized using Docker.**
- **Redis** for caching, **PostgreSQL** with read replicas (planned)
- auto-scaling friendly infrastructure. Each service will be having auto-scaled setup.

3.4 High Availability

- Load balancers across services
- Retry policies and message queues for resilience
- Regular **backups** of database and file storage
- Monitoring and health checks via **OCS API**

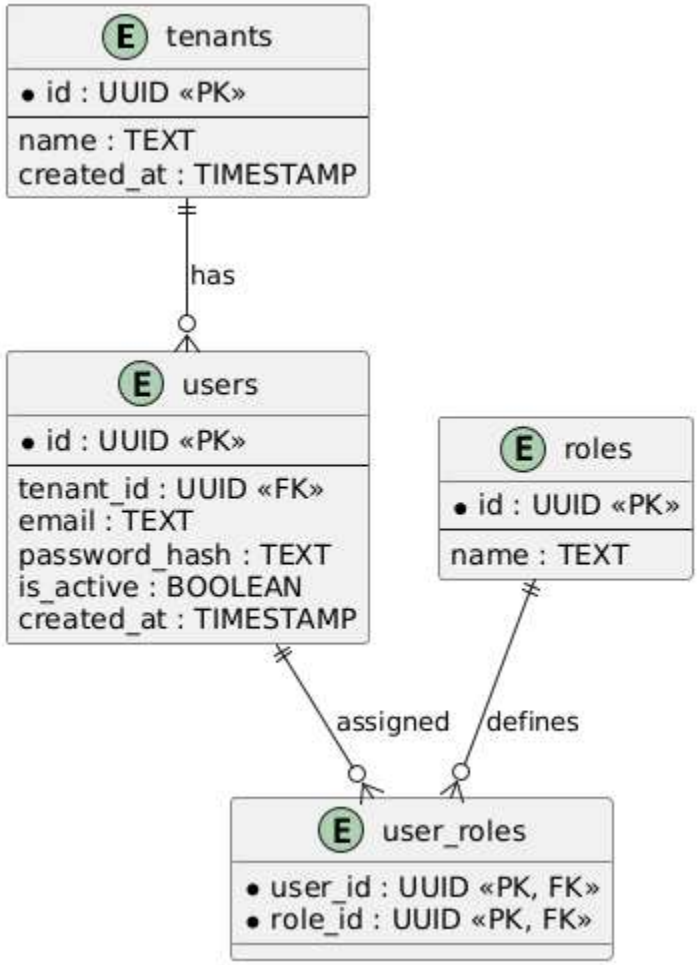
---

4. PoC

A fully asynchronous service that handles user creation and authentication, with the ability to perform admin-level user CRUD operations via an /admin/users endpoint.
The application has been built in C# and .NET Tool Stack. The code can be fond at:
https://github.com/EfrinGonzalez/SOA-GenerationSystem/tree/main/SOA-GenerationSystem-Solution

4.1 Data Base Design

DB ER Diagram
(A PlantUML-based ER diagram is also attached as: SOA-ERD.png)



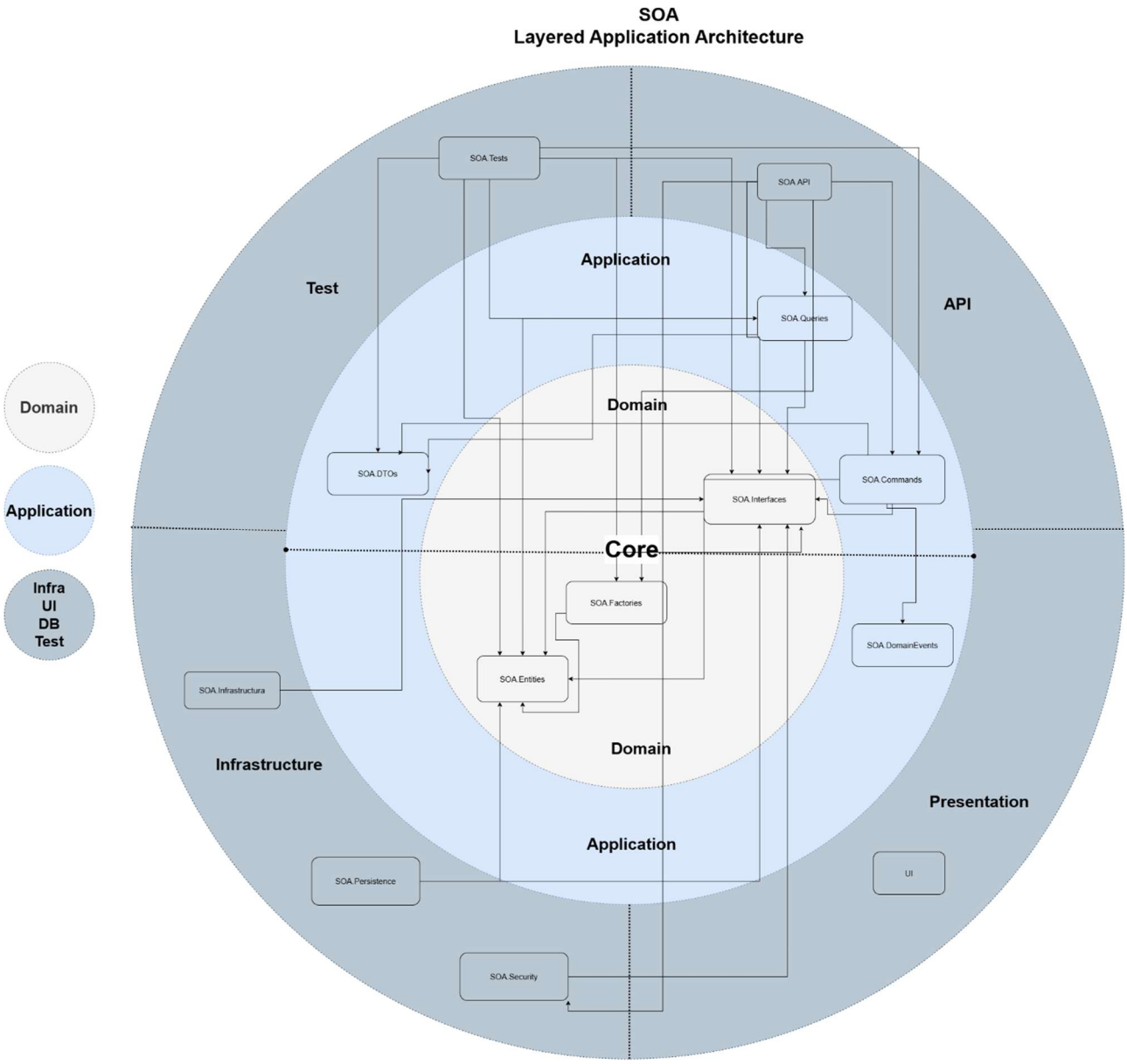
Script to create it in PostgreSQL

```
-- Create table: tenants
CREATE TABLE tenants (
   id UUID PRIMARY KEY,
   name TEXT NOT NULL,
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

-- Create table: users
CREATE TABLE users (
   id UUID PRIMARY KEY,
   tenant_id UUID NOT NULL,
   email TEXT NOT NULL UNIQUE,
   password_hash TEXT NOT NULL,
   is_active BOOLEAN NOT NULL DEFAULT TRUE,
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (tenant_id) REFERENCES tenants(id) ON DELETE CASCADE
);


-- Create table: roles
CREATE TABLE roles (
   id UUID PRIMARY KEY,
   name TEXT NOT NULL UNIQUE
);

-- Create table: user_roles
CREATE TABLE user_roles (
   user_id UUID NOT NULL,
   role_id UUID NOT NULL,
   PRIMARY KEY (user_id, role_id),
   FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
   FOREIGN KEY (role_id) REFERENCES roles(id) ON DELETE CASCADE
);
```

4.2 PoC Application Architecture

As mentioned before in this document, each Microservice will contain the following structure, giving a Clean Architecture, following best practices in Software Development and Architecture.

(Attached SOA-Layered-Architecture.png)



4.3 Deploying Instructions

- Run PostgreSQL Database and create a “SOA_Users_DB” database.
- Run the scripts in section 4.1 to create the tables.
- Clone the repository from github
- Open the solution in Visual Studio
- Set the SOA.UsersAPI as a Start Project
- Go to an endpoint following one of the examples in 4.4

4.4 Examples to hit the Enpoint inside visual studio "SOA.UsersAPI.http" File

```
@SOA.UsersAPI_HostAddress = http://localhost:5164

POST {{SOA.UsersAPI_HostAddress}}/admin/users/create
Content-Type: application/json
{
 "tenantId": "11111111-1111-1111-1111-111111111111",
  "email": "admin-created-user4@example.com",
  "password": "MyPass123!"
}

###
DELETE {{SOA.UsersAPI_HostAddress}}/admin/users/delete/{74f7b48a-490c-4137-8fc6-78a51309977c}

###
GET {{SOA.UsersAPI_HostAddress}}/admin/users/get-all

###
GET {{SOA.UsersAPI_HostAddress}}/admin/users/get/{898befcf-316d-4ba5-85a0-b001cf01d86f}

###
PUT {{SOA.UsersAPI_HostAddress}}/admin/users/update/{8c4700ef-750b-42a3-a018-70d6344fd876}
Content-Type: application/json
{
 "email": "updated@example.com",
  "isActive": false
}

###
POST {{SOA.UsersAPI_HostAddress}}/users/register
Content-Type: application/json
{
  "tenantId": "11111111-1111-1111-1111-111111111111",
  "email": "user8@example.com",
  "password": "MyPass123!"
}
```

4.5 API Endpoints Specification

```
openapi: 3.0.3
info:
  title: SOA.UsersAPI
  version: 1.0.0
  description: API for managing users within the SOA System
servers:
  - url: http://localhost:5164
paths:
  /admin/users/create:
    post:
```

```
    summary: Create a new user
    tags: [Admin]
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              tenantId:
                type: string
                format: uuid
              email:
                type: string
              password:
                type: string
    responses:
      '201': { description: User created }
      '400': { description: Validation error }
      '401': { description: Unauthorized }
      '403': { description: Forbidden }

  /admin/users/delete/{userId}:
    delete:
      summary: Delete a user
      tags: [Admin]
      parameters:
        - name: userId
          in: path
          required: true
          schema:
            type: string
            format: uuid
      responses:
        '204': { description: User deleted }
        '404': { description: User not found }

  /admin/users/get-all:
    get:
      summary: Get all users
      tags: [Admin]
      responses:
        '200':
          description: List of users

  /admin/users/get/{userId}:
    get:
      summary: Get a user by ID
      tags: [Admin]
      parameters:
        - name: userId
          in: path
```

```
      required: true
      schema:
        type: string
        format: uuid
  responses:
    '200': { description: User details returned }
    '404': { description: User not found }

/admin/users/update/{userId}:
  put:
    summary: Update a user
    tags: [Admin]
    parameters:
      - name: userId
        in: path
        required: true
        schema:
          type: string
          format: uuid
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              email:
                type: string
              isActive:
                type: boolean
    responses:
      '200': { description: User updated }
      '400': { description: Validation error }
      '404': { description: User not found }

/users/register:
  post:
    summary: Register a new user
    tags: [Public]
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              tenantId:
                type: string
                format: uuid
              email:
                type: string
              password:
```

```
            type: string
    responses:
      '201': { description: User registered }
      '400': { description: Validation error or missing tenant }

components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
security:
  - bearerAuth: []
```

[This document end here]