CampusCore

System Requirements Specification

Authors:

Boca Ioan Doru
Birlea Bogdan Alexandru
Boca Alina Maria
Blaga Alexandra

January 15, 2026

Contents

1 Project Description **1**
- 1.1 Overview . . . 1
- 1.2 Purpose & Scope . . . 1
- 1.3 System Philosophy . . . 1

2 Technology Stack **2**
- 2.1 Server-Side Architecture . . . 2
- 2.2 Client-Side Architecture . . . 3

3 Functional Requirements **4**
- 3.1 Authentication & Onboarding . . . 4
- 3.2 User Administration & Data Handling . . . 5
- 3.3 Schedule & Announcements . . . 5
- 3.4 Course & Grade Management (Professor) . . . 6
- 3.5 Student Portal . . . 6
- 3.6 Campus & Accommodation Services . . . 6
- 3.7 Communication (Chat) . . . 7

4 Non-Functional Requirements **8**
- 4.1 Performance & Scalability . . . 8
- 4.2 Security . . . 8
- 4.3 Reliability & Availability . . . 8
- 4.4 Usability & Interface . . . 8
- 4.5 Implementation Constraints . . . 9

5 Design Patterns **10**
- 5.1 Service Communication Envelope (Boca Ioan Doru) . . . 10
  - 5.1.1 Overview . . . 10
  - 5.1.2 The Problem . . . 10
  - 5.1.3 The Solution: MessageResponse Structure . . . 10
  - 5.1.4 Implementation Reference . . . 11
  - 5.1.5 Benefits . . . 11
- 5.2 Endpoint Protocol Adapter (Birlea Bogdan Alexandru) . . . 11
  - 5.2.1 Overview . . . 11
  - 5.2.2 The Problem . . . 12
  - 5.2.3 The Solution: CampusEndpoint Adapter . . . 12
  - 5.2.4 Implementation Reference . . . 12
  - 5.2.5 Benefits . . . 13
- 5.3 Users Generalization–Specialization Pattern (Boca Alina Maria) . . . 14
  - 5.3.1 Issues with a Model Without Generalization–Specialization . . . 14
  - 5.3.2 Benefits of the Adopted Solution . . . 14
- 5.4 gRPC Interceptor Proxy Pattern (Blaga Alexandra) . . . 14
  - 5.4.1 Benefits of the Interceptor-Based Approach . . . 14
  - 5.4.2 Consequences of Not Using an Interceptor . . . 15

6 Diagrams **16**
6.1 Get Message History Diagrams (Boca Ioan Doru) 16
6.2 Create Announcement Diagrams (Birlea Bogdan Alexandru) 18
6.3 Verify User Diagrams(Boca Alina Maria) 20
6.4 Reset Password Diagrams(Blaga Alexandra) 22

1 Project Description

1.1 Overview

CampusCore is a comprehensive university management system designed to streamline the complex flow of information between administration, faculty, and students. Developed as a robust desktop application, it serves as a centralized hub for academic logistics, residential management, and internal communication.

1.2 Purpose & Scope

The primary objective of CampusCore is to replace fragmented management tools with a unified digital ecosystem. The system is architected to handle key university operations, including:

- **User Lifecycle Management:** Secure onboarding and role-based access control for Administrators, Management, Professors, and Students.
- **Academic Administration:** Efficient handling of course schedules, enrollment, and grading systems.
- **Campus Services:** Dedicated modules for dormitory accommodation management and maintenance reporting.
- **Real-Time Collaboration:** An integrated messaging platform to facilitate seamless communication between all university stakeholders.

1.3 System Philosophy

CampusCore is built on three pillars:

1. **Efficiency through Automation:** Recognizing the volume of data in academic settings, the system prioritizes bulk operations. Features like "Smart Excel Import" allow administrators and professors to process hundreds of records (users, grades, schedules) in seconds rather than manual entry.
2. **Role-Specific Visibility:** The system enforces strict data segregation. Information is filtered so that users see only what is relevant to their specific group, role, or dormitory status, preventing information overload.
3. **Local Robustness:** Designed as a university project, CampusCore operates as a standalone desktop client in a local environment (Localhost). It simulates real-world constraints—such as payment gateways and email notifications—to provide a complete functional experience without requiring external infrastructure.

2 Technology Stack

2.1 Server-Side Architecture

The backend is built as a robust distributed system using **ASP.NET Core 9**. It follows a **Microservices Architecture**, separating concerns into distinct, independently deployable units.

Infrastructure & Data:

- **Containerization (Docker):** The entire backend ecosystem is containerized using Docker, ensuring consistent build and execution environments across development and deployment.
- **Database (MongoDB):** Data persistence is managed using MongoDB, a NoSQL database selected for its scalability and flexibility in handling diverse data schemas.
- **Logging (Serilog):** Aggregated logging is implemented using Serilog, providing structured, centralized logs for monitoring health and diagnosing issues across all microservices.

Core Components & Communication:

- **API Gateway:** Acts as the single entry point for the client. It is implemented as an HTTP service using the `FastEndpoints` package to aggregate responses.
- **Authentication:** Security is centralized at the Gateway level using `Authentication` and `JwtBearer` for validating JSON Web Tokens (JWT).
- **Inter-Service Communication (gRPC):** Internal communication between microservices is handled via high-performance gRPC calls, utilizing `Grpc.AspNetCore`, `Grpc.Net.Client`, and `Grpc.Tools`.
- **Common Service (Shared Kernel):** A dedicated "Offline Service" handles base classes, abstractions, and common interaction definitions, ensuring consistency across the ecosystem.

Asynchronous & Real-Time Operations:

- **Event Bus (MassTransit):** A publish-consumer pattern is used for background tasks (e.g., asynchronous group deletion to prevent UI freezing) to ensure system responsiveness.
- **Real-Time Notifications (SignalR):** Instant messaging and chat notifications are powered by SignalR, allowing immediate server-to-client updates.

Specialized Modules & Standardization:

- **Data Processing:** Excel import/export functionality is handled by `ClosedXML`.
- **Email Service:** Transactional emails (e.g., OTPs, notifications) are routed through the external provider **Brevo**.

- **Unified Response:** The entire system implements a standardized response wrapper, ensuring that every service returns a single, consistent data structure for predictable client handling.

2.2 Client-Side Architecture

The user interface is a native Desktop Application built for the Windows ecosystem, focusing on performance and native OS integration.

- **Framework:** Windows Presentation Foundation (WPF).
- **Language:** C# for logic and XAML for declarative UI definitions.
- **Design:** A lightweight, responsive interface designed to consume the unified API response structure provided by the server.

3 Functional Requirements

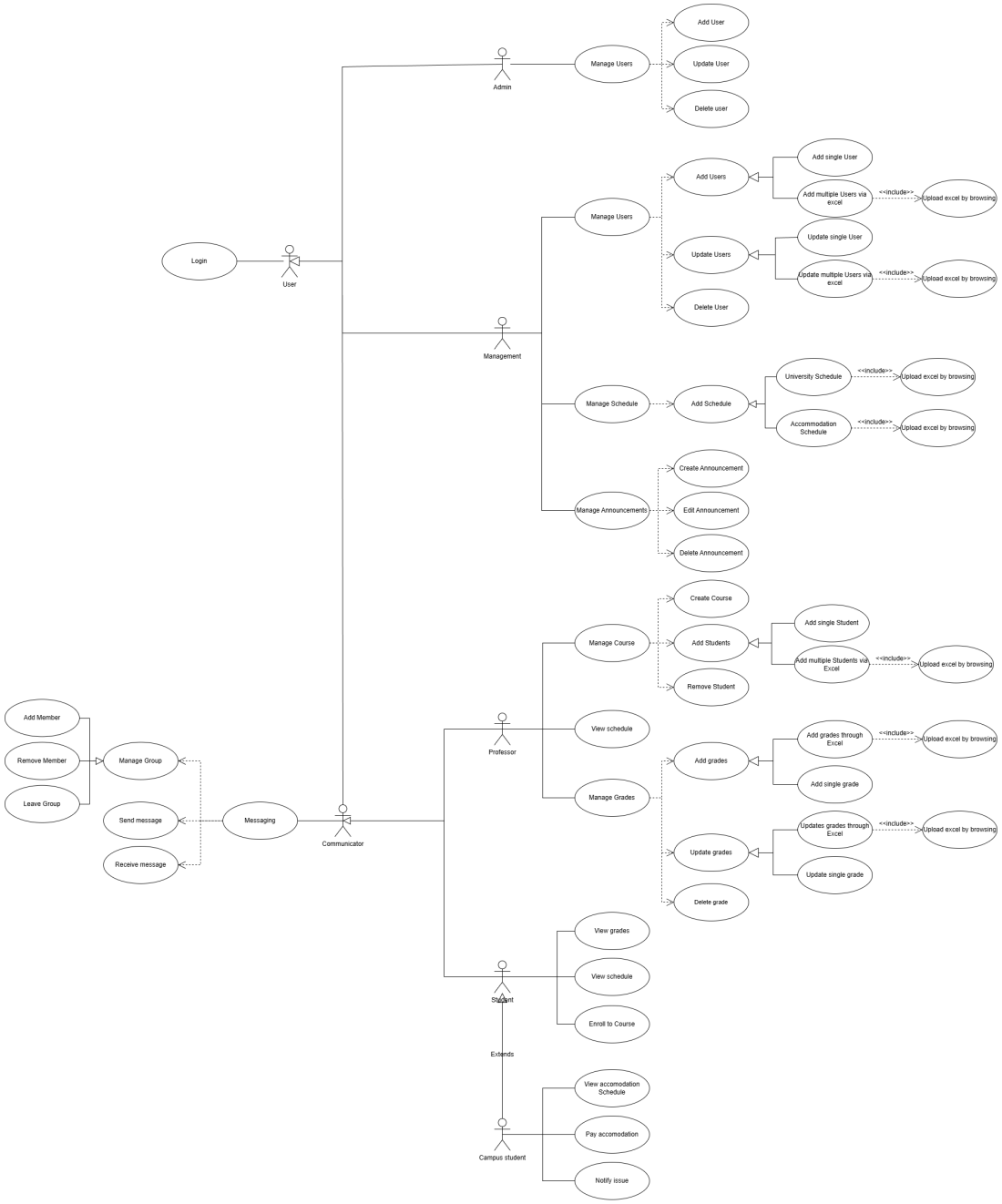

Figure 1: Use Case diagram

3.1 Authentication & Onboarding

Description:
Instead of a public sign-up page, this system uses an **Invitation-Based Onboarding** model. Management controls who enters the system. This ensures that only verified students and staff have access. The security model relies on a unique One-Time Password (OTP) verification for the initial account activation.

Key Functional Requirements:

- **Account Pre-creation:** Management users create accounts by inputting personal data (Name, Role, Email, etc.) but *excluding* the password.
- **First-Time Login Flow:**
  - Upon account creation, the system automatically sends an email to the user containing a unique **Verification Code**.
  - The user accesses the login page and enters their Email.
  - The system detects it is a first login and prompts for the Verification Code, a New Password, and a Confirm Password field.
- **Standard Login:** Subsequent logins use only Email and Password.
- **Recovery:** A "Forgot Password" feature allows users to reset credentials via email. A "Resend Verification Code" feature exists for users who lost their initial onboarding email.

3.2 User Administration & Data Handling

Description:
This module is designed for flexibility and bulk operations. The system uses a **Smart Import Logic**. It understands that a single Excel template might contain columns for all possible data (Email Address, Name, Group, etc.), but not every user needs every column filled. The system parses the file, ignores empty cells that aren't mandatory for the specific role, and validates the rest.

Key Functional Requirements:

- **Universal Excel Template:** The system accepts Excel files where columns represent all possible user attributes.
- **Conditional Parsing:** If a cell is empty (e.g., a Professor doesn't have a "Student Group" or "Dorm Room"), the system ignores it without throwing an error, provided that field is not mandatory for that specific role.
- **Strict Validation:** Upon upload, the system scans the file. If it finds invalid data (e.g., a duplicate email or text in a number field), it rejects the specific row and displays an error message detailing the row number and the issue.

3.3 Schedule & Announcements

Description:
Content visibility is highly segmented in this system to prevent information overload. Schedules are targeted to specific groups, while announcements are broadcasted institution-wide to ensure important news reaches everyone.

Key Functional Requirements:

- **Group-Based Schedule:** The "University Schedule" is linked to specific user groups. A student in "Group A" will not see the schedule for "Group B."

- **Role-Based Accommodation Schedule:** This schedule is visible *only* to users with the "Campus Student" role (students living in dorms).
- **Announcement Broadcast:** Announcements are visible to the entire university (all roles).
- **Push Notifications:** When Management creates or edits an announcement, the system automatically sends a notification to all relevant users.

3.4 Course & Grade Management (Professor)

Description:
This module is streamlined for speed and ease of use. It avoids complex academic syllabi in favor of a straightforward **Key-Based Enrollment** system. Grading is transparent and immediate.

Key Functional Requirements:

- **Simplified Course Creation:** Professors create a course by defining only the **Course Name** and a unique **Enrollment Key** (password).
- **Grading Scale:** The system enforces a grade input range of **1 to 10**.
- **Real-Time Visibility:** There is no "Draft" state for grades. As soon as a Professor uploads or enters a grade, it is visible to the student.
- **Bulk Grade Management:** Professors can download a template for their course, fill in grades for all students, and re-upload it to update the system instantly.

3.5 Student Portal

Description:
The student view is designed for self-service. Students act autonomously to join courses using the keys provided by professors. The dashboard prioritizes recent activity.

Key Functional Requirements:

- **Key-Based Enrollment:** Students search for a course or enter the **Enrollment Key** provided by the professor to join. There are no limits on how many students can join a course.
- **Chronological Sorting:** When viewing grades, the list is sorted by date (Newest First), so the most recent exam results appear at the top.
- **Historical Data:** Students retain access to grade history from previous academic years.

3.6 Campus & Accommodation Services

Description:
This module manages the residential aspect of student life. It includes a specific workflow for maintenance issues and a payment validation process.

Key Functional Requirements:

- **Payment Verification Process:**
  - The user must submit specific payment information (e.g., card details or transaction reference).
  - The system performs a verification check on the submitted information.
  - Once verified, the payment is validated.
  - Upon validation, the database updates the user's status to "Paid" for the current month, and a confirmation email ("Accommodation Paid") is dispatched to the student.
- **Directed Issue Reporting:** When a student reports a maintenance issue (e.g., "Broken Light"), they must select a *specific* Management user from a list. The system then emails the details solely to that selected manager.

3.7 Communication (Chat)

Description:
A real-time collaboration tool similar to WhatsApp or Slack, embedded within the application. It empowers students to organize their own study groups without needing faculty permission.

Key Functional Requirements:

- **Real-Time Messaging:** Messages are delivered instantly (WebSockets/Push).
- **File Transfer:** Users can attach and send documents/images within the chat.
- **Democratized Groups:** Any user (Student or Professor) can create a group chat.
- **Group Administration:**
  - The creator of the group is the **Group Admin**.
  - **Add/Remove:** Users can add or remove members.
  - **Delete Privilege:** Only the Group Admin can permanently delete the group chat.

4 Non-Functional Requirements

4.1 Performance & Scalability

- **Response Time (Latency):**
  - **Standard Operations:** Interactive UI elements (navigation, viewing details) must respond within $<$ **1 second** to ensure a fluid user experience.
  - **Complex Operations:** Bulk processing tasks (e.g., parsing an Excel file with 100+ students) should complete within $<$ **10 seconds**. The system must display a loading indicator during these operations.
- **Scalability:** The system architecture must support **Horizontal Scalability**. It should be capable of handling increased load by spawning additional instances (images) of the service, even within a local simulation environment.

4.2 Security

- **Password Policy:** The system enforces a strict password complexity rule. Passwords must contain at least:
  - 8 characters total length.
  - 1 uppercase letter.
  - 1 numeric digit.
  - 1 special character (e.g., !, @, #).
- **Data Protection:** User passwords must never be stored in plain text. They must be secured using a strong hashing algorithm (e.g., BCrypt or Argon2) before storage.
- **Session Management:** Sessions are persistent. Users remain logged in indefinitely until they explicitly click "Logout" or the application is closed.

4.3 Reliability & Availability

- **Availability:** The application is designed for high availability during uptime. As a local desktop application, availability relies on the host machine's stability.
- **Data Recovery:** For the scope of this project, there is **no automated backup mechanism**. Data persistence relies on the integrity of the local database file.

4.4 Usability & Interface

- **Language:** The application interface (menus, buttons, error messages) is strictly in **English**.
- **Platform:** The application is a **Desktop Client**. It requires no web browser support and is optimized for standard desktop resolutions.

4.5 Implementation Constraints

- **Deployment:** The system is intended for **Local Environment** deployment (Localhost).
- **Connectivity:** The desktop client communicates with the backend services locally; no external internet connection is required for core functionality (except for simulated email transmission).

5 Design Patterns

5.1 Service Communication Envelope (Boca Ioan Doru)

Pattern Name: Envelope Pattern (Canonical Data Model)
Component: `MessageResponse`
Context: Inter-service gRPC Communication

5.1.1 Overview

The system employs the **Envelope Pattern** to standardize communication across the distributed microservices architecture. Rather than exposing raw domain objects or relying exclusively on transport-level status codes (e.g., HTTP 500), all services exchange data via a unified wrapper contract, `MessageResponse` (see source file: `MessageResponse.cs`).

This pattern acts as a protocol-agnostic container, decoupling the **transport layer** (gRPC status) from the **business logic layer** (operation success/failure).

5.1.2 The Problem

In a distributed environment, direct service-to-service communication introduces several challenges:

- **Inconsistent Error Handling:** Different services may return errors via exceptions, null values, or disparate status codes, forcing the client to implement unique error-handling logic for every dependency.
- **Serialization Coupling:** Clients receiving raw objects must strictly share the exact type definitions of the server, making contract evolution difficult.
- **Opaque Failures:** Differentiating between a network failure (Transport error) and a validation failure (Business error) is often ambiguous without a standardized metadata layer.

5.1.3 The Solution: MessageResponse Structure

The `MessageResponse` structure creates a predictable contract for every network call. Regardless of the business outcome, the immediate return type is always a `MessageResponse` containing:

Field	Type	Description
`Success`	`bool`	High-level indicator of operation success. Clients check this first.
`Code`	`int`	Semantic status code (e.g., 200, 400, 404) independent of the gRPC transport status.
`Body`	`string`	The serialized JSON payload. This allows the envelope to carry polymorphic data types.
`Errors`	`string`	Human-readable error message if `Success` is false.

This structure facilitates a standardized "Unwrap" workflow: **Check Success → Inspect Code → Deserialize Payload.**

5.1.4 Implementation Reference

The efficacy of this pattern is demonstrated in the `UsersServiceImplementation` when consuming the `ExcelService`. The client service does not need to wrap calls in complex `try/catch` blocks to handle business logic failures; it simply inspects the envelope (see source file: `UsersServiceImplementation.cs`).

```
// 1. Uniform Execution
// The method returns a standard MessageResponse, shielding the client
// from internal exceptions occurring within the Excel Service.
var response = await _excelService.ParseExcelAsync(request);

// 2. Standardized Failure Handling
// Instead of catching specific exceptions, the client checks the 'Success' flag.
if (!response.Success)
    throw new InternalErrorException(response.Errors);

// 3. Payload Decoupling
// The envelope carries the data as a serialized payload. The client explicitly
// unwraps it into the expected 'ExcelData' type only after verification.
ExcelData? data = response.GetPayload<ExcelData>();
```

Listing 1: Consuming the Envelope in UsersServiceImplementation.cs

5.1.5 Benefits

- **Reliability:** Guarantees a consistent response format even during partial system failures.
- **Interoperability:** The JSON-based `Body` allows services to evolve their internal models without breaking the outer gRPC contract.
- **Observability:** Middleware and Interceptors can log the `Success` and `Code` fields of the envelope globally without needing to inspect the business payload.

5.2 Endpoint Protocol Adapter (Birlea Bogdan Alexandru)

Pattern Name: Interface Adapter (Gateway Adapter)
Component: `CampusEndpoint<TReq>`
Context: HTTP API Gateway / FastEndpoints Implementation

5.2.1 Overview

The system utilizes an **Adapter Pattern** within the HTTP layer to bridge the gap between the internal gRPC communication protocol and the external RESTful HTTP standard. The base class `CampusEndpoint` (see: `CampusEndpoint.cs`) acts as the adapter, translating the protocol-agnostic `MessageResponse` envelope used by microservices into standard HTTP responses required by web clients.

5.2.2 The Problem

The internal microservices communicate using a unified `MessageResponse` contract, where status is indicated by an integer `Code` (e.g., 200, 404) and a boolean `Success` flag, rather than native HTTP exceptions or status codes. However, the frontend framework (`FastEndpoints`) and external REST clients expect:

- Standard HTTP Status Codes (e.g., 200 OK, 401 Unauthorized).
- Raw JSON bodies without the internal envelope wrapper.
- Specific headers for authentication failures.

Without an adapter, every endpoint would require repetitive boilerplate code to manually map internal integer codes to HTTP responses, leading to code duplication and inconsistent API behavior.

5.2.3 The Solution: CampusEndpoint Adapter

The `CampusEndpoint` class inherits from the framework's `Endpoint` class but extends it to accept the internal `MessageResponse`. It centralizes the translation logic in the `SendAsync` and `HandleErrorsAsync` methods.

Translation Logic:

1. **Success Path:** If `response.Success` is true, the adapter automatically extracts the `Payload.Json` from the envelope and sends it as a raw HTTP 200 OK response.
2. **Failure Path:** If `Success` is false, the adapter inspects the `response.Code` and maps it to the corresponding HTTP method:
   - `400` → `SendErrorsAsync` (Bad Request)
   - `401` → `SendUnauthorizedAsync`
   - `403` → `SendForbiddenAsync`
   - `404` → `SendNotFoundAsync`

5.2.4 Implementation Reference

The `Login` endpoint demonstrates the adapter in action. The developer simply passes the raw gRPC response to `SendAsync`, and the base class handles the protocol translation (see: `Login.cs`).

```
1  public override async Task HandleAsync(LoginApiRequest req,
       CancellationToken ct)
2  {
3      // 1. Logic Execution
4      // The gRPC client returns the internal MessageResponse envelope.
5      MessageResponse grpcResponse = Client.Login(grpcRequest, null, null,
       ct);
6
7      // 2. Protocol Adaptation
8      // The SendAsync method adapts this internal envelope into a
9      // proper HTTP 200/400/401 response automatically.
10     await SendAsync(grpcResponse, ct);
```

```
11 }
```

Listing 2: Adapter Usage in Login.cs

5.2.5 Benefits

- **Consistency:** Ensures that a specific internal error (e.g., "User Not Found") always maps to the exact same HTTP Status Code (404) across the entire API surface.
- **Decoupling:** The individual endpoints (`Login`, `Register`, etc.) remain unaware of HTTP specificities. They operate purely on data input and the internal response contract.
- **Productivity:** Reduces the lines of code per endpoint by removing the need for manual `if/else` status checking and HTTP response construction.

5.3 Users Generalization–Specialization Pattern (Boca Alina Maria)

Within the user management service, it was necessary to model entities that share a common set of attributes while also presenting significant variations based on the role held in the system (for example: student, instructor, administrator). This situation represents a classic modeling challenge in object-oriented systems, where users have a shared identity but different responsibilities and specific data.

To address this challenge, the *Generalization–Specialization* pattern was adopted, based on inheritance and polymorphism. This approach enables the definition of a general entity (`User`) containing attributes common to all users (identifier, email, name, role, verification status), as well as its extension through specialized entities that add role-specific information.

5.3.1 Issues with a Model Without Generalization–Specialization

Without this pattern, one possible solution would have been to use a monolithic `User` class containing all possible attributes for every role. Such an approach would lead to numerous unused fields, null values, and artificial validations, increasing the risk of data inconsistency and making code maintenance more difficult.

5.3.2 Benefits of the Adopted Solution

By applying the Generalization–Specialization pattern, the user model becomes clearly structured, extensible, and coherent. Data is logically distributed according to role, eliminating unnecessary fields and enabling proper validation of information. Furthermore, the polymorphism provided by this model allows uniform treatment of users at the service level, without depending on their concrete type.

This same approach is reflected at the user service level as well, where a general interface defines common behavior, while logic specialization is achieved dynamically based on user role. Thus, a coherent, extensible architecture aligned with object-oriented development principles is obtained.

5.4 gRPC Interceptor Proxy Pattern (Blaga Alexandra)

The use of a gRPC interceptor in the proposed system is motivated by the need to manage cross-cutting concerns in a centralized, consistent, and scalable manner. In distributed service-oriented architectures, aspects such as error handling, validation, logging, and response normalization tend to affect all service operations equally, yet they are not part of the core business logic. Embedding these responsibilities directly within each service method leads to increased coupling and reduced maintainability.

The interceptor layer addresses this issue by introducing an intermediary component that uniformly governs the execution of all gRPC service calls. This design ensures that common behaviors are applied consistently across the entire service boundary, independently of the specific business functionality being executed.

5.4.1 Benefits of the Interceptor-Based Approach

The interceptor provides several critical advantages:

- **Centralized Error Handling:** All domain-specific and unexpected exceptions are intercepted at a single point and translated into standardized protocol-level responses. This guarantees predictable error semantics for all clients and prevents gRPC channel failures caused by uncaught exceptions.
- **Uniform Response Structure:** By enforcing a unified response envelope, the system ensures that all service operations return consistent metadata, status codes, and payload representations, simplifying client-side processing and reducing integration complexity.
- **Separation of Concerns:** Business logic remains isolated from infrastructural responsibilities such as logging, validation enforcement, and error mapping. This separation leads to cleaner service implementations and improves code readability.
- **Improved Maintainability and Extensibility:** Modifications to cross-cutting behavior can be performed in a single location without requiring changes to individual service handlers, thereby reducing the risk of regression and improving long-term maintainability.

5.4.2 Consequences of Not Using an Interceptor

In the absence of an interceptor-based middleware layer, each gRPC service method would be responsible for implementing its own error handling, validation checks, and response formatting logic. This approach introduces several significant drawbacks:

- **Code Duplication:** Identical try-catch blocks and response-mapping logic would need to be replicated across multiple service implementations, increasing code volume and reducing clarity.
- **Inconsistent Behavior:** Differences in error handling strategies between service methods may lead to inconsistent client experiences, with similar failures producing different response formats or status codes.
- **Tight Coupling:** Service handlers would become tightly coupled to transport-level concerns, making them harder to test, reuse, or migrate to alternative communication protocols.
- **Higher Risk of Unhandled Failures:** Uncaught exceptions could propagate beyond service boundaries, potentially terminating gRPC calls abruptly and causing undefined client-side behavior.
- **Reduced Scalability of System Evolution:** Introducing new cross-cutting requirements, such as authentication, authorization, or observability, would require invasive changes across all service implementations.

6 Diagrams

6.1 Get Message History Diagrams (Boca Ioan Doru)

This section visualizes the internal logic and flow of the "Get Messages" functionality using standard UML diagrams.

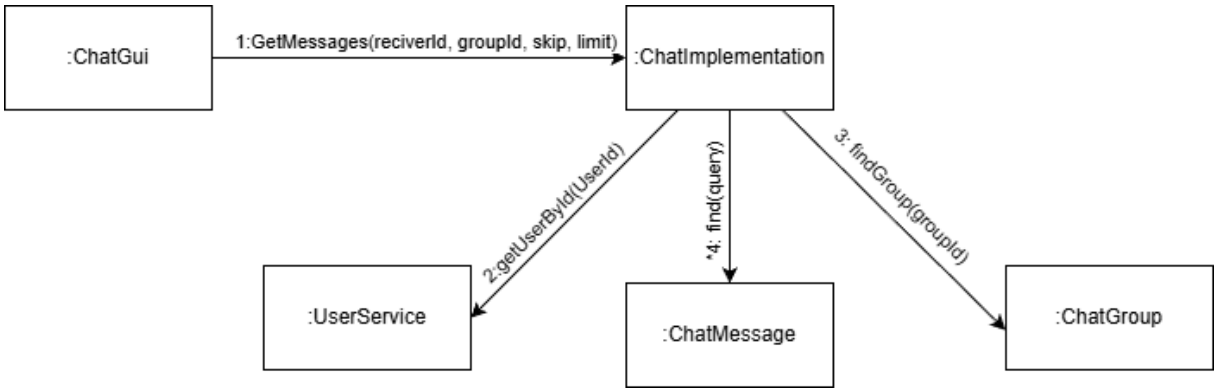

Figure 2: Communication Diagram: Object Interactions

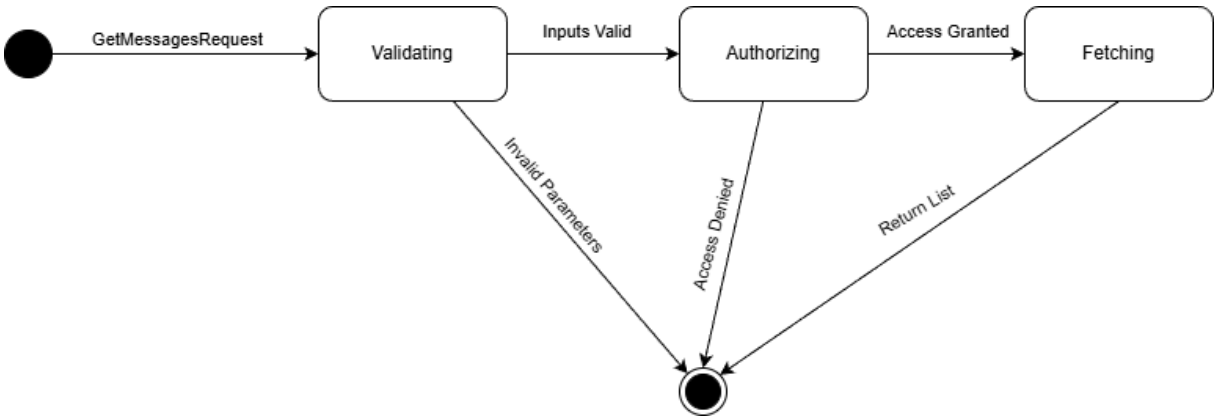

Figure 3: State Machine Diagram: Request Processing States

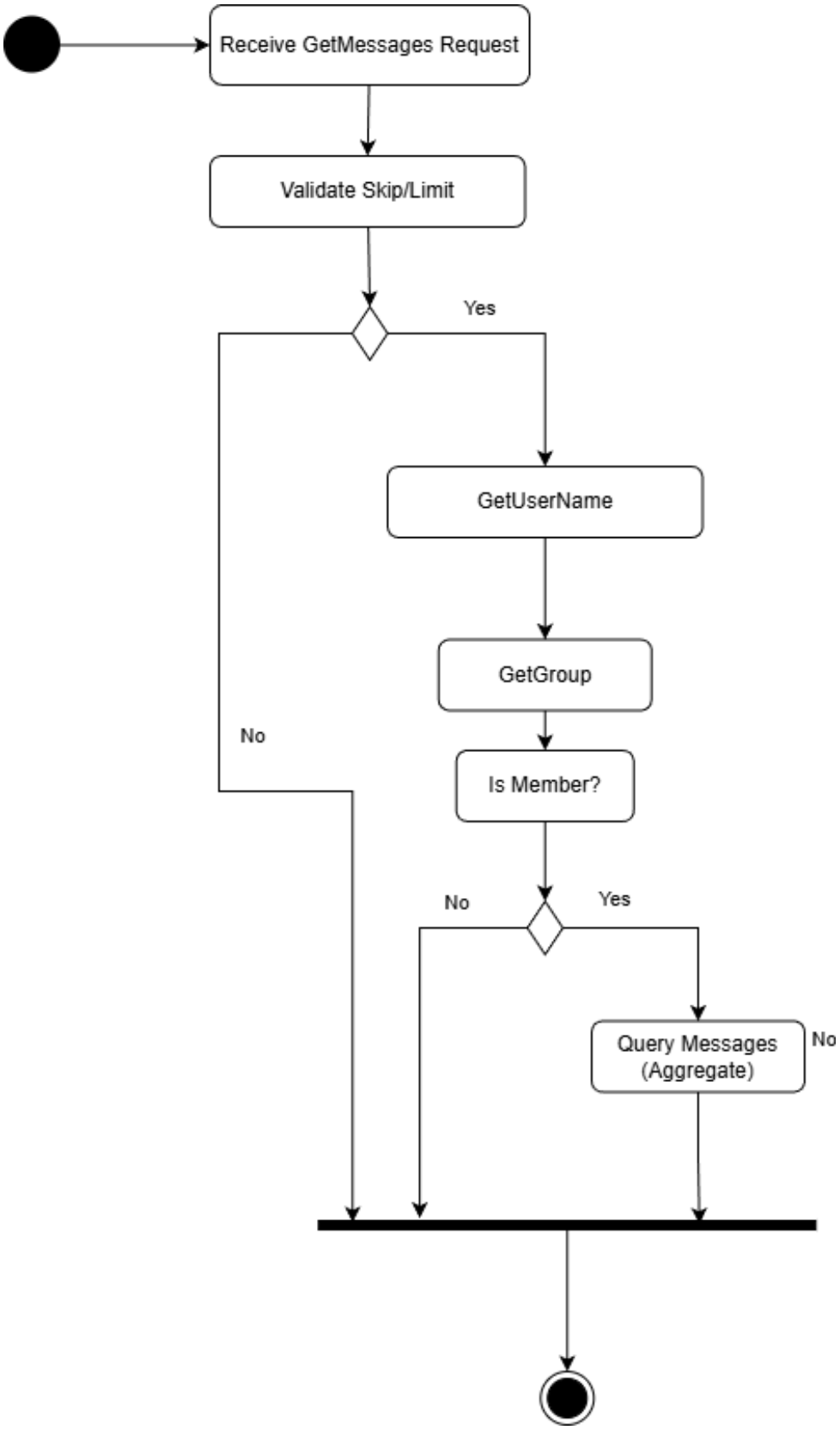

Figure 4: Activity Diagram: Get Messages Process Flow

6.2 Create Announcement Diagrams (Birlea Bogdan Alexandru)

This section details the architectural flow for creating system-wide announcements, highlighting the parallel execution of database insertion and asynchronous email notifications.

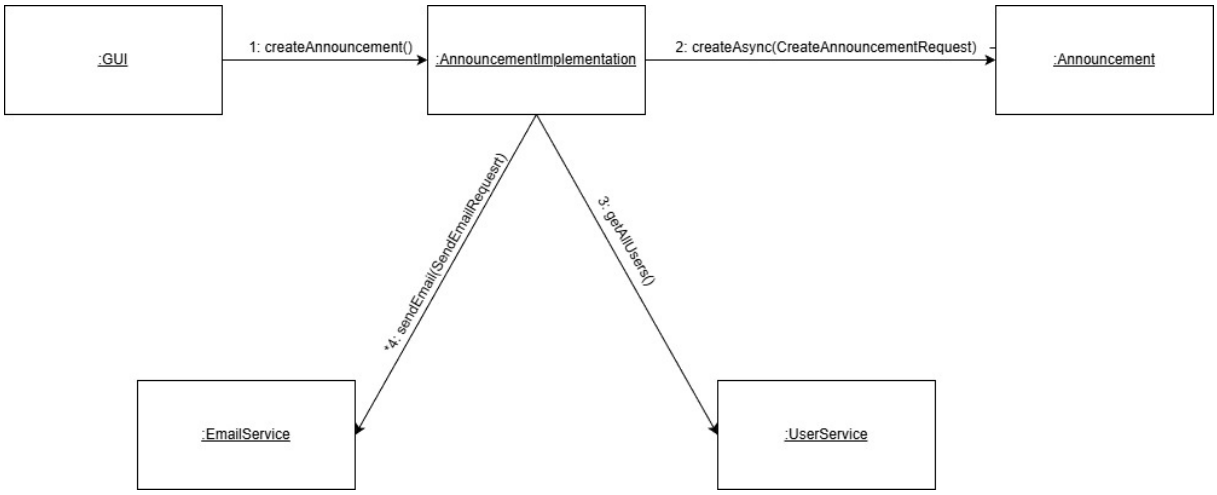

Figure 5: Communication Diagram: Service Interaction for Announcements

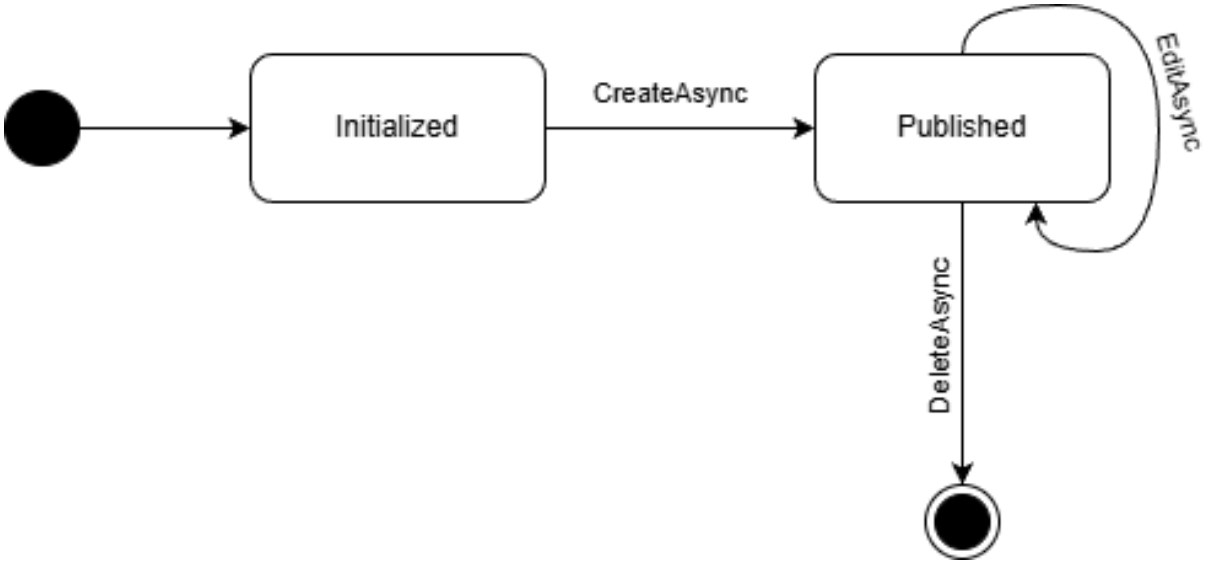

Figure 6: State Machine Diagram: Announcement Lifecycle

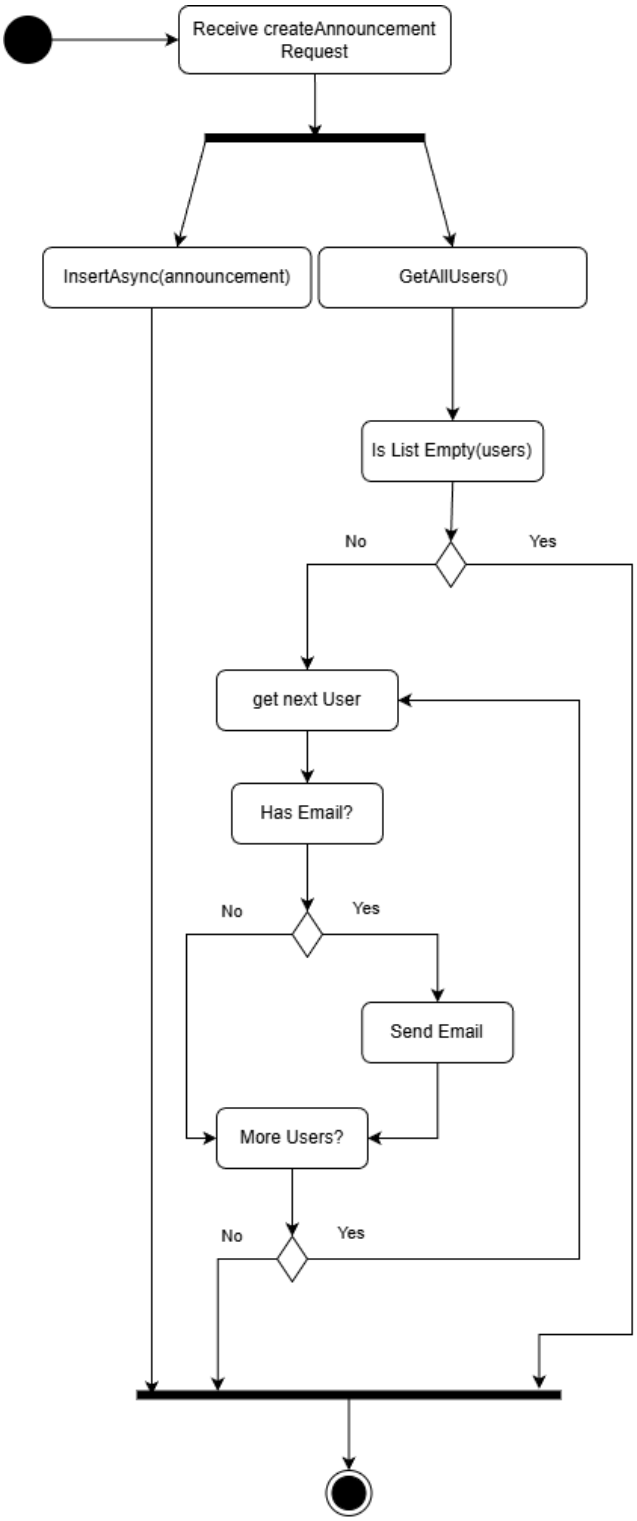

Figure 7: Activity Diagram: Announcement Creation & Broadcast Logic

6.3 Verify User Diagrams(Boca Alina Maria)

This section illustrates the security workflow for user verification, detailing the interaction between the GUI, Authentication Service, and the backend User Service during the first-time login process.

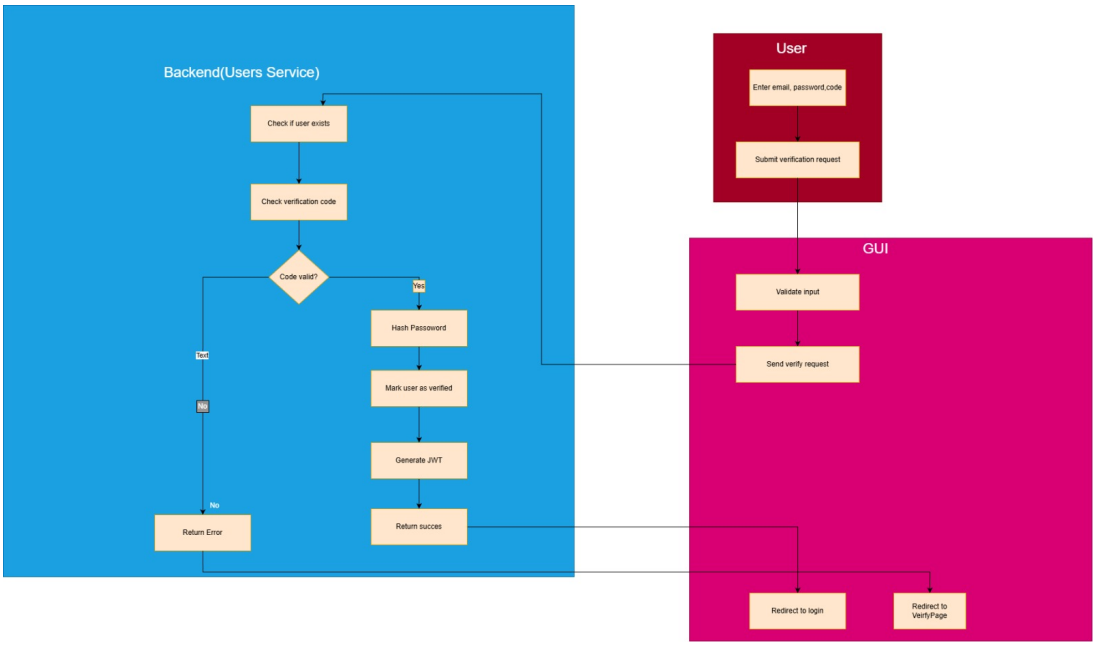

Figure 8: Activity Diagram: User Verification Logic

Figure 9: Communication Diagram: Verification Data Flow

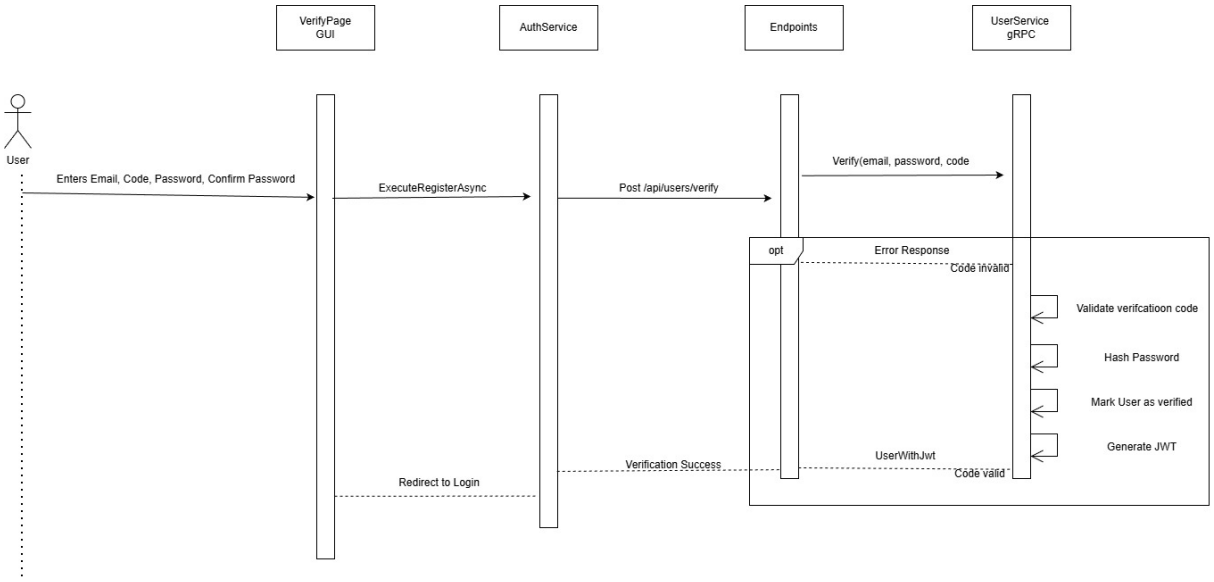

Figure 10: Sequence Diagram: End-to-End Verification Request

6.4 Reset Password Diagrams(Blaga Alexandra)

This section outlines the secure password recovery process, depicting the flow from the user's initial request to the email verification step and the final password update confirmation.

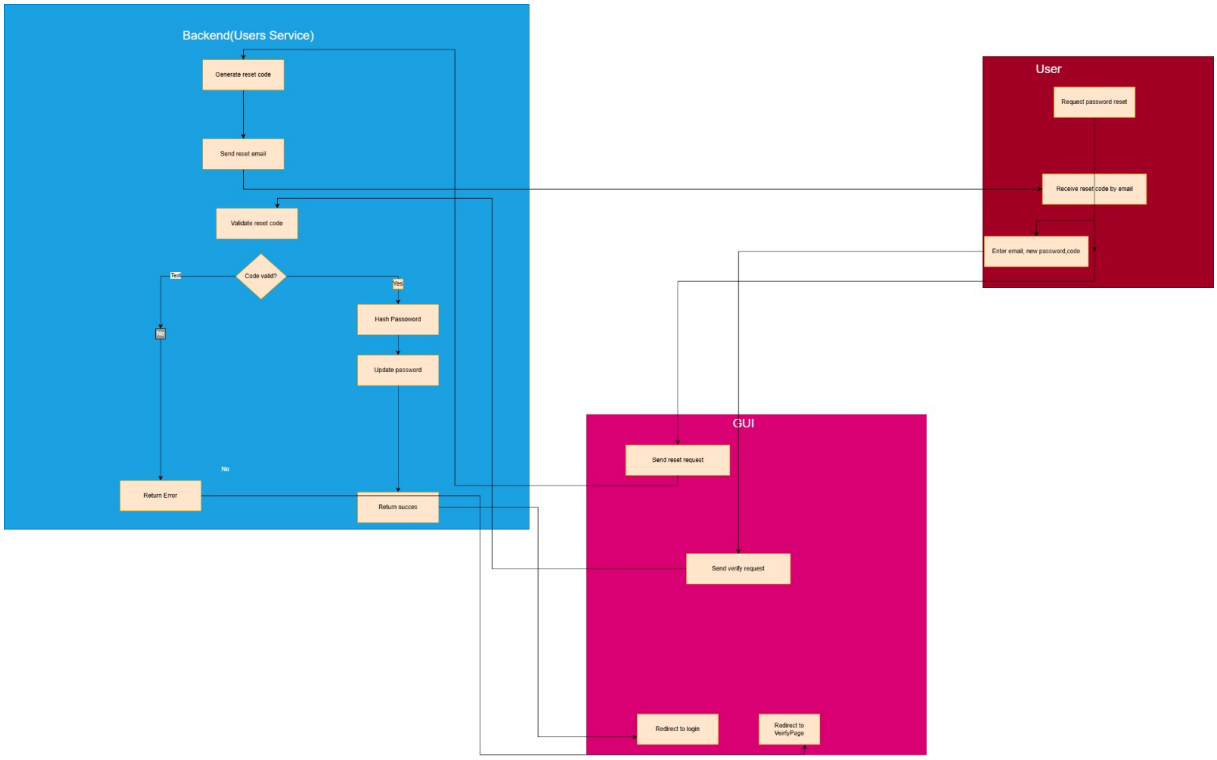

Figure 11: Activity Diagram: Password Reset Workflow

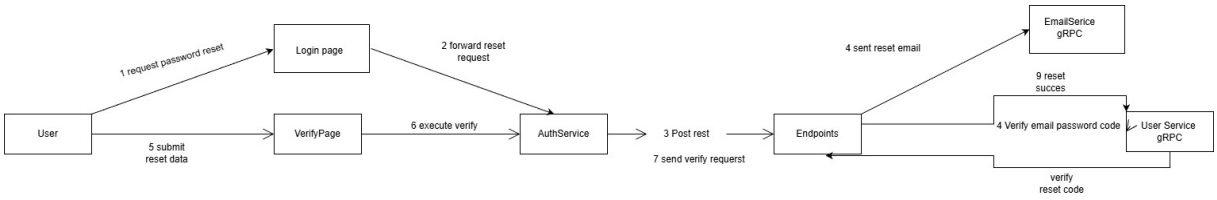

Figure 12: Communication Diagram: Service Interactions for Reset

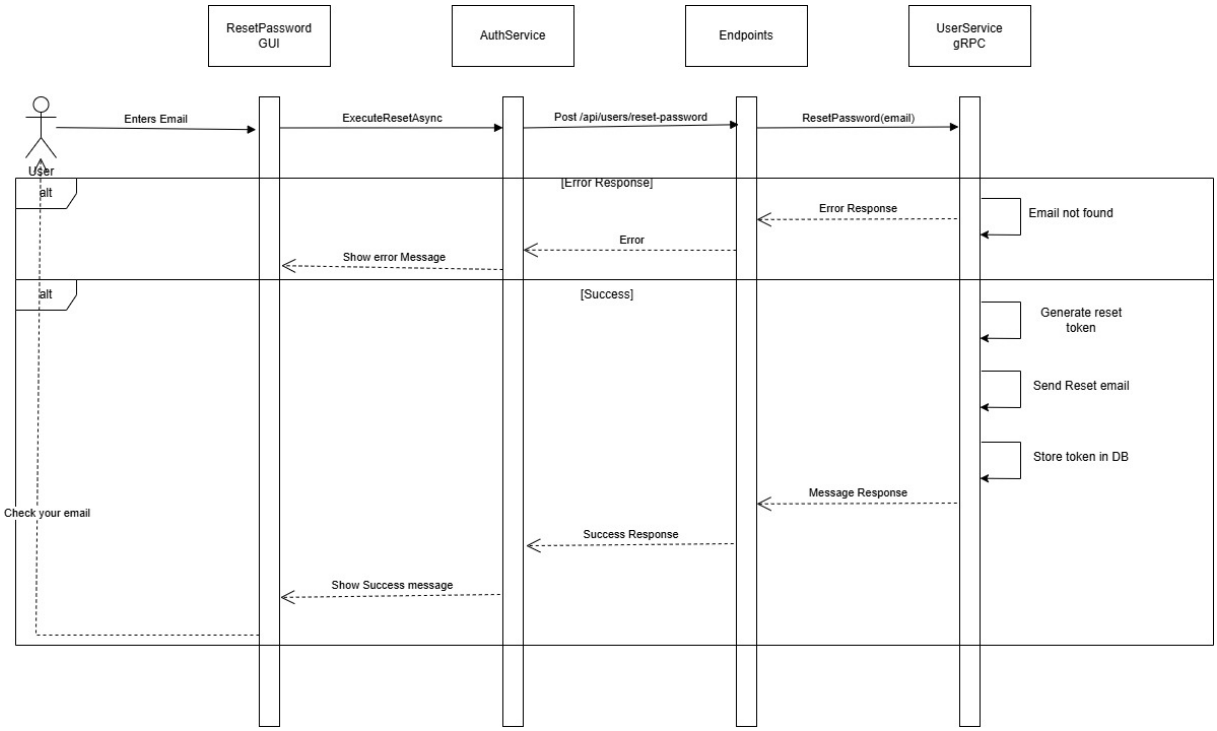

Figure 13: Sequence Diagram: Reset Token Generation & Validation