****

**Software Design Description**

**Project foodo**

**Team 11**

**Alper Çelik, 21903022**

**Batuhan Duras, 22002854**

**Baturalp Sönmez, 22103933**

**Cemal Fırat Dağ, 22003819**

**Davut Durmaz, 21903291**

Bilkent University

Department of Information Systems and Technologies

08.01.2025

Change History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File Name** | **Document Type** | **Deliverable** | **Version** | **Submission Date** |
| Deliverable\_4.docx | MS Word 2013 | 4 | 0.1 | - |
| Deliverable\_4\_2.docx | MS Word 2013 | 4 | 0.2 | - |
| Team11\_SDD\_v3.docx | MS Word 2013 | 4 | 0.3 | - |
| Team11\_SDD\_v4.docx | MS Word 2013 | 4 | 0.4 | - |
| Team11\_SDD\_FR1 | MS Word 2013 | 4 | 1 | 08.01.2025 |

Project Team

|  |  |  |
| --- | --- | --- |
| Name, Surname | Student Id Number | e-mail |
| Alper Çelik | 21903022 | alper.celik@ug.bilkent.edu.tr |
| Batuhan Duras | 22002854 | batuhan.duras@ug.bilkent.edu.tr |
| Baturalp Sönmez | 22103933 | baturalp.sonmez@ug.bilkent.edu.tr |
| Cemal Fırat Dağ | 22003819 | firat.dag@ug.bilkent.edu.tr |
| Davut Durmaz | 21903291 | davut.durmaz@ug.bilkent.edu.tr |

Project Details

|  |  |
| --- | --- |
| **Project Name** | **foodo** |
| **Software Name** | **foodo** |
| **Company Name** | **No sponsor company** |
| **Academic Advisor** | **Dr. Serkan Genç** |
| **GitHub URL** | **https://github.com/foodoHub** |
| **WEB page** | **https://foodo.cfd** |

Individual Contributions Overview

|  |  |
| --- | --- |
| **Name, Surname** | **Summary of Contributions to the SDD Document** |
| Alper Çelik | None. |
| Batuhan Duras | Contributed to the whole document on equal terms as his teammates. |
| Baturalp Sönmez | Contributed to the whole document on equal terms as his teammates. |
| Cemal Fırat Dağ | Contributed to the whole document on equal terms as his teammates. |
| Davut Durmaz | None. |

Executive Summary

This Software Design Description outlines the comprehensive design framework for foodo, a mobile application designed to transform how users approach cooking. By combining an AI-powered agent with personalized meal suggestions and social features, foodo aims to create a seamless and engaging user experience. The document bridges the gap between the high-level requirements defined in the Software Requirements Specification and the technical implementation of a scalable, maintainable, and user-focused system. The design of foodo is centered on a microservice architecture chosen for its scalability, reliability, and ability to support modular development. This architecture ensures that each service can be developed, tested, and scaled independently, from the AI assistant to the social platform. Using modern tools and technologies such as React Native, Spring Boot, Nest.js, Clerk, and AWS further enhances the system's ability to deliver reliable and efficient performance. The application's backend is supported by PostgreSQL for structured data and QDrant for AI-driven contextual searches, ensuring robust data management and intelligent user interactions. Development is structured into six carefully planned increments, each delivering key features progressively. The early stages focus on user authentication, basic navigation, and initial AI interactions. Later phases refine these features and introduce advanced functionalities such as notifications, achievement tracking, and complete meal suggestions. This incremental approach allows the team to prioritize core functionality early while iteratively improving and expanding the application's capabilities. The analysis model includes updated functional and non-functional requirements to ensure foodo meets user needs. Constraints such as a zero-budget model and the limited availability of development resources have been carefully addressed, balancing practicality with ambition. Strategic use of cost-effective tools and cloud services ensures the project remains feasible while maintaining high-quality outcomes. The system is also designed to meet strict non-functional requirements. Verification strategies ensure that key features, such as AI responsiveness, perform reliably under varying conditions.[[1]](#footnote-1)

Table of Contents

**Page Number**

[Executive Summary i](#_Toc187208420)

[Table of Contents ii](#_Toc187208421)

[1 Scope 1](#_Toc187208422)

[2 Analysis Model & Planning 1](#_Toc187208423)

[3 High-Level (Architecture) Design 26](#_Toc187208424)

[4 Low Level Design 66](#_Toc187208425)

[5 Discussions 118](#_Toc187208426)

[5.1 Limitations and Constraints 118](#_Toc187208427)

[5.2 Health and Safety Issues 119](#_Toc187208428)

[5.3 Legal Issues 120](#_Toc187208429)

[5.4 Economic Issues and Constraints 120](#_Toc187208430)

[5.5 Sustainability 121](#_Toc187208431)

[5.6 Ethical Issues 121](#_Toc187208432)

[5.7 Multidisciplinary Collaboration 121](#_Toc187208433)

[6 References 122](#_Toc187208434)

List of Tables

**Page Number**

**No table of figures entries found.**

List of Figures

[Figure 1. UML Use Case Diagram 1](#_Toc187208437)

[Figure 2. Block Diagram of foodo Software Architecture 29](#_Toc187208438)

[Figure 3. EER Diagram 30](#_Toc187208439)

[Figure 4. UML Communication Diagram of UC1 30](#_Toc187208440)

[Figure 5. UML Communication Diagram of UC2 32](#_Toc187208441)

[Figure 6. UML Communication Diagram of UC3 34](#_Toc187208442)

[Figure 7. UML Communication Diagram of UC4 36](#_Toc187208443)

[Figure 8. UML Communication Diagram of UC5 38](#_Toc187208444)

[Figure 9. UML Communication Diagram of UC6 40](#_Toc187208445)

[Figure 10. UML Communication Diagram of UC7 43](#_Toc187208446)

[Figure 11. UML Communication Diagram of UC8 46](#_Toc187208447)

[Figure 12. UML Communication Diagram of UC9 48](#_Toc187208448)

[Figure 13. UML Communication Diagram of UC10 50](#_Toc187208449)

[Figure 14. UML Communication Diagram of UC21 52](#_Toc187208450)

[Figure 15. UML Communication Diagram of UC22 54](#_Toc187208451)

[Figure 16. UML Deployment Diagram 56](#_Toc187208452)

[Figure 17. Pseudocode of UC1 66](#_Toc187208453)

[Figure 18. UML Class Diagram of UC1 68](#_Toc187208454)

[Figure 19. UML Sequence Diagram of UC1 69](#_Toc187208455)

[Figure 20. Pseudocode of UC2 70](#_Toc187208456)

[Figure 21. UML Class Diagram of UC2 72](#_Toc187208457)

[Figure 22. UML Sequence Diagram of UC2 73](#_Toc187208458)

[Figure 23. Pseudocode of UC3 74](#_Toc187208459)

[Figure 24. UML Class Diagram of UC3 76](#_Toc187208460)

[Figure 25. UML Sequence Diagram of UC3 76](#_Toc187208461)

[Figure 26. Pseudocode of UC4 77](#_Toc187208462)

[Figure 27. UML Class Diagram of UC4 78](#_Toc187208463)

[Figure 28. UML Sequence Diagram of UC4 79](#_Toc187208464)

[Figure 29. Pseudocode of UC5 80](#_Toc187208465)

[Figure 30. UML Sequence Diagram of UC5 83](#_Toc187208466)

[Figure 31. Pseudocode of UC6 85](#_Toc187208467)

[Figure 32. UML Sequence Diagram of UC6 89](#_Toc187208468)

[Figure 33. UML Class Diagram of UC 5&6 89](#_Toc187208469)

[Figure 34. Pseudocode of UC7.1 90](#_Toc187208470)

[Figure 35. UML Sequence Diagram of UC7.1 93](#_Toc187208471)

[Figure 36. UML Class Diagram of UC 7&8 93](#_Toc187208472)

[Figure 37. Pseudocode of UC7.2 94](#_Toc187208473)

[Figure 38. UML Sequence Diagram of 7.2 96](#_Toc187208474)

[Figure 39. Pseudocode of UC7.3 97](#_Toc187208475)

[Figure 40. Sequence Diagram of UC7.3 98](#_Toc187208476)

[Figure 41. Pseudocode of UC8 99](#_Toc187208477)

[Figure 42. UML Class Diagram of UC 7&8 101](#_Toc187208478)

[Figure 43. UML Sequence Diagram of UC8 102](#_Toc187208479)

[Figure 44. Pseudocode of UC9 103](#_Toc187208480)

[Figure 45. UML Class Diagram of UC9 104](#_Toc187208481)

[Figure 46. UML Sequence Diagram of UC9 104](#_Toc187208482)

[Figure 47. Pseudocode of UC10 105](#_Toc187208483)

[Figure 48. UML Class Diagram of UC10 107](#_Toc187208484)

[Figure 49. UML Sequence Diagram of UC10 108](#_Toc187208485)

[Figure 50. Pseudocode of UC21 109](#_Toc187208486)

[Figure 51. UML Class Diagram of UC21 111](#_Toc187208487)

[Figure 52. UML Sequence Diagram of UC21 112](#_Toc187208488)

[Figure 53. Pseudocode of UC22.1 113](#_Toc187208489)

[Figure 54. UML Sequence Diagram of UC22.1 114](#_Toc187208490)

[Figure 55. Pseudocode of UC22.2 115](#_Toc187208491)

[Figure 56. UML Sequence Diagram of UC22.2 116](#_Toc187208492)

[Figure 57. UML Class Diagram of UC22.2 117](#_Toc187208493)

[Figure 58. The broken water fountain in the A building 118](#_Toc187208494)

[Figure 59. Receipt for vending machine coffee purchase at Ankara Otomat 119](#_Toc187208495)

[Figure 60. Receipts for lamb döner meals from ANKARA GECE DÖNERCİSİ 119](#_Toc187208496)

[Figure 61. Baturalp recovering from muscle soreness after a weight-lifting session 120](#_Toc187208497)

Abbreviations

|  |  |
| --- | --- |
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| AWS  DB | Amazon Web Services  Database |
| DTO | Data Transfer Object |
| EC2 | Elastic Compute Cloud (AWS) |
| EER | Enhanced Entity-Relationship |
| GUI | Graphical User Interface |
| JWT | JSON Web Token |
| LLM | Large Language Model |
| NFR | Non-Functional Requirement |
| NPV | Net Present Value |
| SDK | Software Development Kit |
| SDD | Software Design Description |
| SRS | Software Requirements Specification |
| UML | Unified Modeling Language |
| URL | Uniform Resource Locator |
| WBS | Work Breakdown Structure |

# Scope

This Software Design Description (SDD) aims to provide a comprehensive and detailed framework for the architectural and design activities involved in developing **foodo.** This document acts as a bridge between the requirements outlined in the Software Requirements Specification (SRS) and the actual implementation of the software.

**foodo** is designed to assist users in preparing meals through a responsive AI agent and engaging social features. By integrating cutting-edge technologies such as machine learning, microservice architecture, and cloud infrastructure, the application aims to deliver a seamless and personalized cooking experience. The SDD ensures a structured approach to developing a scalable, reliable, and maintainable software system.

This document outlines the following aspects of the design process:

**Architectural Design**:

* + Details of the chosen microservice architecture and its alignment with the project's scalability, maintainability, and reliability goals.
  + Justification for architectural decisions, considering non-functional requirements and trade-offs.
  + Logical, process, and physical views to showcase component interactions and integrations.

**Analysis Model**:

* + Updated functional and non-functional requirements, reflecting design priorities and considerations.
  + Incremental planning with a breakdown of use cases, features, and estimated completion percentages for each of the six increments we planned.
  + Assessment of constraints, usability, performance, and error-handling requirements.

**High-Level Design**:

* + A logical representation of the system using block diagrams and UML communication diagrams. Use Case Realizations are also included after each diagram.
  + A process view showing workflows and interactions for key use cases.
  + A physical view highlighting the deployment environment and the interaction between components.

**Low-Level Design**:

* + Detailed class diagrams, pseudo-code, and sequence diagrams for first-increment use cases.
  + Inclusion of all essential details, such as method parameters, return types, visibility, multiplicity, and primitive types for attributes.
  + Explanations of low-level interactions between components to meet the functional requirements.

**Frameworks, Services, and APIs**:

* + A detailed overview of the frameworks, tools, and third-party services employed in the project.
  + Integration timelines for each technology and their role in the development lifecycle, including authentication, vector databases, cloud services, and backend infrastructure.

**Discussions**:

* + Insights into challenges encountered during the design process and strategies to address them.
  + Analysis of constraints, health and safety issues, legal and ethical considerations, sustainability efforts, and multidisciplinary collaboration.

**Design Quality and Verification**:

* + A thorough explanation of how the design meets each non-functional requirement, including usability, reliability, maintainability, and performance.
  + Verification and validation strategies to ensure compliance with the design goals.

The design activities described in this document ensure a robust, scalable, and user-friendly application. This SDD serves as a blueprint for successfully implementing **foodo** by sticking to industry best practices such as modularity and efficient resource utilization. Each section is crafted to reflect a comprehensive understanding of the system's technical and functional aspects, ensuring alignment with project objectives and user expectations.

# Analysis Model & Planning

1. **Functional Requirements**

Figure . UML Use Case Diagram

* 1. A diagram of a diagram

     Description automatically generated**Use Cases and Increments**

Here is list of the use cases and what percent of them will be implemented for each increment.

**Increment 1 (Deliverable 9):**

* UC1 (Login) 100% Completed
* UC2 (Recover Password) 100% Completed
* UC3 (Register New User) 100% Completed
* UC4 (Navigate Pages) 100% Completed
* UC5 (Get Meal Suggestions) 30% Completed
* UC6 (Get Recipe Instructions) 30% Completed
* UC7 (Manage Posts) 40% Completed
* UC8 (View Posts) 100% Completed
* UC9 (View Profile) 50% Completed
* UC10 (Edit Profile) 100% Completed
* UC21 (View AI Chat Lists) 60% Completed
* UC22 (Manage AI Chats) 60% Completed

**Increment 2 (Deliverable 10):**

* UC5 (Get Meal Suggestions) 40% Completed
* UC6 (Get Recipe Instructions) 40% Completed
* UC9 (View Profile) 100% Completed
* UC15 (Send Notification) %40 Completed
* UC16 (View Notifications) %30 Completed
* UC20 (Search Users) 100% Completed

**Increment 3 (Deliverable 11):**

* UC5 (Get Meal Suggestions) 60% Completed
* UC6 (Get Recipe Instructions) 60% Completed
* UC11 (View Achievements) 20% Completed
* UC12 (Earn Achievements) 20% Completed
* UC15 (Send Notifications) 80% Completed
* UC16 (View Notifications) 80% Completed
* UC17 (Manage Notifications) 90% Completed
* UC21 (View AI Chat Lists) 70% Completed
* UC22 (Manage AI Chats) 70% Completed
* UC13 (Manage Friends) 100% Completed

**Increment 4 (Deliverable 12):**

* UC5 (Get Meal Suggestions) 70% Completed
* UC6 (Get Recipe Instructions) 70% Completed
* UC11 (View Achievements) 100% Completed
* UC12 (Earn Achievements) 80% Completed
* UC19 (Interact with Posts) 40% Completed
* UC22 (Manage AI Chats) 100% Completed
* UC21 (View AI Chat Lists) 100% Completed

**Increment 5 (Deliverable 13):**

* UC5 (Get Meal Suggestions) 90% Completed
* UC6 (Get Recipe Instructions) 90% Completed
* UC7 (Manage Posts) 100% Completed
* UC14 (View Friends) 100% Completed
* UC15 (Send Notifications) 100% Completed
* UC16 (View Notifications) 100% Completed
* UC17 (Manage Notifications) 100% Completed
* UC19 (Interact with Posts) 70% Completed

**Increment 6 (Deliverable 14):**

* UC5 (Get Meal Suggestions) 100% Completed
* UC6 (Get Recipe Instructions) 100% Completed
* UC18 (Share Experience Feedback) 100% Completed
* UC19 (Interact with Posts) 100% Completed
  1. **List of Functional Requirements**

**FReq1.1:** Upon the first launch, the mobile app shall display a minimal homepage with the following elements:

* A **"Login"** button shall direct users to the login screen to complete **FReq1.2.**
* A **"Sign Up"** button shall direct users to the registration screen to complete **FReq1.4.**

**FReq1.2:** Upon completing **FReq1.1**, the mobile app shall display a GUI with the following elements:

* An **"Email Address"** input field, accepting a maximum of 256 characters and validated against standard email format rules.
* A **"Password"** input field, accepting a minimum of 8 and a maximum of 256 characters. The password shall be hidden by default, with an option to display the entered text.
* A **"Login"** button remains disabled until both fields are valid. When pressed, the app shall use Clerk Authentication to log in the user. If the credentials are incorrect or the account does not exist, an error message (**EH-U2**) will be displayed. If there is no error, the system shall redirect the user to the social feed (**FReq4.4.1**).
* A **"Forgot Password?"** link, allowing users to initiate the **FReq1.3.**
* A **"Back"** button which returns the user to homepage, also clears “Email Address” and “Password” fields.

**FReq1.3:** The mobile app shall display a GUI with "Forgot Password?" title and the following elements:

* An **"Email Address"** input field, accepting a maximum of 256 characters and validated against standard email format rules.
* A **"Send Reset Link"** button remains disabled until the “Email Address” field is valid. When pressed, the app shall use Clerk Authentication to send a password reset link. If Clerk Authentication System handles password requests successfully then the system shall notify the user to check his/her email inbox.

**FReq1.4**: Upon completing **FReq1.1**, the mobile app shall display a GUI containing:

* A "Next" button.
* A "Name" input field, accepting a minimum of 2 and a maximum of 64 characters.
* An "Email Address" input field, accepting a maximum of 256 characters, which shall be validated against standard email format rules.
* A "Password" input field, accepting a minimum of 8 and a maximum of 256 characters, which must contain at least one number, one lowercase letter, one uppercase letter, and one special character from the set “!"#$%&'()\*+,-./:;<=>?@[]^\_`{|}~”.
* A "Re-enter Password" input field, which must match the "Password" field.

The "Next" button shall remain disabled until all fields are valid according to the specified criteria. Once all criteria are met, the "Next" button shall become enabled. When the "Next" button is pressed, the app shall send all collected information to the Clerk Authentication Service to create a new user account. If errors are detected in the entered information, the app shall display an error label in accordance with error handling requirements (**EH-U1**). If there is no error, the system shall show the user the onboarding GUI to complete **FReq1.5.**

**FReq1.5:** After **FReq1.4** is complete, the mobile app shall display a new GUI with the following elements:

* A list of checkboxes for selecting dietary restrictions.
* A dropdown menu labeled "Cooking Level" with the options "Novice", “Average”, “Expert".
* A list of checkboxes labeled "Favorite Meals" that allows users to select their favorite meals.
* A "Start" button at the bottom of the GUI, initially disabled.

The "Start" button shall become enabled when both a dietary restriction and a cooking level have been selected, once the “Start” button is clicked the system shall record this information to be used in personalization of recipes to be given to the user and shall redirect the user to the AI Chatbot Page (**FReq3.1.1**).

**FReq2.1**: The mobile app shall provide a navigation bar on the footer including:

* “Social Feed” Icon Button on the left which redirects users to Social Feed Page (**FReq2.2**).
* “AI Chatbot” Icon Button in the middle which redirects users to AI Chatbot Page (**Freq2.3**).
* “User Profile” Icon Button on the right which redirects users to the profile page (**FReq2.4**).

**FReq2.2:** After the completion of **FReq2.1**, the mobile app shall enable users to use the social feed navigation header including:

* Text field with a “Search” placeholder to search users (**FReq4.3**).
* A “notifications” icon button which redirects users to notifications page (**FReq6.1**).

**FReq2.3:** Upon the completion of **FReq2.1**, the mobile app shall show the AI Chat Page (**FReq3.1.1**) with a navigation bar on top including:

* “Show More” Icon Button on the left, which redirects users to the “Chat List” page (**FReq3.3.1**) when pressed.
* “Delete Chat” Icon Button on the right, which enables users to delete chat (**FReq3.3.2**).

**FReq2.4:** The mobile app shall enable users to navigate to the profile page (**FReq4.1.1**) and a “Settings” icon button on top right of the corner, which is only shown when user is viewing his/her profile page, redirects user to profile settings page (**FReq4.1.2**).

**FReq3.1.1**: The mobile app shall provide users with the ability to interact with an AI assistant via text input accepting a minimum of 1 and a maximum of 1907 characters, and a “Send” button. When the “Send” button is clicked all the user input content will be sent to foodo AI to process (**FReq3.1.2.1**). If there is an error on the server side, an error message (**EH-S1**) will be shown to the user. If not and the response coming from foodo AI is a meal recipe, the system shall show a “Let’s cook this!” button to start the cooking instructions process (**FReq3.1.3**).

**FReq3.1.2.1**: When the **FReq3.1.1** is completed, the system shall get the user input and check if the user’s request is valid or not by prompting it to the LLM Provider. If the user request is not a valid request or wants something that will abuse the system, the user shall be warned with an error message (EH-AI1). Also, a function call check should be done within the same LLM request to check whether the user wants suggestions or detailed recipes. If the user wants detailed instructions the system shall continue with instructions (FReq3.1.2.3) else should continue with the suggestion process **(FReq3.1.2.2)**.

**FReq3.1.2.2**: When the **FReq3.1.2.1** is completed with a valid user request and suggestion request, the system shall vectorize the user request and search for the request in the Vector Database. The system shall filter results based on similarity scores which are greater than 0.5. If no results are present the system shall prompt the user to specify more details. If there are more than one results the system shall get details from results like name, ingredients, cooking time, the system shall get user preferences like dietary restrictions, cooking level and favorite meals from database, and embed all these inside the LLM prompt’s meal context part and make an LLM call to LLM Provider. The system shall show the response back to the user and also make meal names highlighted and clickable. After clicking on these linked meal names, the mobile app shall send a predefined message as “I want to prepare X” (X as the meal name) and reinitialize.

**FReq3.1.2.3**: When the **FReq3.1.2.1** is completed with a valid user request and detailed recipe instructions request, the system shall l make an LLM call to check user input whether it has a meal name or not. If no meal name is present, the system shall warn users (EH-AI1) about missing meal names. If the meal name is present, then the system shall vectorize the meal name and query Vector Database. The system shall filter results based on similarity scores which are higher than 0.8. If no result is present the system shall warn users about no similar recipe found and prompt users to give more details about the meal. If there are results, then the recipe instructions shall be sent to the LLM provider for a more structured output to use in instruction cards (**FReq3.1.3**).

**FReq3.1.3:** The app shall present cooking instructions in a card-based format, allowing users to easily navigate through each step of the process by swiping (swipe left for previous step, swipe right for next step) or previous and next icon buttons. Each card will represent a single cooking step and shall include the following elements:

* **Progress Tracker (FReq3.1.5)** on the top,
* **Step Number and Title** below the Progress Tracker,
* **Instruction Details** below the Step Number and Title,
* **Clarification Button** for requesting further clarification (**FReq3.1.7**) or more details about the current step below the Instruction Details,
* **Set Timer Button (Optional)** enabling the user to set a timer **(FReq3.1.4)** if the current cooking step involves a timed action,
* **Next Button** allowing users to move to the next step on the bottom right of the card,
* **Previous Button** allowing users to move to the previous step on the bottom left of the card.

**Freq3.1.4**: The app shall include a timer feature that users can set for cooking steps (**FReq3.1.3**) requiring specific time durations (e.g., baking, boiling). The system shall provide automatic notifications when the set time has elapsed. The timer settings shall be adjustable, and users shall be able to stop, pause, or reset the timer as needed.

**Freq3.1.5**: The app shall provide users with a visual progress tracker on top of the step cards (**FReq3.1.3**), showing which steps have been completed and which steps remain. The progress tracker shall update automatically as the user moves through each step.

**Freq3.1.6**: If a user navigates away from the cooking instructions screen (**FReq3.1.3**), the app shall save the user’s current position within the recipe and any timer settings (**FReq3.1.4**). Upon returning to the instructions, the app shall display the user’s saved position and prompt them to resume from the last completed step or restart from the beginning.

**FReq3.2.1:** Upon completion of a recipe guided by the AI assistant, the mobile app shall show the user "Share Your Dish" button that redirects them to the post sharing screen (**FReq4.4.2**) for sharing their cooked meal. The recipe field will be immutably prefilled with the recipe to complete **FReq4.4.2**.”

**FReq3.3.1:** Upon accessing the "AI Chat" section (from the navigation bar or other designated access points), the mobile app shall display a list of previous AI chat sessions. Each entry in the list shall include the following:

* Chat preview text, displaying the first few characters of the last exchange within the chat.
* Timestamp indicating when the last message was sent/received.
* An icon button to "Delete Chat" (**FReq3.3.2**). When a user selects a chat entry, the system shall redirect the user to the AI chat page **(FReq3.1.1)**.

**FReq3.3.2:** Within the AI Chat list (**FReq3.3.1**), users shall have the option to delete a specific chat session by tapping a "Delete" icon button next to the chat preview. Upon selection, the system shall display a confirmation dialog box asking, "Are you sure you want to delete this chat session?" If the user confirms, the system shall permanently delete the chat and update the chat list accordingly. If the user cancels, no changes shall occur, and the chat shall remain in the list.

**FReq4.1.1**: When a user navigates to their profile or another user's profile, the mobile app shall display a detailed profile view, including:

* Profile picture (if set by the user).
* Name.
* Count of friends which is clickable if the user is viewing his/her own profile page and should redirect the user to the friends list page **(FReq4.2.3)**.
* List of recent posts (**FReq4.4.1**).
* A "Settings" icon shall appear on the top right corner if the user is viewing their own profile, allowing access to profile settings (**FReq4.1.2**).

**FReq4.1.2:** When a user selects the "Settings" icon from their profile view, the system shall display a "Profile Settings" screen containing the following fields:

* "Name" input field, allowing the user to update their name with a minimum of 2 and a maximum of 64 characters.
* "Profile Picture" upload option allows users to upload or change their profile picture. Users can either select an image from their device gallery or capture a new one using the device camera (if supported). The system shall validate the image file format such as JPEG, PNG, and file size (maximum 5 MB), rejecting unsupported or excessively large files bigger than 5 MB. A preview of the selected image shall be displayed before saving.
* "Save Changes" button, which remains disabled until at least one field is modified. Upon saving, the system shall validate the input and update the user's profile. If validation fails, the system shall display relevant error (**EH-U1**) messages.

**FReq4.1.3:** The profile page shall display a dedicated section for achievements and badges earned by the user **(FReq5.1)**.

**FReq4.2.1**: Users shall be able to send friend requests to other users via their profile page or other designated access points (e.g., search results). Upon pressing the "Send Friend Request" button, the system shall validate the request (e.g., check for existing requests or friendships). If valid, the system shall send a friend request notification to the recipient (**FReq6.2**).

**FReq4.2.2**: Users shall receive friend requests via notifications. When a user views a friend request, the system shall present "Accept" and "Reject" buttons. If "Accept" is selected, the system shall update both users' friend lists (**FReq4.2.3**) and send a confirmation notification. If "Reject" is selected, the request shall be removed, and the sender shall not be notified.

**FReq4.2.3**: Users shall be able to view a list of their friends from their profile. The list shall display the friend's profile picture, name, and options to view their profile or remove them (**FReq4.2.4**).

**FReq4.2.4**: Users shall have the option to remove friends from their friend list. Upon selecting "Remove Friend," the system shall display a confirmation dialog box. If confirmed, the system shall update the user's friend list and remove the friend relationship.

**FReq4.3:** The system shall provide a search bar with accepting minimum of 2 and maximum of 64 characters within the "Social Feed", allowing users to search for other users by name. As the user types in the search field, a drop-down list shall display matching profiles from the foodo. Each result shall show the profile photo and name side by side. Selecting a result shall redirect the user to the selected profile.

**FReq4.4.1:** The mobile app shall display posts from other users in a social feed, sorted in reverse chronological order (most recent posts first). Each post shall include:

* User's profile picture and name.
* Post content (text, images, etc.).
* Date and time of posting.
* Interaction buttons (e.g., "Like", "Comment").

**FReq4.4.2:** Users shall be able to create posts by accessing a "Create Post" screen. This screen shall include:

* Caption input field accepting a maximum of 1907 characters.
* Recipe input field accepting a maximum of 1907 characters
* Image upload field accepting only image files such as JPEG, PNG etc. and not bigger than 10MB in size.
* "Post" button that remains disabled until at least caption or recipe field is filled, and image is uploaded.

Upon posting, the system shall validate and publish the content to the social feed.

**FReq4.4.3:** Users shall be able to edit their posts by selecting the "Edit" button from a post's options menu. The system shall display the original content in an editable format. Users can modify the caption and recipe (if not immutable recipe from AI) and then save the changes. Upon saving, the system shall validate and update the post.

**FReq4.4.4:** Users shall have the option to delete their own posts. Upon selecting "Delete Post," the system shall display a confirmation dialog. If confirmed, the post shall be permanently removed from the feed.

**FReq4.5.1:** Users shall be able to "Like" posts by pressing a "Like" button.

**FReq4.5.2:** If a user has liked a post, they shall be able to "Unlike" it by pressing the same button.

**FReq4.6.1:** Users shall be able to add comments to posts by using the comment icon button. The system shall display a comment input field, when pressed on the comment icon button, allowing a maximum of 256 characters below each post. Upon submission, the system shall validate and add the comment to the post's comment section.

**FReq4.6.2:** Users shall have the option to edit their own comments. Upon clicking the "Edit" icon button on the comment the system shall display the original comment in an editable format. Once changes are saved, the system shall validate and update the comment.

**FReq4.6.3:** Users shall have the option to delete their own comments. Upon clicking the "Delete" icon button the system shall display a confirmation dialog. If confirmed, the comment shall be removed.

**FReq5.1:** The mobile app shall provide a dedicated section within the user's profile to view all earned achievements and badges. This section shall include the following features:

* A grid or list view displaying each earned badge, with a visual icon, badge name, and brief description.
* Clicking on a badge shall display additional details, including when and how the badge was earned, milestones achieved (e.g., "5 Recipes Completed" for Recipe Mastery).
* The app shall notify (**FReq6.2**) users when they earn a new achievement. Users shall be able to view details of the new achievement through the notification.

**FReq5.2:** The app shall automatically award achievements to users based on predefined criteria and milestones. Examples include:

* Recipe Completion Milestones: Users shall earn badges based on the number of completed recipes (e.g., 5, 10, 25, 50 recipes). Upon reaching each milestone, the system shall display a new achievement badge within the "Recipe Mastery" category.
* Post Sharing Milestones: Users shall earn badges for sharing posts within the app. Each milestone (e.g., 10, 20, 50 posts shared) shall grant a unique achievement badge, displayed in the "Social Sharing" category.
* Category-Based Recognition: Achievements shall be grouped by types such as "Recipe Mastery," "Social Sharing," and "Ingredient Explorer," with badges awarded based on activity and proficiency in specific areas.
* Notifications shall be triggered whenever a user earns a new badge, appearing in the app’s notification center and including details of how and when the achievement was earned. Users shall be able to access these details from the achievements section of their profile.

**FReq6.1:** Users shall be able to view notifications related to friend requests, post interactions, earned achievements, etc., on a dedicated "Notifications" page.

**FReq6.2:** The system shall send notifications to users based on interactions, such as friend requests and post likes as a notification item in a list with a “Delete” icon button **(FReq6.3)**.

**FReq6.3:** Users shall have the option to delete individual notifications by clicking a "Delete" icon on the right side of the notification item.

**FReq6.4:** In the upper right corner of the notifications page there shall be a trash can icon for deleting all notifications, and when the user clicks on this button, the system shall give the user a confirmation dialog asking if they are sure they want to delete all notifications in the notification box. If the user chooses to delete all notifications, all the notifications stored for the user shall be deleted. If the user decides not to delete the notifications, there shall be no change in the notifications.

**FReq7.1**: If a user interacts with the AI assistant for help during a recipe, the app shall send a feedback message after the interaction, asking whether the AI response was helpful. Options shall include “Helpful,” “Somewhat Helpful,” and “Not Helpful,” along with an optional comment field for additional feedback. This input shall be used to improve the accuracy and relevance of AI responses.

**FReq7.2**: Upon completing a cooking session, the app shall display a summary feedback pop-up allowing users to rate their overall experience with the recipe. This summary shall prompt users to rate multiple aspects, such as taste, cooking instructions, and ingredient availability, to gather comprehensive feedback on the entire experience.

1. **Non-Functional Requirements**

**Usability requirements**

**UR1:** Notifications, including those for timed cooking steps, alerts, and achievements, must be displayed within **3 seconds** of the **%80** of the events to ensure clarity for the user.

**UR2:** Essential tasks, such as interacting with the AI assistant and accessing social features, should be accessible within **five taps** from the home screen.

**Performance requirements**

**PR1:**Social actions (e.g., likes, comments), shall be processed in under **5 seconds** for **90%** of **25** requests, maintaining a high level of responsiveness.

**PR2:** The AI cooking assistant shall answer the prompts in **under 15 seconds** after receiving input supporting smooth and prompt user engagement.

**PR3:**User-generated content, such as images in the social feed, shall load within **5 seconds** for **90%** of **20 requests**, ensuring smooth content interaction.

**PR4:** Push notifications (cooking reminders, social alerts) shall be delivered within **10 seconds** of the triggered event for **95%** of cases, verified through real-world testing with **2 devices**.

**PR5:** Image uploads up to **5 MB** shall complete within **20 seconds**, and images shall display in the feed within **20 seconds**, tested over **30 uploads** under average network conditions.

**Software system attributes**

**a) Reliability**:

**SA-R1:** The system shall maintain **95%** uptime for essential functions (user authentication, AI assistance) over an **8-hours** period, monitored using uptime tracking tools.

**b) Availability:**

**SA-A1:** The system shall maintain an availability level of **95%** in **8 hours**, minimizing the risk of downtime.

**c) Security:**

**SA-S1:** The application shall implement a valid SSL/TLS certificate for HTTPS connection for **%100** of the endpoints that transmit data between the client and server.

**d) Maintainability:**

**SA-M1:** The codebase shall be organized into modules (AI processing, user management, etc.), with each module documented. Code reviews shall confirm adherence to modular design.

**e) Portability**

**SA-P1:** The app shall function on devices running **iOS 13+** and **Android 12+**, verified through testing on **at least 2 devices** for each platform.

**Constraints**

**C1: Network Dependency:** foodo’s core features, such as the AI cooking assistant and social network, require reliable internet access to function fully. Offline access may be limited, and the application is thus constrained by the availability and stability of network connectivity, impacting user experience in low or no connectivity environments.

**C2: Resource Availability:** The project relies on cloud infrastructure (e.g., AWS) and third-party AI models, meaning the application is dependent on the continued availability and performance of these resources. Any service limitations, cost increases, or policy changes in cloud or AI providers can impact foodo’s functionality, requiring potential adjustments in design or feature availability.

**C3: Development Team Availability and Schedule:** Given the part-time status of the development team members and overlapping academic and work commitments, time availability is constrained. This limits development capacity, potentially requiring prioritization of certain features or adjustments in timelines to meet project milestones.

**C4: Language and Localization:** The initial launch will be limited to English, as full multilingual support is out of scope due to time and resource constraints. This limits the immediate target audience to English-speaking users, though additional languages may be considered for future iterations.

**C5: Budget Constraints:** The project operates with an official budget of zero. However, to enhance the project beyond basic functionality, team members are using their own pocket money and salaries. This constraint means we rely heavily on free resources, student versions, and low-cost tools. Limited funding impacts our choices in development tools, cloud infrastructure, and third-party services, potentially restricting initial scalability and access to advanced features. Despite these limitations, we aim to make strategic, cost-effective decisions to ensure quality and functionality within our financial boundaries.

**Error Handling Requirements.**

This section outlines how foodo shall respond to various error types to ensure reliability and user experience. The following error-handling mechanisms address both user-driven errors and system malfunctions.

**EH-U1: Input Validation Errors**

**Description**: If a user inputs data that does not match expected formats (e.g., special characters in names, invalid email formats, passwords that do not meet complexity requirements).

* **Response**: The system shall display a clear, contextual error message within 2 seconds, guiding the user to correct the input.
* **Mechanism**: The system shall validate input fields in real-time where applicable and provide immediate feedback. All invalid inputs shall be logged for analytics purposes.
* **Example**: "Invalid email address format."

**EH-U2: Authentication Errors**

**Description**: Errors encountered during user authentication, such as incorrect login credentials or non-existent accounts.

* **Response**: The system shall display a clear, contextual error message within 2 seconds, providing appropriate guidance (e.g., re-entering credentials or creating a new account).
* **Mechanism**: The system shall validate login credentials against the Clerk Authentication Service. Unsuccessful authentication attempts shall be logged, with a limit on the number of attempts allowed before a cooldown period.
* **Example**: "Invalid email or password. Please try again."

**EH-S1: System Errors**

**Description**: Errors related to server connectivity, database access, or unexpected internal issues that prevent normal operation.

* **Response**: The system shall display a non-technical, user-friendly error message and attempt to recover gracefully where possible (e.g., retrying connection). The user shall be informed of the issue within 2 seconds.
* **Mechanism**: System errors shall be logged with detailed information for further investigation. Alerts may be triggered for administrators in case of critical issues.
* **Example**: "Unable to connect to the server. Please check your internet connection and try again."

**EH-AI1: AI Interaction Errors**

**Description**: Errors encountered during interactions with the AI system, including invalid or unclear requests, unsupported inputs, or failures in data processing (e.g., missing critical details like meal names or unrecognized commands).

* **Response**: The system shall display a user-friendly error message within 2 seconds, clearly explaining the issue and providing actionable guidance for correction or refinement of the input.
* **Mechanism**: The system shall validate user inputs through the LLM Provider and internal function checks. Invalid or incomplete requests shall trigger an error response and be logged for analytics and monitoring. In case of unrecognized inputs, the system shall suggest possible corrections or clarifications.

**Example**: "Your input could not be processed. Please provide more details or clarify your request."

1. **Software Increments**

**Increment 1 (Estimated Effort: 365 person-hours):**

In the **first increment**, the development will focus on establishing the application's foundational features for user access, navigation, and initial content management.

1. **User Access**: The user authentication system will be implemented, allowing users to log in and recover forgotten passwords. A robust user registration feature will also enable new users to create accounts securely.
2. **Navigation**: Users can navigate across the application through a structured page navigation system, ensuring intuitive and smooth transitions between different sections of the application.
3. **Social Interaction**:
   * Users can view posts across the platform.
   * Post-management will be partially implemented, covering 40% of the planned functionalities. This includes essential actions like creating posts but not yet complete editing or deletion capabilities.
4. **Initial AI Implementation**:
   * Initial work on **meal suggestions** and **recipe instructions** will be completed (30% progress). While users may not yet experience fully dynamic suggestions or detailed instructions, this work will establish the necessary backend and interface integrations for future increments.
   * The AI chat interface, a key aspect of the application, will reach 60% completion. Users can access and partially manage chat lists, preparing the system for advanced interactions in the following increments.

This increment will prioritize technical stability and usability by configuring essential backend services, setting up database tables, and creating API integrations for user authentication, navigation, and data handling.

**Increment 2 (Estimated Effort: 345 person-hours):**

The **second increment** will expand on the foundational features by enhancing user interactivity and introducing notifications and user search functionalities. Users will:

* Fully view their profiles with the implementation of the profile viewing feature.
* Receive notifications (40% progress), allowing them to stay updated on key events.
* Search for other users efficiently by completing the search feature.

This increment will also progress the meal suggestions and recipe instruction feature to 40%, refining their functionality to provide more personalized outputs. Notifications and user searches will use a database-backed service layer, and the front end will integrate with these capabilities.

**Increment 3 (Estimated Effort: 345 person-hours):**

In the **third increment**, the application will focus on extending user engagement and implementing social features:

* The achievement system will be introduced, allowing users to view and begin earning achievements (20% completion).
* Notifications will advance to 80% completion, including functionality for managing notifications.
* Meal suggestions and recipe instructions will achieve 60% completion, offering enhanced personalization and usability.
* Friend management will be fully implemented, enabling users to send and receive friend requests and establish social connections.
* The AI chat system will reach 70% completion, offering improved interactivity.
* Achievements and notifications will be integrated using an event-driven architecture, with message brokers such as Kafka facilitating asynchronous updates for real-time user experiences.

**Increment 4 (Estimated Effort: 345 person-hours):**

The **fourth increment** will focus on refining the user experience:

* AI chat features will be fully operational, including viewing and managing chat lists.
* The achievement system will reach 100% completion, and meal suggestions and recipe instructions will reach 70%.
* Users can interact with posts, such as liking and commenting (40% completion).

This increment will focus on enhancing backend scalability for real-time interactions and integrating UI elements for user engagement.

**Increment 5 (Estimated Effort: 345 person-hours):**

By the **fifth increment**, the application will approach feature completion:

* Meal suggestions and recipe instructions will be 90% complete, providing near-finalized functionality.
* Notifications and post interactions will be fully implemented, allowing users to send and receive notifications, like, edit, or delete posts.
* Push notifications will be implemented using Firebase Cloud Messaging, enabling real-time updates and improved user engagement.
* Friend lists and profile management will be finalized, offering users a complete social experience.

The backend will implement advanced notification routing and support for CRUD operations on user-generated content.

**Increment 6 (Estimated Effort: 345 person-hours):**

The **final increment** will deliver a feature-complete application:

* Meal suggestions and recipe instructions will reach 100%, offering polished functionality.
* Users will be able to share experience feedback, interact fully with posts, and explore advanced social and AI-driven features.
* Feedback from users will be analyzed, providing insights into user behavior and system performance.

This increment will finalize all functionalities, ensuring the application is ready for deployment with a fully integrated backend, optimized database schemas, and a responsive user interface.

Each increment builds upon the previous one, ensuring a structured and scalable development approach while progressively delivering features to enhance user experience.

1. **Frameworks, Libraries, Services, Databases and APIs**

* **Clerk:** Handles user authentication, authorization, and accountability for the foodo application. It will be implemented in the 1st increment to enable secure user login and registration features
* **React Native:** A framework for building cross-platform mobile applications. It is the main framework we are going use for mobile development. It will be introduced in the 1st increment and will be used throughout the project.
* **Spring Framework:** It is a popular backend framework for building and managing foodo’s Social Media Service. It will utilize Java and Spring Boot. It will be introduced in the first increment.
* **Nest.js:** It is progressive Node.js framework for building efficient, reliable, and scalable server-side applications. foodo will utilize Nest.js for Chat Service and Users Service. It will be introduced in the first increment.
* **Expo:** Itsimplifies the development and deployment process for the React Native mobile application with tools for testing, debugging, and over-the-air updates. It will be introduced in the first increment.
* **OpenAI API:** Acts as the primary LLM provider for processing recipe instructions and generating meal suggestions. It will be introduced in the first increment.
* **QDrant:** A vector database that stores AI embeddings for retrieving contextually relevant results, such as meals and receipes.
* **Docker:** It providescontainerization for backend services to ensure consistency across our microservice architecture. It will be introduced in the first increment.
* **PostgreSQL:** Acts as the primary relational database management system (RDBMS) for storing structured data. It will be introduced in the first increment and will be used in every backend service.
* **Nginx:** It will act as a reverse proxy. It will be the gateway between clients and foodo. It will handle secure connection with SSL certificate. It will be configured in the 1st increment to distribute traffic among microservices.
* **AWS EC2:** An EC2 instance will host Docker containers running microservices and databases, ensuring reliable cloud infrastructure. It will be set up in the first increment**.**
* **Kafka:** Kafka is employed for managing real-time data streams, ensuring reliable communication between services and enabling event-driven architecture. It will play a key role in handling notifications and achievement tracking. It will be introcued in second increment and will be fully implemented in third increment.
* **AWS Lambda:** It executes serverless backend functions. It will be used for LLM calls to different providers. We will introduce it at first increment.
* **AWS S3:** User-generated content (e.g., profile pictures, shared meal images) is stored in AWS S3 buckets. S3’s pay-as-you-go model ensures cost-effective storage, especially during the early stages of the project with limited budgets. It will be integrated at the first increment.
* **GitHub Actions:** GitHub Actions workflows will enable automated deployment to cloud environments AWS EC2.
* **ESLint:** It enforces coding standards for JavaScript/TypeScript. It will be implemented in the first increment.
* **Prettier:** It ensures consistent code formatting. It will be implemented in the first increment.
* **Junit:** It is a testing library framework. It will be used to do unit tests in Social Media Service. It will be introduced in the first increment.
* **Postman:** It will be utilized to perform API calls to test backend services. We will start using it starting from first increment.
* **Google Vertex AI:** It supports training and deployment of AI models. When it’s going to be implemented is not certain yet.
* **Claude API:** Acts as a LLM provider for processing recipe instructions and generating meal suggestions. It will be implemented in the 2nd increment and beyond.

# High-Level (Architecture) Design

1. **Selected Architecture**

The architecture chosen for the development of **foodo** is **Microservice Architecture**, a decision made after careful consideration and aligned with the non-functional requirements outlined in Section 2.2, particularly regarding scalability, reliability, maintainability, and performance. While a monolithic architecture might seem like a more familiar and straightforward choice for a project of this scale, we have been inspired by our exposure to microservice architectures through our professional experiences, including internships and part-time work. This exposure has provided us with valuable insights into the benefits and challenges of microservices. Acknowledging our advisor's concerns about the complexity involved, taking on this architecture for our senior project offers a unique opportunity to enhance our knowledge and abilities in implementing a modern, scalable architecture. Combining our enthusiasm with prior experience, we aim to deliver a dependable, feature-rich application while solidifying our microservices expertise.

**Justification for Microservice Architecture**

**Scalability:** The microservice architecture allows individual services (e.g., AI Assistant, Social Media, Notifications) to scale independently based on their workload. For instance:

* The AI Chatbot Service can handle high computational loads by deploying more instances during peak hours without affecting other services like Profile Management.
* AWS Lambda and AWS S3 integrations ensure that computational and storage scalability are seamlessly achieved.
* NFR Impact: Meets PR1 and PR2 by handling high traffic volumes for AI interactions and content uploads efficiently.

**Reliability:** By isolating services, the failure of one service (e.g., Notification Service) does not impact the functionality of others (e.g., User Authentication). This is crucial for maintaining the application's 95% uptime requirement over 8-hour periods.

* **NFR Impact**: Meets SA-R1 and SA-A1.

**Maintainability:** The modularity of microservices ensures that each service can be developed, tested, and deployed independently. This is particularly important given the team’s part-time availability and academic commitments.

* The Social Media Service team can focus on optimizing post interactions while another team works on Achievement Tracking. This modularity reduces interdependence, supporting incremental updates and faster issue resolution. For instance, the Social Media Service team can focus on optimizing post interactions while another team works on Achievement Tracking.
* **NFR Impact:** Meets SA-M1 by ensuring the codebase remains modular and maintainable.

**Performance:** Each microservice is optimized for its specific task and runs in isolated environments (e.g., Docker containers). This ensures faster response times for critical interactions, such as AI prompt processing and social feed updates.

* Faster response times for AI prompts and real-time notifications are achieved by scaling resource-heavy services like AI Assistant independently.

**NFR Impact:** Supports PR1, PR2, and PR4 by reducing latency in core interactions and real-time notifications.

**Advantages of Microservice Architecture**

**Flexibility in Technology Stack:** Each service can use the most suitable framework or language. Allows using the most suitable frameworks for each service, such as React Native for mobile, Nest.js for User Services, and Spring Boot for social media. For example:

* **React Native** for mobile app development.
* **Nest.js** for user services and **Spring Boot** for social media functionalities.
* **PostgreSQL** for structured data storage and **QDrant** for vectorized AI data.

**Ease of Continuous Deployment:** With **GitHub Actions** and containerization, microservices allow for faster updates and independent deployments without downtime. Independent service deployments minimize downtime and accelerate updates.

**Resilience:** Service failures are contained, preventing cascading issues across the application. Isolated failures prevent overlapping issues, ensuring system stability.

**Disadvantages of Microservice Architecture**

**Higher Initial Setup Effort:** Setting up microservice architecture requires more time and expertise compared to monolithic designs. However, this is a one-time effort justified by long-term scalability and maintainability.

**Inter-Service Communication Overhead:** Microservices depend heavily on APIs for communication. Latency or failures in communication can impact the overall performance. Using **Nginx** as a reverse proxy and **Kafka** for asynchronous message handling reduces this risk.

**Data Management Challenges:** Data consistency across services can be challenging. This is managed using **PostgreSQL** for transactional operations and **AWS S3** for static content storage.

**Development Process Impact**

**Incremental Development:** The modularity of microservices ensures that features can be developed and deployed incrementally, aligning with the defined increments.

**Resource Utilization:** Docker-based containerization minimizes resource consumption, enabling development even under budget constraints.

**Team Collaboration:** The architecture enables parallel development, where team members can focus on specific services independently.

While microservice architecture introduces additional complexity in management and inter-service communication, these challenges are outweighed by the benefits of scalability, reliability, and maintainability. By leveraging modern tools like Docker, Kafka, and AWS, foodo's architecture ensures a robust and future-ready foundation.

1. **Logical View**

A group of icons with text

Description automatically generated with medium confidence

Figure 2. Block Diagram of foodo Software Architecture

A diagram of a computer

Description automatically generated

Figure 3. EER Diagram

1. **Process View**

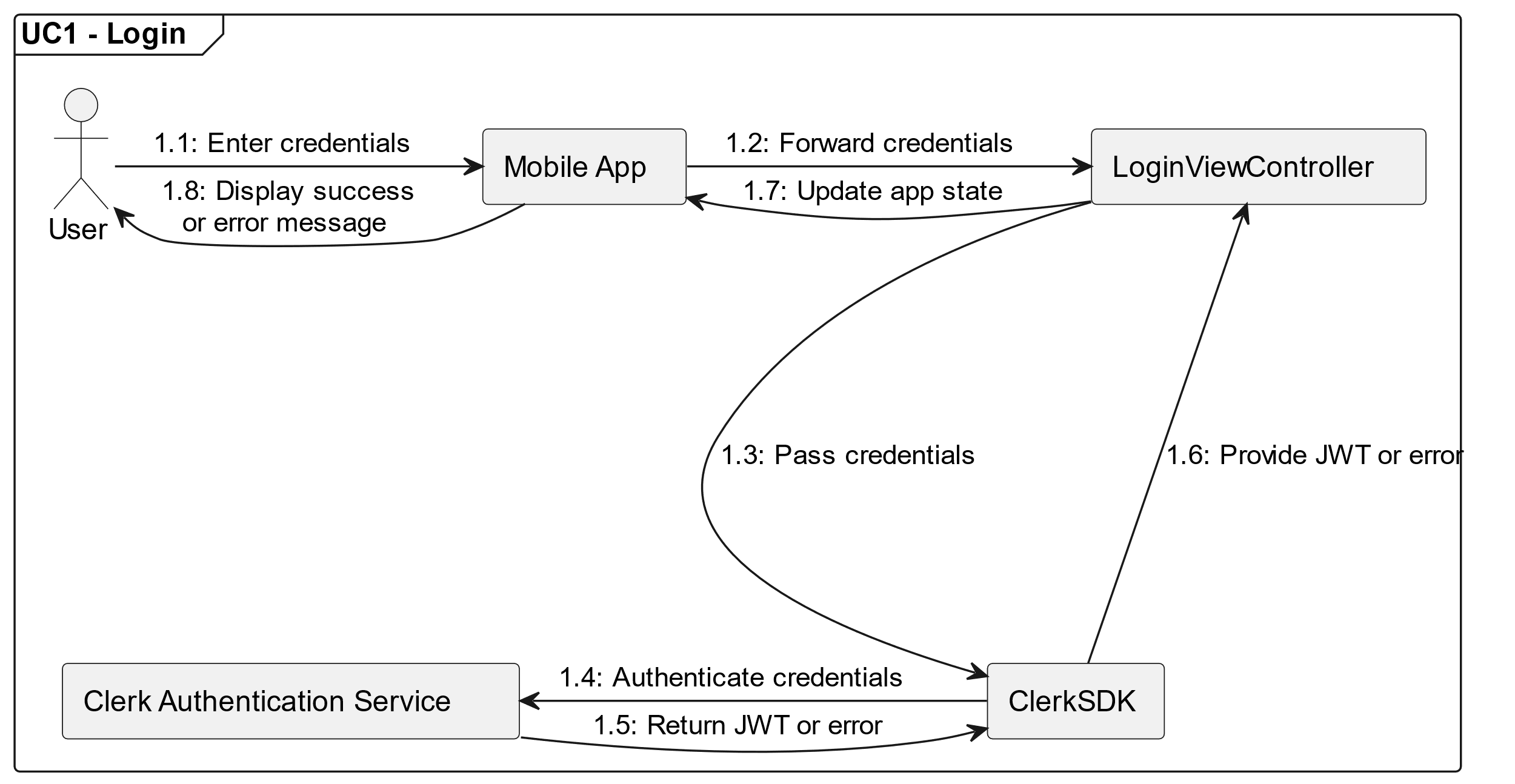


Figure 4. UML Communication Diagram of UC1

**Use Case Name: UC1 – Login**

**Description:** The user logs into the mobile app by providing credentials. The mobile app forwards these credentials to Clerk Authentication Service. If the credentials are valid, a JSON Web Token (JWT) is returned to the app, confirming authentication.

**Actors:**

* **User:** The individual using the mobile app to log in.
* **Clerk Authentication Service:** An external service handling authentication.

**Participating Classes/Objects:**

* **LoginViewController (Mobile App):** Handles the login screen UI and collects user credentials.
* **ClerkSDK:** Provides an abstraction layer for communicating with the Clerk Authentication Service.
* **Clerk Authentication Service:** Verifies credentials and issues a JWT token.

**Flow of Events**:

1. **User Action**:

* The user opens the login screen on the mobile app and enters their username and password.

1. **Interaction with ClerkSDK**:

* The LoginViewController collects the credentials and passes them to the ClerkSDK.

1. **Authentication Process**:

* The ClerkSDK securely communicates with the external Clerk Authentication Service.
* The Clerk Authentication Service verifies the credentials.
* If valid, the service generates a JWT and sends it back to the ClerkSDK.
* If invalid, an error response is returned.

1. **Response to Mobile App**:

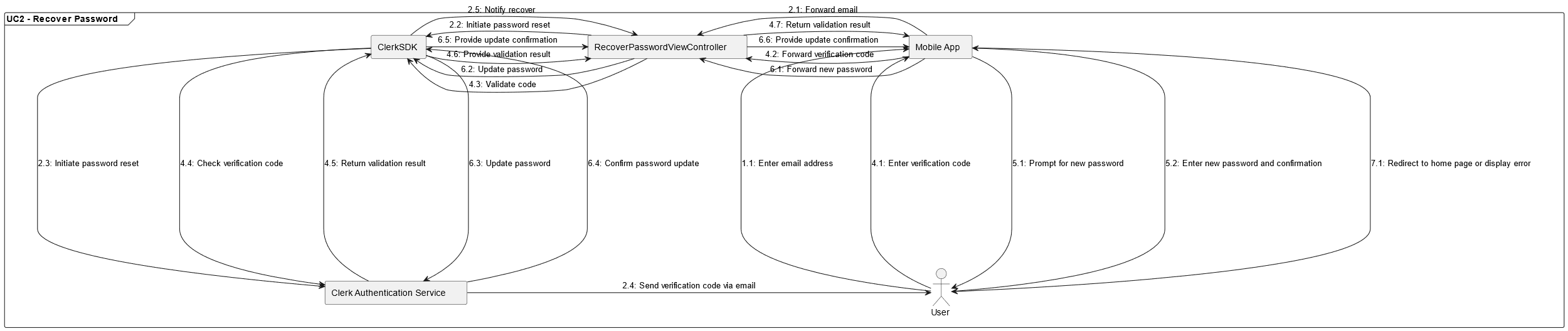
* On success:
  + The ClerkSDK provides the JWT to the LoginViewController.
  + The app securely stores the JWT.
  + The app transitions to the home page.
* On failure:
  + The ClerkSDK returns an error response to the LoginViewController.
  + The app displays an appropriate error message to the user.

Figure . UML Communication Diagram of UC2

**Use Case Name: UC2 – Recover Password**

**Description:** The user initiates a password recovery process through the mobile app. The mobile app uses **ClerkSDK** to interact with the external Clerk Authentication Service to send a verification code via email. The user inputs the verification code in the app, and the app validates it through ClerkSDK. If valid, the user is prompted to set a new password. Once the new password is confirmed and updated via ClerkSDK, the user is redirected to the home page.

**Actors:**

* **User:** The individual requesting a password reset.
* **Clerk Authentication Service:** The external service responsible for sending verification codes, validating them, and updating passwords.

**Participating Classes/Objects:**

* **RecoverPasswordViewController (Mobile App):** Handles UI and collects user inputs.
* **ClerkSDK:** Provides an abstraction layer for communicating with the Clerk Authentication Service.
* **Clerk Authentication Service:** Sends verification codes, validates them, and updates the user’s password.

**Flow of Events**:

1. **User Action**:

* The user opens the "Forgot Password" screen and enters their email address.

1. **Send Password Reset Request**:

* The RecoverPasswordViewController forwards the email to the ClerkSDK.
* ClerkSDK communicates with the Clerk Authentication Service to initiate the password reset.
* The Clerk Authentication Service sends a verification code via email to the user.

1. **Prompt for Verification Code**:

* The app transitions to a screen prompting the user to input the verification code.

1. **Validate Verification Code:**

* The user enters the verification code.
* The RecoverPasswordViewController passes the code to ClerkSDK.
* ClerkSDK validates the code with the Clerk Authentication Service.

1. **Prompt for New Password**:

* If the verification code is valid, the app transitions to the "Set New Password" screen.
* The user inputs a new password and confirms it.

1. **Update Password**:

* The RecoverPasswordViewController sends the new password (and confirmation) to the ClerkSDK.
* ClerkSDK updates the password through the Clerk Authentication Service.

1. **Completion**:

* If the password update is successful, the app redirects the user to tsshe home page.
* If there’s an error, an appropriate message is displayed.

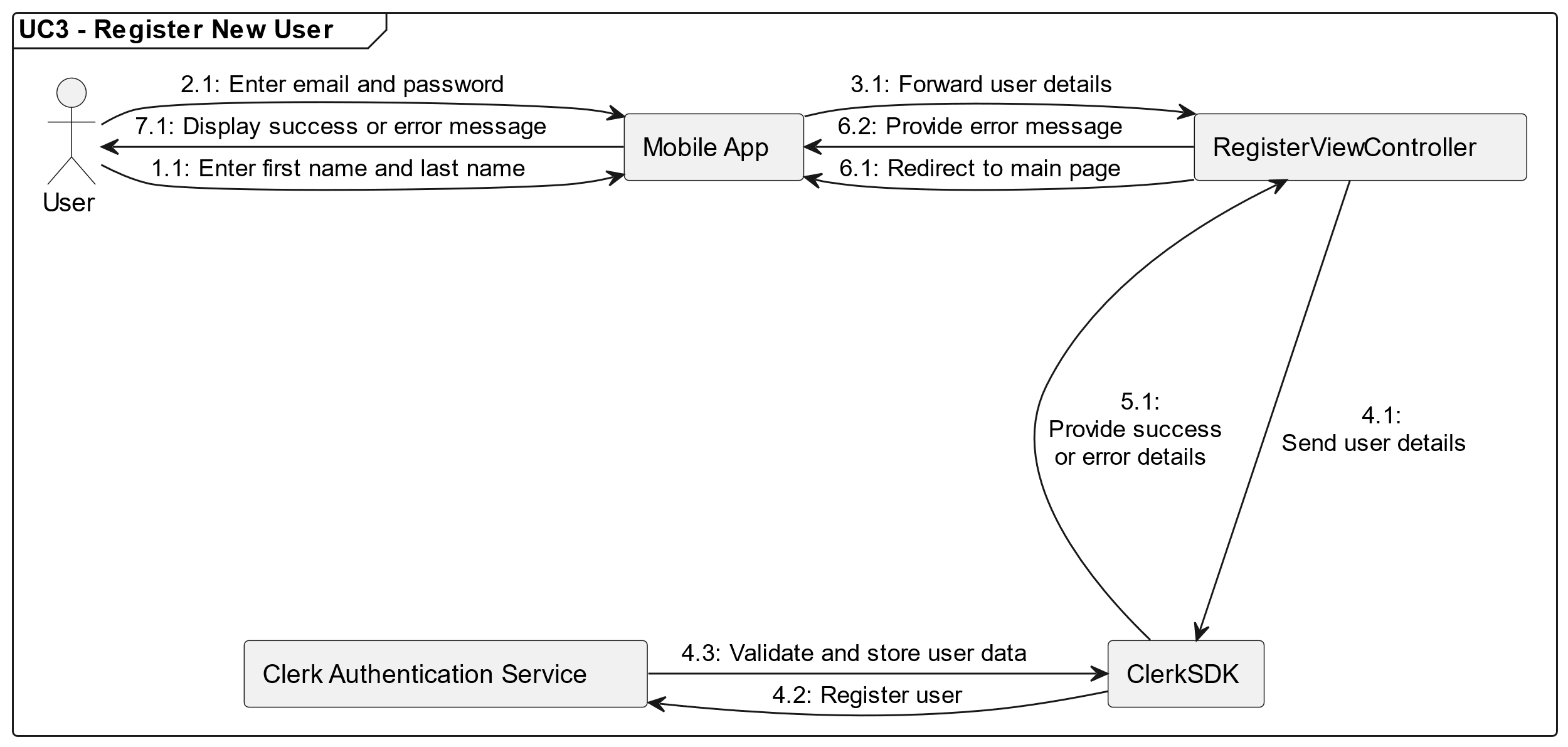


Figure 6. UML Communication Diagram of UC3

**Use Case Name: UC3 – Register New User**

**Description:** The user registers for an account by providing their first name, last name, email, and password in the mobile app. The app uses **ClerkSDK** to send these details to the external Clerk Authentication Service. If the registration is successful (no errors are returned by Clerk), the user is automatically logged in and redirected to the main page. If an error occurs, it is displayed to the user.

**Actors:**

* **User:** The individual registering for an account.
* **Clerk Authentication Service:** The external service responsible for creating the user account.

**Participating Classes/Objects:**

* **RegisterViewController (Mobile App):** Handles UI for input fields and manages user interactions during the registration process.
* **ClerkSDK:** Provides an abstraction for sending user data and handling registration with the Clerk Authentication Service.
* **Clerk Authentication Service:** Validates and stores user account data.

**Flow of Events**:

1. **User Action - Enter Personal Details**:

* On the first registration screen, the user inputs their first name and last name.

1. **User Action - Enter Credentials**:

* On the second screen, the user enters their email and password.s

1. **Send Registration Request**:

* The RegisterViewController collects all the user inputs (first name, last name, email, and password) and sends them to the ClerkSDK.

1. **Clerk Registration Process:**

* ClerkSDK communicates with the Clerk Authentication Service to register the user.
* The Clerk Authentication Service validates the data.

1. **Response Handling:**

* If no errors are returned by Clerk:
  + ClerkSDK confirms successful registration.
  + The RegisterViewController redirects the user to the main page.
* If an error occurs (e.g., invalid email format, weak password, or duplicate account):
  + ClerkSDK provides the error details to RegisterViewController.
  + The app displays the error message to the user.

1. **Completion:**

* On successful registration, the user is automatically logged in and redirected to the main page.
* On failure, the user can correct the input and retry.

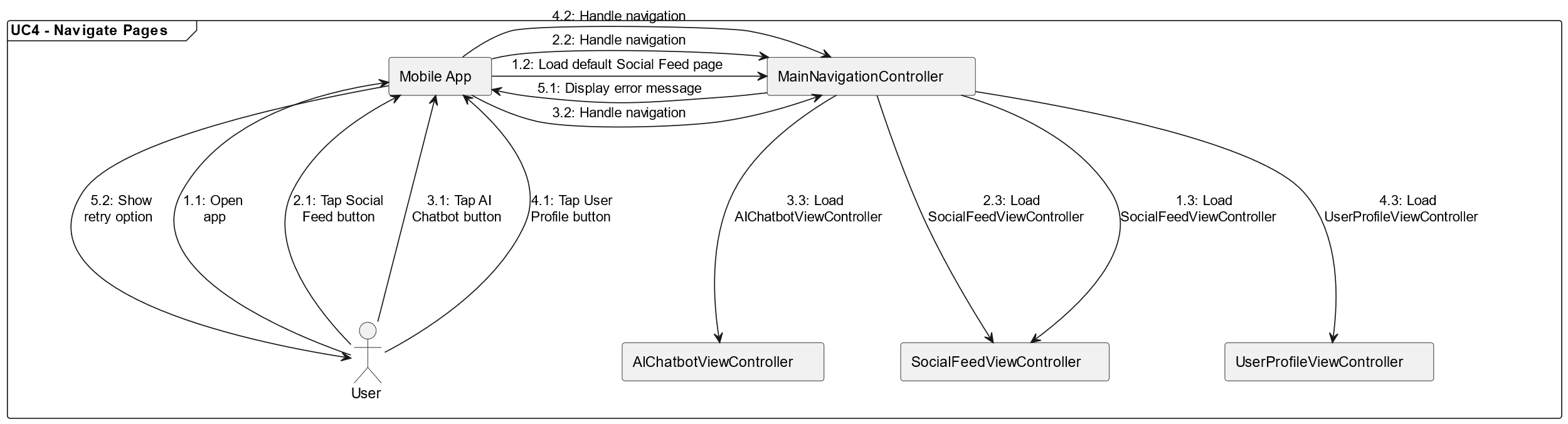


Figure 7. UML Communication Diagram of UC4

**Use Case Name: UC4 – Navigate Pages**

**Description:** The user navigates between various pages of the mobile app using the navigation bar at the bottom of the screen. Each button on the navigation bar leads to a specific page: Social Feed, AI Chatbot, or User Profile. Additional navigation options, such as accessing notifications or settings, are available depending on the page.

**Actors:**

* **User:** The individual navigating through the app.

**Participating Classes/Objects:**

* **MainNavigationController (Mobile App):** Manages the navigation bar and transitions between pages.
* **SocialFeedViewController:** Displays the social feed page, including a search bar and notifications.
* **AIChatbotViewController:** Displays the AI Chatbot page with options for managing chats.
* **UserProfileViewController:** Displays the user profile page, including a settings button if it's the user’s own profile.

**Flow of Events**:

1. **Initial View**:

* When the user opens the app, the MainNavigationController displays the navigation bar at the bottom of the screen.
* The default page (e.g., Social Feed) is loaded.

1. **User Action - Enter Credentials**:

* The user taps the **Social Feed** button on the navigation bar.
* MainNavigationController loads the SocialFeedViewController
* The SocialFeedViewController displays:
  + A search bar at the top for finding other users.
  + A notification icon for accessing notifications.

1. **Navigate to AI Chatbot:**

* The user taps the AI Chatbot button on the navigation bar.
* MainNavigationController loads the AIChatbotViewController.
* The AIChatbotViewController displays:
  + A top navigation bar with options to open the chat list or delete conversations.

1. **Navigate to User Profile:**

* The user taps the **User Profile** button on the navigation bar.
* MainNavigationController loads the UserProfileView-Controller.
* The UserProfileViewController displays:
  + The profile information of the user.
  + If the profile belongs to the logged-in user, a **Settings** button appears in the top-right corner for accessing profile settings.

1. **Error Handling:**

* If a page fails to load due to a network issue or other error, the app displays an error message and allows the user to retry.

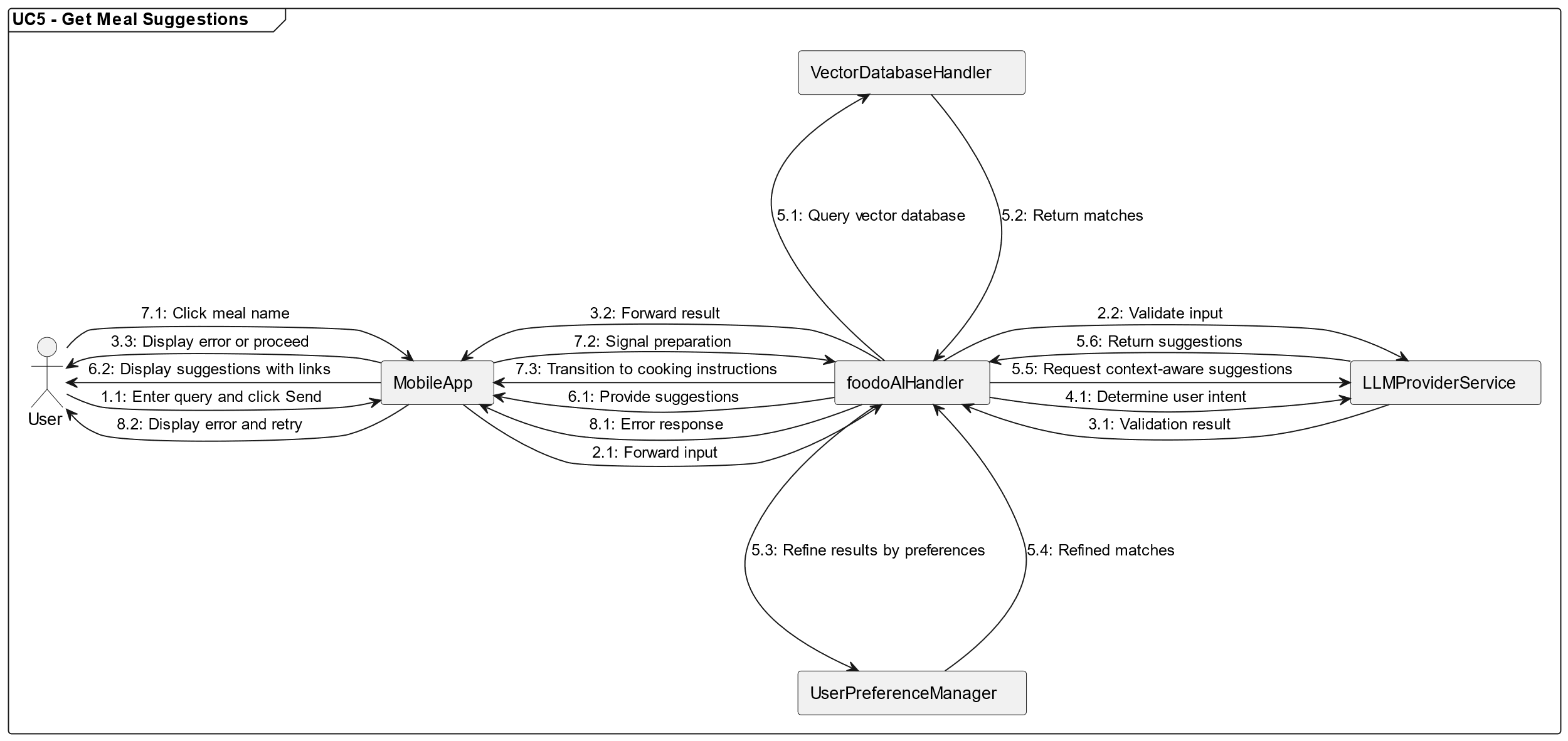


Figure 8. UML Communication Diagram of UC5

**Use Case Name: UC5 – Get Meal Suggestions**

**Description:** The user interacts with the AI assistant in the mobile app to receive meal suggestions. User inputs are sent to **foodo AI** for processing and undergo validation. The system identifies the user’s intent and either provides meal suggestions based on their preferences and the Vector Database or prompts for more details. If meal suggestions are found, the user can click on meal names for more information or begin cooking instructions.

**Actors:**

* **User:** The individual requesting meal suggestions.
* **LLM Provider:** Validates input and generates context-aware suggestions.

**Participating Classes/Objects:**

* **MobileApp:** The overarching system that coordinates user interactions, sends requests, and processes responses.
* **MealSuggestionViewController (Mobile App):** Handles user input, manages interactions with backend services, and displays results or error messages.
* **foodoAIHandler:** Facilitates communication between the mobile app and foodo AI, processing responses and managing errors.
* **LLMProviderService:** Validates user inputs, identifies intent, and provides refined, context-aware suggestions.
* **VectorDatabaseHandler:** Queries vectorized meal data to find relevant matches.
* **UserPreferenceManager:** Applies user-specific preferences (e.g., dietary restrictions and cooking level) to refine meal suggestions.

**Flow of Events**:

1. **User Action - Input Text:**

* The user enters a query (1–1907 characters) in the input box and clicks the "Send" button.

1. **Send Input for Validation**:

* The MealSuggestionViewController sends the input to the MobileApp, which forwards it to foodoAIHandler.
* foodoAIHandler sends the input to the LLMProviderService for validation.

1. **Validation Result:**

* If the input is invalid, the MealSuggestionViewController displays an error message.
* If the input is valid, the system proceeds to intent determination.

1. **Determine User Intent:**

* The system evaluates whether the user’s input is requesting meal suggestions or detailed recipes.

1. **Process Suggestions Request:**

* For meal suggestions:
  + The foodoAIHandler vectorizes the input and queries the VectorDatabaseHandler for similar matches (similarity score > 0.5).
  + If no matches are found, the app prompts the user to provide more details.
  + If matches are found:
    - The UserPreferenceManager refines the results based on user preferences.
    - The system forwards the refined results to the LLMProviderService for context aware suggestions.

1. **Display Suggestions:**

* Meal suggestions are displayed with meal names as clickable links.

1. **Click on Meal Name:**

* Clicking a meal name sends a predefined message to foodo AI, signaling the user’s intent to prepare the meal.
* The process transitions to the cooking instructions phase.

1. **Error Handling:**

* If foodo AI or any backend service returns an error, the MealSuggestionViewController displays an appropriate error message and prompts the user to retry.



Figure 9. UML Communication Diagram of UC6

**Use Case Name: UC6 – Get Recipe Instructions**

**Description:** The user interacts with the AI assistant in the mobile app to obtain detailed cooking instructions for a specific meal. The app validates user input using the LLM Provider, identifies the user’s intent, and queries the Vector Database to find and retrieve recipes. The LLM Provider also generates detailed recipe instructions from the retrieved data. If a valid recipe is found, the instructions are displayed in an interactive, card-based format with features like step tracking, timers, and navigation. Progress is saved automatically if the user navigates away, allowing them to resume or restart later.

**Actors:**

* **User:** The individual requesting recipe instructions.
* **LLM Provider:** Validates user input and generates structured recipe instructions from retrieved data.

**Participating Classes/Objects:**

* **MobileApp:** Coordinates interactions between the user and backend services.
* **RecipeInstructionViewController (Mobile App):** Manages UI and user interaction during the recipe instruction process.
* **foodoAIHandler:** Orchestrates communication between the mobile app, LLM Provider, and Vector Database.
* **LLMProviderService:** Validates user input, identifies intent, and generates recipe instructions from data retrieved from the Vector Database.
* **VectorDatabaseHandler:** Searches for similar recipes based on vectorized meal data.
* **TimerManager:** Manages timers for cooking steps and sends notifications.
* **ProgressTracker:** Tracks the user’s progress through recipe steps and stores session data.

**Flow of Events**:

1. **User Action- Input Recipe Request:**

* The user types a request in the input box (1–1907 characters) and clicks the "Send" button.

1. **Send Input for Validation**:

* The RecipeInstructionViewController forwards the input to the MobileApp, which sends it to foodoAIHandler.
* foodoAIHandler sends the input to the LLMProviderService for validation.

1. **Validation Result:**

* If the input is invalid, the RecipeInstructionViewController displays an error message.
* If valid, the system determines the user’s intent.

1. **Determine User Intent:**

* If the user requests detailed recipes, the system checks if a valid meal name is included in the input.
* If no meal name is found, the app prompts the user to provide one.

1. **Query Vector Database for Recipes:**

* The system vectorizes the meal name and queries the VectorDatabaseHandler to find recipes with a similarity score above 0.8.
* If no matches are found, the user is prompted to provide more details.
* If matches are found, the recipe is retrieved from the Vector Database.

1. **Generate Recipe Instructions:**

* The retrieved recipe data is sent to the LLMProviderService to generate structured recipe instructions, ensuring clarity and consistency.

1. **Display Recipe Instructions:**

* Recipe instructions are displayed in a card-based format, including:
  + Progress tracker at the top.
  + Step numbers and titles.
  + Detailed instructions for each step.
  + Interactive buttons for clarifications, timers, and navigation.

1. **Use Interactive Features:**

* Users can swipe or click to navigate between steps.
* Timers can be set for specific steps and managed through the TimerManager.
* Notifications are sent when timers complete. Timers can also be paused, adjusted, or reset.

1. **Save Progress:**

* If the user navigates away, the ProgressTracker saves the current step and timer settings.

1. **Resume or Restart:**

* Upon returning, the app allows the user to resume from the last saved step or restart the recipe from the beginning.

1. **Error Handling:**

* If foodo AI, the LLM Provider, or any backend service fails, the RecipeInstructionViewController displays an appropriate error message and allows the user to retry.

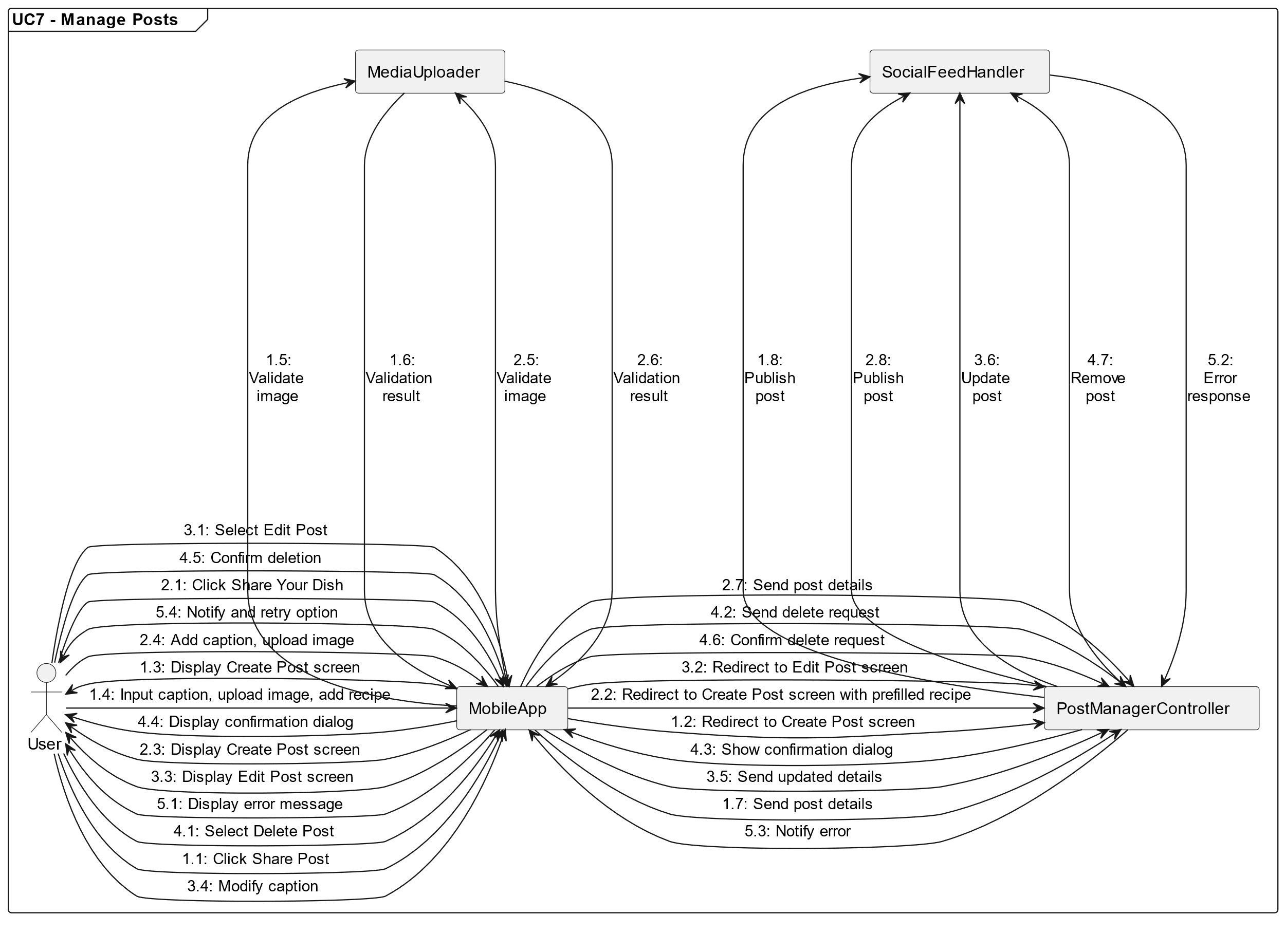


Figure 10. UML Communication Diagram of UC7

**Use Case Name: UC7 – Manage Posts**

**Description:** The mobile app allows users to create, edit, and delete posts on the social feed. Users can manually create posts by providing captions, uploading images, and optionally including recipes. Posts can also be created directly from completed AI-guided recipes with prefilled recipe data, which is immutable. Users can edit captions or delete posts as needed, ensuring the social feed remains dynamic and user-driven.

**Actors:**

* **User:** The individual creating, editing, or deleting posts.

**Participating Classes/Objects:**

* **MobileApp:** Coordinates user interactions, validation, and communication with backend services, including handling immutability of prefilled recipes.
* **PostManagerController**: Manages the creation, editing, and deletion of posts.
* **SocialFeedHandler:** Handles publishing posts to and removing posts from the social feed.
* **MediaUploader:** Manages the upload and validation of image files.

**Flow of Events**:

1. **Create Post - Manual Entry:**

* The user clicks the **Share Post** button on the home page.
* The PostManagerController redirects the user to the **Create Post** screen.
* The user inputs a caption (up to 1907 characters), uploads an image file (JPEG, PNG, etc., max 10MB), and optionally adds a recipe manually.
* The "Post" button is enabled once a caption or recipe is entered and an image is uploaded.
* When the user clicks "Post," the PostManagerController validates the content and sends it to SocialFeedHandler for publishing.

1. **Create Post - Prefilled Recipe:**

* If the user clicks the **Share Your Dish** button after completing a recipe guided by the AI assistant, they are redirected to the **Create Post** screen.
* The recipe field is prefilled with data from the completed recipe and marked immutable by the **Mobile App** UI logic.
* The user adds a caption and uploads an image before clicking "Post."
* The PostManagerController validates the content and sends it to SocialFeedHandler for publishing.

1. **Edit Post:**

* The user selects the "Edit" option from a post's menu.
* The PostManagerController redirects the user to the **Edit Post** screen, prefilled with the current post’s data.
* The recipe field remains immutable, enforced by the **Mobile App** UI, but the user can modify the caption.
* Upon clicking "Save," the PostManagerController validates the updated content and sends it to SocialFeedHandler to update the post in the feed.

1. **Delete Post:**

* The user selects the "Delete Post" option from a post’s menu.
* A confirmation dialog appears.
* If the user confirms, the PostManagerController sends a deletion request to SocialFeedHandler, which removes the post from the feed permanently.

1. **Error Handling:**

* If validation fails during post creation or editing (e.g., missing required fields, invalid image file), the app displays an error message.
* If the post cannot be published, edited, or deleted due to a backend error, the app notifies the user and allows them to retry.

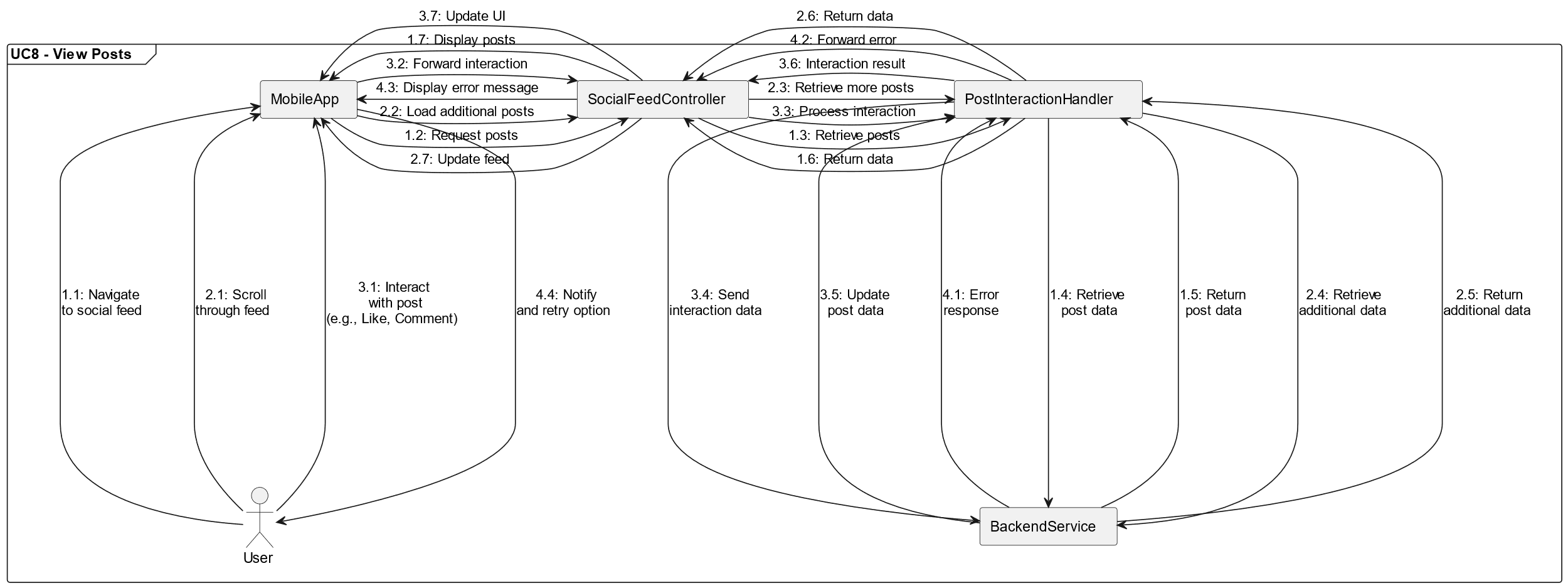


Figure 11. UML Communication Diagram of UC8

**Use Case Name: UC8 – View Posts**

**Description:** The mobile app features a social feed that displays posts from other users in reverse chronological order. Each post shows the user's profile picture, name, content, and the date and time of posting. Users can engage with posts using interaction buttons like "Like" and "Comment," enhancing engagement and interactivity.

**Actors:**

* **User:** The individual viewing and interacting with posts.

**Participating Classes/Objects:**

* **MobileApp:** Coordinates the loading and display of the social feed.
* **SocialFeedController:** Manages the retrieval and presentation of posts in the social feed.
* **PostInteractionHandler:** Handles user interactions with posts, such as liking and commenting.
* **BackendService:** Provides the data for posts, including user information, content, and metadata.

**Flow of Events**:

1. **Load Social Feed:**

* When the user navigates to the social feed, the SocialFeedController sends a request to the BackendService to retrieve posts.
* The BackendService returns a list of posts, sorted in reverse chronological order.
* The SocialFeedController displays the posts in the feed, including:
  + The user's profile picture and name.
  + The post content (e.g., text, images).
  + The date and time of posting.

1. **View Post Details:**

* The user scrolls through the feed to view posts.
* The app ensures smooth scrolling and loads additional posts as the user reaches the bottom of the feed (infinite scrolling).

1. **Engage with Posts:**

* The user interacts with a post by clicking buttons such as "Like" or "Comment."
* The PostInteractionHandler sends the user's action to the BackendService.
* The BackendService processes the interaction and updates the post data (e.g., increments the like count or adds a comment).
* The SocialFeedController updates the UI to reflect the new interaction (e.g., like button changes state or a comment appears).

1. **Error Handling:**

* If posts fail to load due to a network error, the app displays an error message and allows the user to retry.
* If an interaction fails (e.g., liking or commenting), the app displays a message and reverts any visual changes.

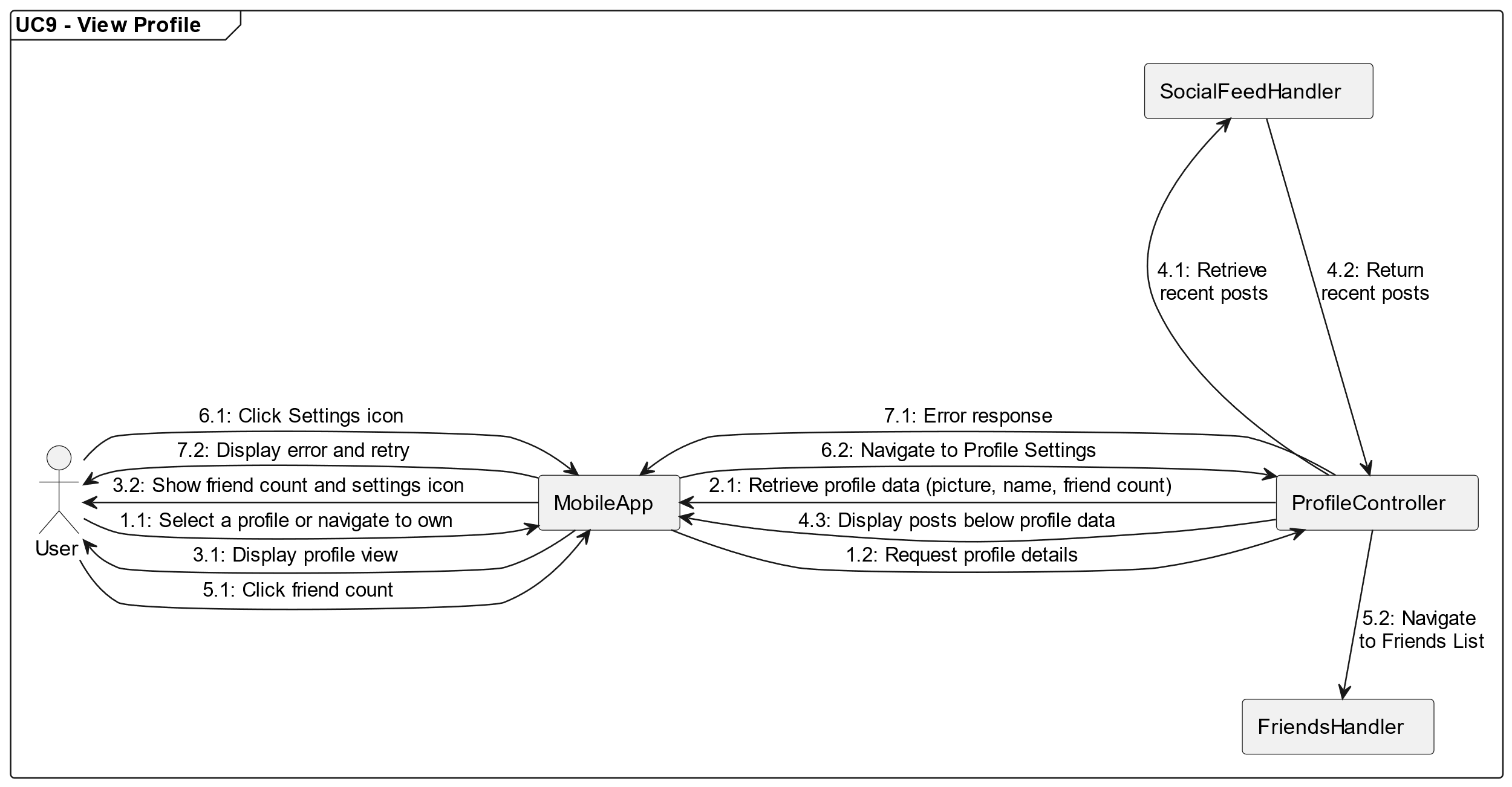


Figure 12. UML Communication Diagram of UC9

**Use Case Name: UC9 – View Profile**

**Description:** The mobile app provides a detailed profile view when a user navigates to their own profile or another user's profile. The profile includes the user's profile picture, name, friend count, and a list of their recent posts. If the user is viewing their own profile, additional options are available, such as navigating to the friends list by clicking the friend count and accessing the profile settings via a "Settings" icon.

**Actors:**

* **User:** The individual viewing their own or another user's profile.

**Participating Classes/Objects:**

* **MobileApp:** Coordinates navigation and displays profile details.
* **ProfileController:** Manages retrieval and display of profile data.
* **SocialFeedHandler:** Retrieves the user's recent posts for display in the profile view.
* **FriendsHandler:** Manages friend count and navigation to the friends list.

**Flow of Events**:

1. **Navigate to Profile:**

* The user selects a profile from the social feed or navigates to their own profile.
* The ProfileController sends a request to retrieve profile details.

1. **Retrieve Profile Data:**

* The ProfileController queries the backend for the user’s profile details, including:
  + Profile picture (if available).
  + Name.
  + Friend count.

1. **Display Profile Data:**

* The app displays the profile view, including the retrieved profile data.
* If the profile belongs to the logged-in user:
  + The friend count is clickable and redirects to the Friends List page.
  + A "Settings" icon is displayed in the top right corner for navigating to the Profile Settings page.

1. **Retrieve Recent Posts:**

* The SocialFeedHandler retrieves the user's recent posts from the social feed.
* The recent posts are displayed below the profile information.

1. **Navigate to Friends List (Own Profile Only):**

* If the user clicks the friend count, the FriendsHandler navigates to the **Friends List** page.

1. **Navigate to Profile Settings (Own Profile Only):**

* If the user clicks the "Settings" icon, the app navigates to the **Profile Settings** page.

1. **Error Handling:**

* If profile data or recent posts fail to load, the app displays an error message and allows the user to retry.

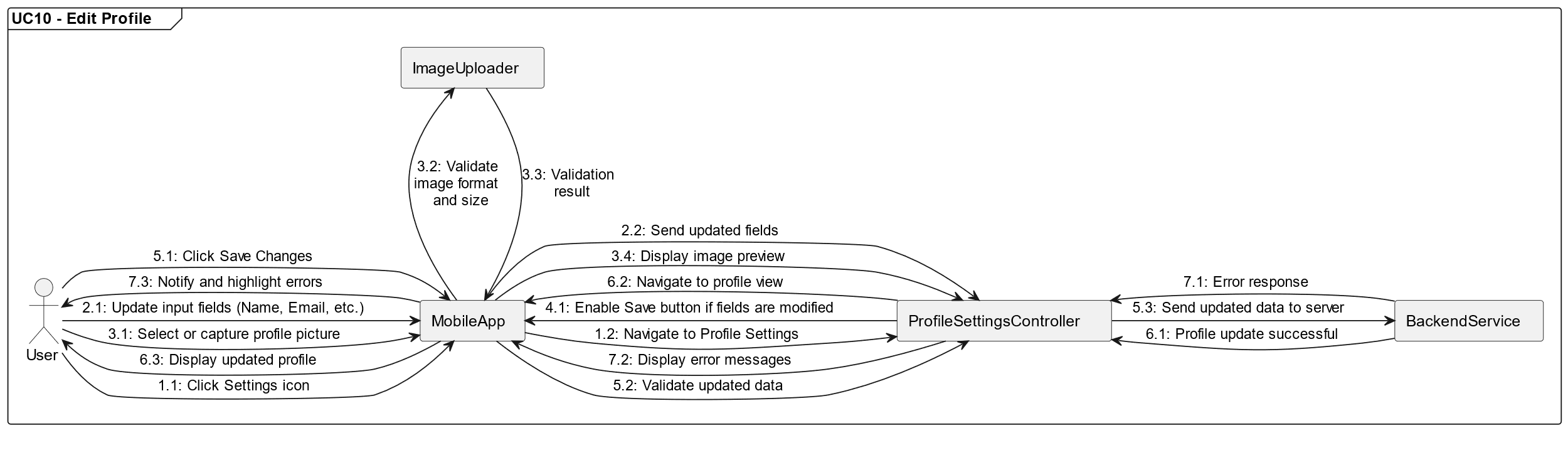


Figure 13. UML Communication Diagram of UC10

**Use Case Name: UC10 – Edit Profile**

**Description:** The mobile app allows users to update all personal information through the **Profile Settings** screen. This includes editing their name, email address, phone number, date of birth, and profile picture. Input fields for each piece of information enforce specific validation rules. Users can only save changes if at least one field is modified. Upon saving, the system validates all inputs and updates the user's profile on the backend. Errors are displayed to guide users in resolving validation issues.

**Actors:**

* **User:** The individual editing their profile.

**Participating Classes/Objects:**

* **MobileApp:** Coordinates navigation and user interactions.
* **ProfileSettingsController:** Manages the input fields, validation, and profile update logic.
* **ImageUploader:** Handles image selection, validation, and preview generation.
* **BackendService:** Updates the profile data on the server.

**Flow of Events**:

1. **Navigate to Profile Settings:**

* The user clicks the "Settings" icon from their profile view.
* The ProfileSettingsController navigates to the **Profile Settings** screen.

1. **Edit Profile Information:**

* The user can update the following fields:
  + First Name: An input field (2–64 characters) validated for length and invalid characters.
  + Last Name: An input field (2–64 characters) validated for length and invalid characters.
  + Favourite Meals: A checkbox list for favourite meals.
  + Dietary Preferences: A checkbox list for dietary preferences.
  + Cooking Level: A dropdown list for cooking level.

1. **Update Profile Picture:**

* The user selects an image from their device gallery or captures a new one using the device camera (if supported).
* The ImageUploader validates the image for format (JPEG or PNG) and size (max 5 MB).
* A preview of the selected image is displayed on the **Profile Settings** screen.

1. **Enable Save Button:**

* The "Save Changes" button remains disabled until at least one field is modified.

1. **Save Changes:**

* When the user clicks the "Save Changes" button:
  + The ProfileSettingsController validates all updated fields.
  + The system ensures no duplicate emails or phone numbers are being used.
  + The updated profile data is sent to the BackendService.

1. **Profile Update Success:**

* If the profile update is successful, the app navigates back to the user’s profile view and displays the updated information.

1. **Error Handling:**

* If any validation errors occur (e.g., invalid email format, unsupported image, or duplicate phone number), the app displays error messages and highlights the relevant fields.
* If the update fails due to a backend error, the app notifies the user and allows them to retry.

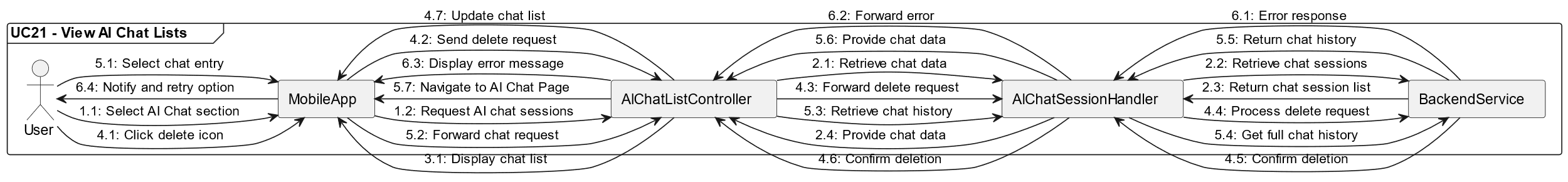


Figure 14. UML Communication Diagram of UC21

**Use Case Name: UC21 – View AI Chat Lists**

**Description:** The mobile app allows users to view a list of previous AI chat sessions in the "AI Chat" section. Each chat entry displays a preview of the last exchange and a timestamp of the most recent interaction. Users can delete chats using an icon button or select a chat entry to continue the conversation seamlessly on the corresponding AI chat page.

**Actors:**

* **User:** The individual viewing and managing AI chat sessions.

**Participating Classes/Objects:**

* **MobileApp:** Coordinates user interactions and navigation within the app.
* **AIChatListController:** Manages the retrieval and display of the AI chat list.
* **AIChatSessionHandler:** Handles navigation to the selected chat session.
* **BackendService:** Provides data for previous chat sessions and processes delete requests.

**Flow of Events**:

1. **Access AI Chat Section:**

* The user selects the **AI Chat** section from the navigation bar or other designated access points.
* The AIChatListController sends a request to the BackendService to retrieve the list of previous chat sessions.

1. **Retrieve AI Chat List:**

* The BackendService returns the list of AI chat sessions, including:
  + A preview of the last exchange within each chat.
  + A timestamp for the most recent message.

1. **Display AI Chat List:**

* The AIChatListController displays the retrieved chat sessions in a scrollable list format.
* Each entry includes:
  + The chat preview.
  + The timestamp of the last message.
  + A delete icon for managing chat sessions.

1. **Delete Chat Session:**

* The user clicks the delete icon for a specific chat entry.
* The AIChatListController sends a delete request to the BackendService.
* Upon confirmation from the BackendService, the entry is removed from the displayed list.

1. **Continue Chat Session:**

* The user selects a chat entry from the list.
* The AIChatSessionHandler retrieves the complete chat history for the selected session from the BackendService.
* The app navigates to the **AI Chat Page**, displaying the full chat history and allowing the user to continue the conversation.

1. **Error Handling:**

* If the chat list fails to load, the app displays an error message and allows the user to retry.
* If deleting a chat fails, the app notifies the user and retains the entry in the list.

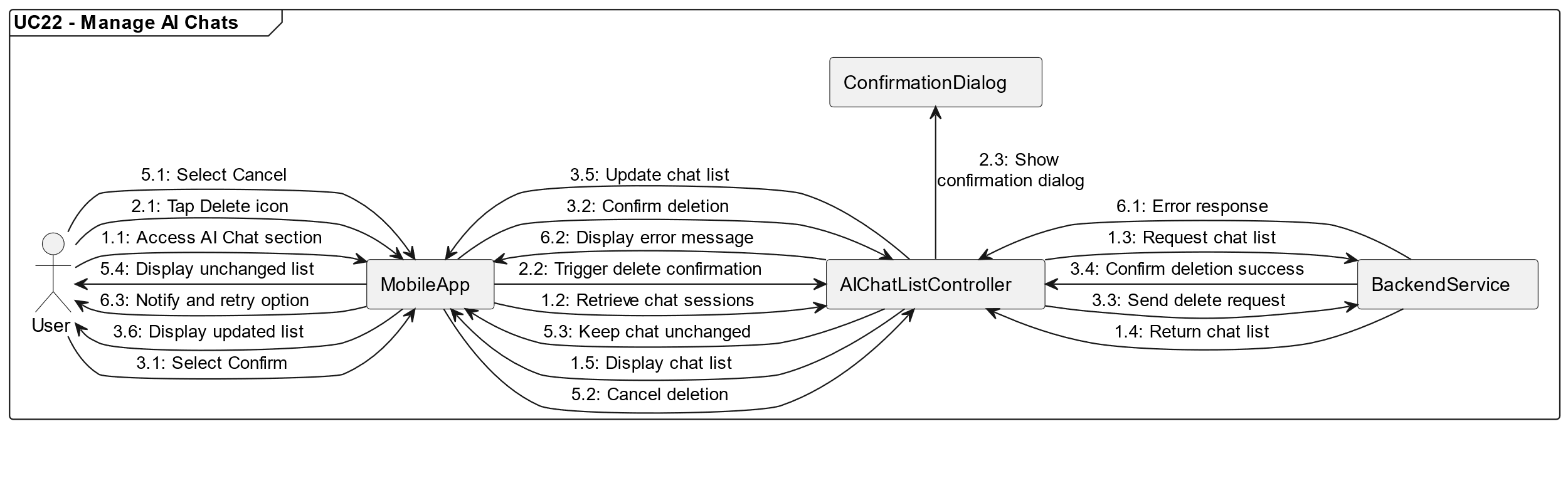


Figure 15. UML Communication Diagram of UC22

**Use Case Name: UC22 – Manage AI Chats**

**Description:** The mobile app allows users to manage their AI chat sessions by deleting individual chat sessions from the AI Chat list. When a user taps the "Delete" icon next to a chat preview, the app displays a confirmation dialog. If the user confirms, the chat is permanently deleted from the system, and the chat list is updated accordingly. If the user cancels the action, the chat remains in the list unchanged.

**Actors:**

* **User:** The individual managing the AI chat sessions.

**Participating Classes/Objects:**

* **MobileApp:** Coordinates user interactions and manages navigation.
* **AIChatListController:** Handles the retrieval and display of AI chat sessions and manages chat deletion requests.
* **ConfirmationDialog:** Displays the confirmation dialog for chat deletion.
* **BackendService:** Processes the deletion request and updates the chat list in the database.

**Flow of Events**:

1. **Navigate to AI Chat List:**

* The user accesses the AI Chat list by navigating to the "AI Chat" section.
* The AIChatListController retrieves the list of previous AI chat sessions from the BackendService and displays them.

1. **Tap "Delete" Icon:**

* The user taps the "Delete" icon next to a specific chat entry.
* The AIChatListController triggers the display of a **ConfirmationDialog** asking, "Are you sure you want to delete this chat session?"

1. **User Confirmation:**

* If the user selects **Confirm**, the AIChatListController sends a delete request to the BackendService.
* The BackendService permanently deletes the chat session and updates the chat list.

1. **Delete Chat Session:**

* The AIChatListController updates the displayed chat list to reflect the removal of the deleted chat.
* The user is shown the updated AI chat list with the chat removed.

1. **Cancel Deletion:**

* If the user selects **Cancel**, the deletion action is aborted, and the chat session remains in the list unchanged.
* The **ConfirmationDialog** is closed without modifying the chat list.

1. **Error Handling:**

* If the deletion fails due to a backend error, the app notifies the user of the issue and allows them to retry or cancel the action.

1. **Physical View**

****

Figure 16. UML Deployment Diagram

The physical view of the foodo system is represented by the UML Deployment Diagram (Figure 16). It illustrates deploying various software components across the physical infrastructure, ensuring scalability, reliability, and maintainability.

The architecture is based on a microservices approach, with each service hosted on AWS EC2. The deployment leverages Nginx to ensure secure communication through HTTPS between the Client and the Main Gateway. Kafka is used for asynchronous messaging, enabling communication between microservices, including User Service, Notification Service, Achievement Service, and Social Media Service.

External integrations, including AWS Lambda and QDrant, provide serverless compute and vectorized database capabilities, supporting AI-driven features. Clerk Authentication Service handles secure user authentication, while Firebase Cloud Messaging enables real-time push notifications to users. User-generated content, such as images, is stored in AWS S3 for efficient access and scalability.

This deployment architecture ensures that foodo's critical components operate in a highly available and distributed environment capable of handling dynamic workloads and delivering a responsive user experience. Each microservice is independently deployable and scalable, allowing for robust fault tolerance and maintainability.

1. **Design Quality**

This section explains how the design addresses each Non-Functional Requirement specified in Section 2.2 and how these requirements will be verified and validated.

**Usability Requirements**

* **UR1:** Notifications, including those for timed cooking steps, alerts, and achievements, must be displayed within **3 seconds** of the **%80** of the events to ensure clarity for the user.

**Design Approach:** Use Firebase Cloud Messaging for push notifications and optimize backend response times. Implement caching for time-critical events.

**Verification:** Conduct manual response time tests simulating notification events, timed cooking steps, alerts, and achievements, ensuring delivery times meet the requirement. Validate backend performance by logging and analyzing event processing times during high-load scenarios.

**Validation:** Perform real-world tests on devices (iOS and Android) with different network conditions to confirm responsiveness and compliance with the 3-second requirement.

* **UR2:** Essential tasks, such as interacting with the AI assistant and accessing social features, should be accessible within **five taps** from the home screen.

**Design Approach:** Utilize a bottom navigation bar for core features like the AI assistant, social feed, and profile, ensuring key tasks are no more than one or two taps away. Optimize screen transitions and navigation workflows to minimize redundant steps.

**Verification:** Conduct manual usability testing with mock scenarios to verify that essential tasks can be completed within five taps from the home screen.

**Validation:** Perform user acceptance testing on iOS and Android devices with representative users to confirm instinctive navigation and task accessibility. Collect feedback to refine workflows and ensure tasks consistently meet the five-tap threshold.

**Performance Requirements**

* **PR1:**Social actions (e.g., likes, comments), shall be processed in under **5 seconds** for **90%** of **25** requests, maintaining a high level of responsiveness.

**Design Approach:** Implement a scalable microservice architecture to distribute workload efficiently across services. Use optimized database queries and caching mechanisms to reduce latency for high-traffic interactions.

**Verification:** Conduct lightweight performance testing by manually initiating concurrent requests for social actions through tools like Postman or small-scale custom scripts. Record response times and verify that 90% of requests meet the 5-second threshold. Review simple server-side logs to detect potential delays.

**Validation:** Perform manual testing under common usage scenarios on iOS and Android devices with a few test users. Note response times during peak and non-peak periods to ensure performance consistency. Collect feedback to confirm perceived responsiveness meets expectations.

* **PR2:** The AI cooking assistant shall answer the prompts in **under 15 seconds** after receiving input supporting smooth and prompt user engagement.

**Design Approach:** Leverage a serverless architecture with AWS Lambda for processing AI prompts, ensuring scalability and optimized response times. Use Qdrant for vectorized AI data retrieval and prioritize caching frequently requested prompts to reduce processing overhead during peak times.

**Verification:** Conduct manual tests simulating scenarios. Measure AI response times using tools like Postman for a range of inputs and verify that responses are under 15 seconds. Analyze AWS Lambda logs to identify potential delays or bottlenecks.

**Validation:** Perform manual testing under common usage scenarios on iOS and Android devices with a few test users. Note response times during peak and non-peak periods to ensure performance consistency. Collect feedback to confirm perceived responsiveness meets expectations.

* **PR3:**User-generated content, such as images in the social feed, shall load within **5 seconds** for **90%** of **20 requests**, ensuring smooth content interaction.

**Design Approach:** Utilize AWS S3 for image storage. Implement lazy loading to prioritize visible content and reduce initial load times.

**Verification:** Perform manual upload and retrieval tests for user-generated content using various image sizes. Time the loading of images in the social feed and verify that 90% of 20 requests are loaded within 5 seconds.

**Validation:** Conduct tests on both iOS and Android devices under different network conditions (e.g., 4G, Wi-Fi). Gather user feedback on image loading speeds and verify the smoothness of interactions with the social feed.

* **PR4:** Push notifications (cooking reminders, social alerts) shall be delivered within **10 seconds** of the triggered event for **95%** of cases, verified through real-world testing with **2 devices**.

**Design Approach:** Implement Firebase Cloud Messaging (FCM) for delivering push notifications with low latency. Optimize backend services to trigger notifications promptly using event-driven mechanisms like Kafka.

**Verification:** Conduct manual tests using controlled scenarios to trigger notifications, such as cooking reminders and social alerts. Measure delivery times and ensure 95% of notifications are delivered within 10 seconds.

**Validation:**  Perform real-world testing on two devices, one iOS and one Android, under various network conditions (e.g., 4G, Wi-Fi). Validate notification delivery times by collecting user feedback and comparing results against the 10-second delivery requirement. Adjust backend configurations if necessary to maintain reliability.

* **PR5:** Image uploads up to **5 MB** shall complete within **20 seconds**, and images shall display in the feed within **20 seconds**, tested over **30 uploads** under average network conditions.

**Design Approach:** Use AWS S3 for efficient and scalable storage of uploaded images. Optimize upload processes by implementing direct-to-S3 uploads and leveraging multipart uploads for larger files.

**Verification:** Perform manual testing by uploading 5 MB images through the app and measuring upload completion times under average network conditions. Verify that the images are displayed in the feed within 20 seconds by simulating 30 consecutive uploads. Use server logs to ensure consistent performance and identify bottlenecks.

**Validation:** Conduct on representative iOS and Android devices, simulating real-world conditions with average and high-latency networks. Validate compliance with the 20-second threshold by observing upload and display times. Collect user feedback to confirm a seamless experience and adjust configurations if delays are observed.

**Software System Attributes**

* **Reliability / SA-R1:** The system shall maintain **95%** uptime for essential functions (user authentication, AI assistance) over an **8-hours** period, monitored using uptime tracking tools.

**Design Approach:** Deploy essential services (user authentication, AI assistance) on highly available infrastructure, such as AWS Lambda & AWS EC2 instances. Use health to ensure service continuity during disruptions.

**Verification:** Use monitoring tools like AWS CloudWatch to simulate 8-hour test periods and track uptime metrics for critical services. Analyze logs to identify any downtime and verify that uptime meets or exceeds 95%.

**Validation:** Conduct real-world testing by running the system continuously for multiple 8-hour intervals under typical and peak loads. Validate uptime compliance by reviewing logs.

* **Availability / SA-A1:** The system shall maintain an availability level of **95%** in **8 hours**, minimizing the risk of downtime.

**Design Approach:** Utilize a load-balanced architecture with redundant instances of critical services to ensure availability during failures. Implement auto-scaling for services hosted on AWS EC2 to handle varying loads efficiently.

**Verification:** Monitor system availability using tools like AWS CloudWatch over an 8-hour window. Verify that availability remains above 95% during these tests.

**Validation:** Perform real-world testing by subjecting the system to typical and peak load conditions across multiple 8-hour intervals. Review user feedback to ensure minimal downtime and compliance with the 95% availability threshold.

* **Security / SA-S1:** The application shall implement a valid SSL/TLS certificate for HTTPS connection for **%100** of the endpoints that transmit data between the client and server.

**Design Approach:** Configure HTTPS for all client-server communication by implementing valid SSL/TLS certificates through a trusted Certificate Authority. Enforce HTTPS redirection at the server level using Nginx as a reverse proxy.

**Verification:** Manually verify HTTPS connections for all endpoints using a browser or Postman.

**Validation:** Conduct testing to confirm the absence of vulnerabilities in HTTPS implementation. Validate SSL/TLS encryption for endpoints during real-world testing, ensuring secure data transmission across different devices and network conditions.

* **Maintainability / SA-M1:** The codebase shall be organized into modules (AI processing, user management, etc.), with each module documented. Code reviews shall confirm adherence to modular design.

**Design Approach:** Structure the codebase into well-defined modules, such as AI processing, user management, and social features, ensuring clear separation of concerns. Provide documentation for each module, including functionality descriptions, API specifications, and usage examples. Implement consistent coding standards and version control practices across the team.

**Verification:** Conduct regular peer code reviews to ensure adherence to modular design principles and coding standards. Use static analysis tools like SonarLint to detect violations of modularity or maintainability metrics. Review documentation completeness and accuracy during team meetings.

**Validation:** Validate maintainability by updating or fixing bugs in isolated modules to ensure changes can be made efficiently. Confirm the clarity of documentation through team reviews and refine it based on suggestions.

* **Portability / SA-M1:** The codebase shall be organized into modules (AI processing, user management, etc.), with each module documented. Code reviews shall confirm adherence to modular design.

**Design Approach:** Utilize React Native for cross-platform development, ensuring compatibility with iOS 13+ and Android 12+. Leverage Expo for live testing on supported devices.

**Verification:** Perform compatibility tests on two devices for each platform using React Native testing tools and Expo. Ensure the app installs, runs, and performs essential tasks without errors.

**Validation:** Conduct manual testing on iOS and Android devices to verify smooth functionality and consistent user experience. Address any platform-specific issues identified during testing.

**Error Handling Requirements**

* **EH-U1: Input Validation Errors**

**Design Approach:** Implement real-time input validation on both the client and server sides for fields such as email, password, and usernames. Use regular expressions and predefined validation rules to ensure compliance with expected formats.

**Verification:** Perform manual and automated tests for input validation by providing invalid data (e.g., special characters, invalid email formats) and verifying that appropriate error messages are displayed within 2 seconds

**Validation:** Conduct user testing to confirm that error messages are clear and help users correct their input. Gather feedback on usability and iterate as needed to improve clarity and guidance.

* **EH-U2: Authentication Errors**

**Design Approach:** Integrate Clerk Authentication to handle user login securely. Display clear, contextual error messages for incorrect credentials or non-existent accounts.

**Verification:** Test authentication flows with incorrect credentials to verify error message display.

**Validation:** Conduct testing with a focus on user feedback for error message clarity and guidance during authentication failures. Confirm that users can recover access smoothly using features like password recovery.

* **EH-S1: System Errors**

**Design Approach:** Implement error handling mechanisms for server-side failures. Provide users with non-technical error messages to reduce frustration.

**Verification:** Simulate server errors (e.g., database unavailability, API timeout) and verify that the system displays user-friendly error messages within 2 seconds.

**Validation:** Perform real-world testing under simulated failure scenarios to validate user experience and recovery strategies. Ensure that error handling does not disrupt unrelated functionality.

* **EH-AI1: AI Interaction Errors**

**Design Approach:** Make sure user inputs are sent to the AI system to ensure completeness and clarity before processing. Implement fallback mechanisms for unsupported or unclear requests. Suggest corrections or additional details to help users refine their queries.

**Verification:** Test AI interactions with valid and invalid inputs to verify that appropriate error messages are displayed promptly. Log errors for analysis to improve input validation and interaction quality.

**Validation:** Conduct usability testing with users submitting both typical and edge-case requests. Collect feedback on the clarity and helpfulness of error messages and suggestions, refining them as needed to ensure smooth user interactions.

# Low Level Design

The pseudocode and diagrams for all use cases referenced in this document are available in the GitHub repository at the following location:  
**GitHub Repository:** [Deliverable 4 - SDD](https://github.com/foodoHub/project-docs/tree/main/deliverable-4-sdd)

<https://github.com/foodoHub/project-docs/tree/main/deliverable-4-sdd>

This repository contains well-structured and detailed pseudocode for each use case and comprehensive UML diagrams, including sequence and class diagrams.

**First Increment Use-Cases**

1. **UC1: Login To System**

**Summary:**

The pseudo-code represents a simple login flow:

1. Validate input.
2. Authenticate user.
3. On success: Store token and redirect.
4. On failure: Show error message.

A screen shot of a computer program

Description automatically generated

Figure 17. Pseudocode of UC1

**Explanation:**

**handleLoginFormSubmit(email, password)**

* **Purpose**: Coordinates the entire login process.
* **Steps**:
  + Validates the input via validateInput.
  + If valid:
    - Authenticates the user using authenticateUser.
    - If authentication succeeds:
      * Stores the token via storeToken.
      * Redirects to the feed screen with redirectToFeed.
    - If authentication fails, displays the error using displayErrorMessage.
  + If invalid input, directly displays an error.

**validateInput(email, password)**

* **Purpose**: Validates the provided email and password.
* Uses Input.validate(email, password) for the actual validation logic.

**authenticateUser(email, password)**

* **Purpose**: Authenticates the user credentials.
* Calls ClerkSDK.authenticate(email, password) to handle authentication

**storeToken(token)**

* **Purpose**: Saves the authentication token.
* Uses TokenStorage.add(token) to store the token securely to local storage.

**redirectToFeed()**

* **Purpose**: Redirects the user to the feed page.
* Implementation logic for navigation is yet to be defined.

**displayErrorMessage(message)**

* **Purpose**: Displays error messages to the user.
* Calls Error.display(message) to show the error.

A diagram of a computer

Description automatically generated with medium confidence

Figure 18. UML Class Diagram of UC1

A screenshot of a computer screen

Description automatically generated

Figure 19. UML Sequence Diagram of UC1

1. **UC2: Recover Password**

**Summary:**

The pseudo-code defines the password recovery flow:

1. Validate the email format.
2. Send a reset link and prompt for a verification code.
3. Verify the code:
   * On success: Prompt for a new password.
   * On failure: Display an error.
4. Update the password:
   * On success: Redirect to the login page.
   * On failure: Display an error.

A screen shot of a computer code

Description automatically generated

Figure 20. Pseudocode of UC2

**Explanation:**

**handleRecoverPasswordFormSubmit(email)**

**Purpose**: Coordinates the password recovery process.

* Validates the email using validateEmail.
* If valid:
  + Sends a reset link via sendResetLink.
  + Displays the verification code field using displayEnterCodeField.
* If invalid:
  + Displays an error message using displayErrorMessage.

**validateEmail(email)**

**Purpose**: Validates the provided email format.

* Uses Input.validateEmail(email) to confirm a proper format.

**sendResetLink(email)**

**Purpose**: Sends a reset link to the given email.

* Uses ClerkSDK.sendResetLink(email) to handle the actual process.

**submitVerificationCode(code)**

**Purpose**: Verifies the provided code.

* Sends the code to ClerkSDK.verifyCode(code).
* On success:
  + Displays the new password field via displayNewPasswordField.
* On failure:
  + Displays an error message using displayErrorMessage.

**submitNewPassword(newPass)**

**Purpose**: Updates the user's password.

* Calls ClerkSDK.updatePassword(newPass) to finalize the reset process.
* On success:
  + Redirects the user to the login page via redirectToLoginPage.
* On failure:
  + Displays an error message using displayErrorMessage.

**displayEnterCodeField()**

**Purpose**: Prompts the user to enter the verification code.

**displayNewPasswordField()**

**Purpose**: Prompts the user to enter a new password.

**redirectToLoginPage()**

**Purpose**: Redirects the user to the login page.

**displayErrorMessage(message)**

**Purpose**: Displays error messages to the user.

* Uses Error.display(message) to show the error.

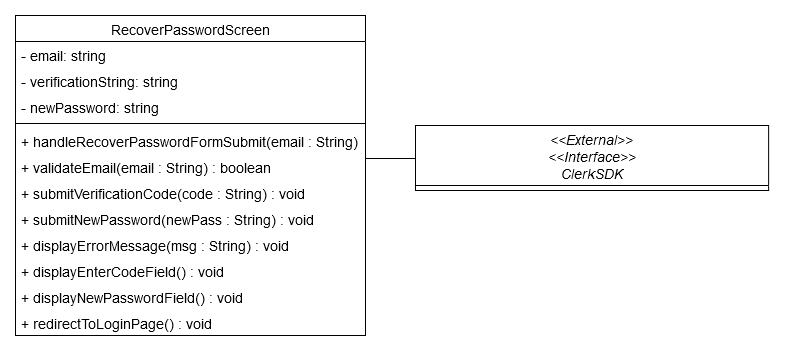


Figure 21. UML Class Diagram of UC2

A diagram of a project

Description automatically generated with medium confidence

Figure 22. UML Sequence Diagram of UC2

1. **UC3: Register New User**

**Summary:**

Validate the user's input.

1. Register the user through an external SDK.
2. On success:
   * Redirect the user to the onboarding screen.
3. On failure:
   * Display an appropriate error message.

A computer screen shot of a program

Description automatically generated

Figure 23. Pseudocode of UC3

**Explanation:**

**handleRegisterFormSubmit(email, firstName, lastName, password)**

**Purpose**: Coordinates the entire user registration process.

**Steps**:

1. Validates the input using validateInputs.
2. If the input is valid:
   * Registers the user via registerUser.
   * On successful registration:
     + Redirects the user to the onboarding screen using redirectToUserOnboardingScreen.
   * On failure:
     + Displays an error message using displayErrorMessage.
3. If the input is invalid:
   * Displays an error message directly.

**validateInputs(email, firstName, lastName, password)**

**Purpose**: Validates the user’s input fields (email, first name, last name, password).

* Utilizes Input.validate methods to ensure correctness.

**registerUser(email, firstName, lastName, password)**

**Purpose**: Handles the interaction with the external backend (e.g., ClerkSDK) for user registration.

* Uses ClerkSDK.register(email, firstName, lastName, password) to create the user in the system.

**redirectToUserOnboardingScreen()**

**Purpose**: Redirects the user to the onboarding screen after successful registration.

* Implements the navigation logic to transition to the next step of the user journey.

**displayErrorMessage(message)**

**Purpose**: Displays an error message to the user.

* Calls Error.display(message) to show user-friendly feedback.

A close-up of a document

Description automatically generated

Figure 24. UML Class Diagram of UC3

A diagram of a project

Description automatically generated

Figure 25. UML Sequence Diagram of UC3

1. **UC4: Navigate Pages**

**Summary:**

The pseudo-code defines a navigation system that:

1. Handles bottom navigation options.
2. Navigates to specific screens based on the selected option:
   * Social Feed Screen
   * AI Chat Screen
   * Profile Screen
3. Each screen implements a load method for initialization.

A screen shot of a computer program

Description automatically generated

Figure 26. Pseudocode of UC4

**Explanation:**

**handleBottomNavigation(option)**

**Purpose**: Manages the user's navigation choice.

**Steps**:

1. Accepts a parameter option, which specifies the selected navigation target.
2. Executes corresponding actions based on the option:
   * Calls showSocialFeed() if option is "SocialFeed".
   * Calls showAIChat() if option is "AIChat".
   * Calls showProfile() if option is "Profile".
3. Assumes only valid options are passed; invalid options handling is not included in this implementation.

**showSocialFeed()**

**Purpose**: Redirects to and initializes the Social Feed Screen.

* Calls the load() method of the SocialFeedScreen class to render and initialize its content.

**showAIChat()**

**Purpose**: Redirects to and initializes the AI Chat Screen.

* Calls the load() method of the AIChatScreen class to set up the chat functionality.

**showProfile()**

**Purpose**: Redirects to and initializes the Profile Screen.

* Calls the load() method of the ProfileScreen class to display the user's profile.

A diagram of a computer program

Description automatically generated with medium confidence

Figure 27. UML Class Diagram of UC4

**A diagram of a project

Description automatically generated**

Figure 28. UML Sequence Diagram of UC4

The class diagrams of UC5: Get Meal Suggestions and UC6: Get Recipe Instructions will be shown in a single diagram. The diagram is included at the end of the UC6. Each will have it’s pseudo code and sequence diagrams.

1. **UC5: Get Meal Suggestions**

**Summary:**

The pseudo-code outlines the process for handling user messages in the chat application:

1. User Interaction: The user sends a message through the chat interface.
2. Backend Coordination: The system validates the message and coordinates message creation across services.
3. History and Preferences Retrieval: The application fetches user preferences and chat history to enhance functionality.
4. AI Integration: A meal suggestion is generated using AI based on user preferences and chat history.
5. Data Filtering: The generated suggestion is filtered for relevance.
6. UI Update: The filtered suggestion is displayed on the chat interface.

Error Handling: Errors are captured, and appropriate messages are displayed to the user.

A screen shot of a computer program

Description automatically generated

Figure 29. Pseudocode of UC5

**Explanation:**

**sendMessage(message, chatID)**

**Purpose**: Orchestrates the process of sending a message and updating the chat list.

**Steps**:

1. Calls ChatController.createMessage to send the message.
2. Calls ChatScreen.updateChatList to refresh the UI.
3. Displays an error message using ChatScreen.displayErrorMessage in case of any exceptions.

**ChatController.createMessage(message, chatID)**

**Purpose**: Delegates the message creation request to the service layer. **Implementation**:

* Invokes ChatService.createMessage.

**ChatController.getChatHistory(chatID)**

**Purpose**: Retrieves chat history for the given chat ID.

**Implementation**:

* Calls ChatService.getHistory.

**ChatService.createMessage(message, chatID)**

**Purpose**: Handles the business logic for creating a message.

**Steps**:

1. Saves the message via ChatRepository.saveMessage.
2. Fetches user preferences using UserService.getUserPreferences.
3. Retrieves chat history from ChatRepository.getHistory.
4. Obtains a meal suggestion using LLMProvider.getMealSuggestion.
5. Filters the suggestion using VectorDBHandler.filterData.
6. Displays the suggestion through ChatScreen.displayAIResponse.

**ChatService.getHistory(chatID)**

**Purpose**: Fetches the chat history for the given chat ID. **Implementation**:

* Delegates to ChatRepository.getHistory.

**ChatRepository.saveMessage(chatID, message)**

**Purpose**: Stores the message in the database.

**Implementation**:

* Uses a database operation to save the message.

**ChatRepository.getHistory(chatID)**

**Purpose**: Retrieves chat history from the database.

**Implementation**:

* Fetches and returns the history for the given chat ID.

**UserService.getUserPreferences(chatID)**

**Purpose**: Retrieves user-specific preferences.

**Implementation**:

* Fetches preferences from the backend.

**LLMProvider.getMealSuggestion(history, preferences)**

**Purpose**: Generates meal suggestions using AI.

**Implementation**:

* Uses chat history and user preferences as inputs for the AI.

**VectorDBHandler.filterData(mealSuggestion)**

**Purpose**: Filters meal suggestions to ensure relevance. **Implementation**:

* Processes the AI-generated suggestion for refinement.

**ChatScreen.displayAIResponse(response)**

**Purpose**: Updates the UI with the AI-generated response. **Implementation**:

* Displays the response in the chat interface.

**ChatScreen.displayErrorMessage(message)**

**Purpose**: Displays an error message in the UI. **Implementation**:

* Uses a UI component to notify the user of the error.

A diagram of a diagram

Description automatically generated with medium confidence

Figure 30. UML Sequence Diagram of UC5

1. **UC6: Get Recipe Instructions**

**Summary:**

This flow includes user input validation, intent determination, backend service interaction, and AI-based processing. The results are refined, displayed, or flagged as necessary.

1. **User Interaction**: The user sends a message or input via the chat interface.
2. **Intent Validation**: The system validates the user’s intent based on input.
3. **Intent Matching**: Specific intents like "Recipe Instruction", "Meal Suggestion", or "Harmful Content" are matched.
4. **Backend Processing**: The system retrieves history, preferences, or relevant data based on the matched intent.
5. **AI Integration**: LLM-based processing retrieves data for the matched intent, such as recipe instructions or content moderation.
6. **Data Filtering**: Results are processed and refined for relevance and safety.
7. **UI Update**: The results are displayed on the chat interface.

**Error Handling**: Errors are captured and presented to the user via the interface.

A screen shot of a computer program

Description automatically generated

Figure 31. Pseudocode of UC6

**sendMessage(message, chatID)**

**Purpose**: Orchestrates the process of handling user messages and updating the chat UI.

**Steps**:

1. Delegates the message processing to ChatController.createMessage.
2. Updates the chat list UI through ChatScreen.updateChatList.
3. Handles errors and displays error messages using ChatScreen.displayErrorMessage.

**ChatController.createMessage(message, chatID)**

**Purpose**: Delegates message processing to the service layer.

**Implementation**:

* Invokes ChatService.handleMessage to process the message.

**ChatController.getChatHistory(chatID)**

**Purpose**: Retrieves chat history for a given chat ID.

**Implementation**:

* Delegates to ChatService.getHistory.

**ChatService.handleMessage(message, chatID)**

**Purpose**: Processes the user message and executes intent-specific logic.

**Steps**:

1. Saves the user message to the database through ChatRepository.saveMessage.
2. Retrieves the chat history using ChatRepository.getHistory.
3. Fetches user preferences via UserService.getUserPreferences.
4. Determines the user intent using LLMProvider.determineIntent, which classifies the message based on history.
5. Executes intent-specific workflows:
   * **Recipe Instruction**: Calls LLMProvider.getRecipeInstruction.
   * **Meal Suggestion**: Calls LLMProvider.getMealSuggestion.
   * **Harmful Content**: Calls ContentModerator.checkContent.
6. Filters the processed data using VectorDBHandler.filterData.
7. Updates the UI with the filtered results via ChatScreen.displayAIResponse.

**ChatService.getHistory(chatID)**

**Purpose**: Retrieves chat history for a given chat ID.

**Implementation**:

* Delegates to ChatRepository.getHistory.

**ChatRepository.saveMessage(chatID, message)**

**Purpose**: Stores the message in the database.

**Implementation**: Calls a database operation to save the message.

**ChatRepository.getHistory(chatID)**

**Purpose**: Fetches chat history from the database.

**Implementation**: Retrieves the history for the provided chatID.

**UserService.getUserPreferences(chatID)**

**Purpose**: Retrieves user-specific preferences.

**Implementation**: Fetches preferences for the user associated with the chatID.

**LLMProvider.determineIntent(message, history)**

**Purpose**: Determines the intent of the user's message.

**Implementation**: Uses AI-based logic to analyze the message and history, returning the identified intent.

**LLMProvider.getRecipeInstruction(history)**

**Purpose**: Retrieves recipe instructions based on chat history.

**Implementation**: Generates recipe instructions using the chat history as input.

**LLMProvider.getMealSuggestion(history, preferences)**

**Purpose**: Generates meal suggestions using chat history and user preferences.

**Implementation**:

* Utilizes AI models to create personalized meal suggestions.

**ContentModerator.checkContent(message)**

**Purpose**: Detects harmful content in user messages.

**Implementation**:

* Analyzes the message using rules-based and AI-powered moderation techniques.

**VectorDBHandler.filterData(data)**

**Purpose**: Filters and refines the data for relevance and safety.

**Implementation**:

* Processes the provided data to ensure it is appropriate for display.

**ChatScreen.displayAIResponse(response)**

**Purpose**: Updates the UI with the processed AI response.

**Implementation**:

* Displays the filtered data on the chat interface.

**ChatScreen.displayErrorMessage(message)**

**Purpose**: Displays error messages in the UI.

**Implementation**:

* Uses a UI component to notify the user of errors during processing.

A white sheet of paper with black lines

Description automatically generated

Figure 32. UML Sequence Diagram of UC6

A screenshot of a computer screen

Description automatically generated

Figure 33. UML Class Diagram of UC 5&6

UC7 “Manage Posts” is divided into 3 parts as UC7.1, UC7.2 and UC7.3. Managing posts covers creating posts, updating posts and deleting posts. In the first increment only create and delete post functions are introduced. Since they share common grounds, we decided to include UC7.2’s low level design. Class diagrams of UC7.1, UC7.2, UC7.3 and UC8 are combined into a single diagram. Both use cases are related to post functionality, and they share components. It is included at the end of the UC7.1 and UC8. The same diagram is included twice in different parts of the document to make the document more readable and understandable.

1. **UC7.1: Manage Posts (Create Post)**

**Summary:**

The pseudo-code outlines the post creation process:

1. Validate input data (caption, recipe, and image file).
2. Handle image upload if provided.
3. Create a post with or without an image.
4. Update the social feed upon successful post creation.
5. Display an error message in case of failures.

A screen shot of a computer program

Description automatically generated

Figure 34. Pseudocode of UC7.1

**Explanation:**

**handleCreatePostFormSubmit(caption, recipe, imageFile)**

* **Purpose**: Manages the entire post creation process.
* **Steps**:
  1. Validates input data using validateInput.
  2. If input is valid:
     + If an image is provided, upload the image via uploadImage.
     + Receives a signed URL and uploads the image file.
  3. Send post data to the PostService.createPost.
  4. Updates the social feed upon success using refreshFeed.
  5. Displays error messages for any failures.

**validateInput(caption, recipe, imageFile)**

* **Purpose**: Checks the validity of the caption, recipe, and optional image file.
* **Implementation**: Calls Input.validate methods to ensure correctness.

**uploadImage(imageFile)**

* **Purpose**: Handles image upload.
* **Steps**:
  1. Fetches a signed URL from PostService.getSignedURL.
  2. Uploads the image file to the signed URL.
  3. Returns the image URL

**createPost(postData)**

* **Purpose**: Sends the post data to the backend to create a new post.
* **Implementation**: Calls PostService.createPost with the post data.

**refreshFeed()**

* **Purpose**: Refreshes the social feed to display the new post.

**displayErrorMessage(message)**

* **Purpose**: Shows an error message to the user.

**A diagram of a project

Description automatically generated**

Figure 35. UML Sequence Diagram of UC7.1

**A screenshot of a computer screen

Description automatically generated**

Figure 36. UML Class Diagram of UC 7&8

1. **UC7.2: Manage Posts(Update Post)**

**Summary:**

The pseudo-code outlines the process for editing a post:

1. Validate input data (new caption, new recipe, and image file if provided).
2. Handle image upload if a new image file is provided.
3. Update the post with the new data (caption, recipe, and image URL).
4. Refresh the social feed upon successful post update.
5. Display an error message in case of failures.

A screen shot of a computer program

Description automatically generated

Figure 37. Pseudocode of UC7.2

**Explanation**:

**handleEditPostFormSubmit(newCaption, newRecipe, newImageFile, postID)**

**Purpose**: Manages the entire post editing process.

**Steps**:

1. Validates input data using validateInput.
2. If input is valid:
   * Uploads a new image (if provided) using uploadImage.
   * Sends updated post data to PostService.updatePost.
   * Refreshes the social feed upon success using refreshFeed.
3. Displays error messages for any failures.

**validateInput(newCaption, newRecipe, newImageFile)**

**Purpose**: Checks the validity of the new caption, recipe, and optional image file.

**Implementation**: Calls Input.validate methods to ensure correctness.

**uploadImage(newImageFile)**

**Purpose**: Handles the upload of a new image file.

**Steps**:

1. Fetches a signed URL from PostService.getSignedURL.
2. Uploads the image file to the signed URL using ImageUploader.upload.
3. Returns the image URL upon success.
4. Displays an error message if the upload fails.

**updatePost(updatedData, postID)**

**Purpose**: Sends updated post data (new caption, recipe, and image URL) to the backend for updating the post.

**Implementation**: Calls PostService.updatePost with the updated data and post ID.

**refreshFeed()**

**Purpose**: Refreshes the social feed to reflect the edited post.

**displayErrorMessage(message)**

**Purpose**: Displays an error message to the user in case of invalid input or backend failures.

A diagram of a project

Description automatically generated with medium confidence

Figure 38. UML Sequence Diagram of 7.2

1. **UC7.3: Manage Posts (Delete Post)**

**Summary:**

The pseudo-code outlines the process for deleting a post:

1. Handle the delete button click event.
2. Delete the post from the backend.
3. Refresh the social feed upon successful deletion.
4. Display an error message in case of failures.

A computer screen with white text and blue text

Description automatically generated

Figure 39. Pseudocode of UC7.3

**Explanation:**

**handleDeletePostButtonClick(postID)**

**Purpose**: Manages the entire post deletion process.

**Steps**:

1. Calls deletePost with the postID.
2. If the deletion is successful:
   * Refreshes the social feed using refreshFeed.
3. Displays an error message for any failures using displayErrorMessage.

**deletePost(postID)**

**Purpose**: Sends a request to the backend to delete the specified post.

**Implementation**: Calls PostService.deletePost with the postID.

**refreshFeed()**

**Purpose**: Refreshes the social feed to remove the deleted post.

**displayErrorMessage(message)**

**Purpose**: Displays an error message to the user if the deletion process fails.

A diagram of a computer program

Description automatically generated with medium confidence

Figure 40. Sequence Diagram of UC7.3

1. **UC8: View Posts**

**Summary**

The pseudo-code outlines the process for displaying all posts in the social feed:

1. Initialize the user interface (UI).
2. Fetch all posts from the backend.
3. Convert backend entities to DTOs (Data Transfer Objects).
4. Render the posts on the UI.
5. Handle errors gracefully and display an error message if the operation fails.

A screen shot of a computer program

Description automatically generated

Figure 41. Pseudocode of UC8

**Explanation:**

**displayFeed()**

**Purpose**: Manages the entire process of fetching and displaying posts in the social feed.

**Steps**:

1. Initializes the UI by calling initUI.
2. Fetches all posts using the getAllPosts function.
3. Maps backend post entities to post DTOs via mapEntitiesToDTOs.
4. Passes the list of DTOs to renderPosts to display them.
5. Displays an error message using displayErrorMessage if an exception occurs.

**initUI()**

**Purpose**: Sets up the initial state of the UI.

**getAllPosts()**

**Purpose**: Fetches all posts from the backend.

**Implementation**:  Calls PostService.getAllPosts.

**mapEntitiesToDTOs(list)**

**Purpose**: Converts a list of backend entities to frontend-friendly DTOs.

**renderPosts(postList)**

**Purpose**: Renders the posts on the social feed.

**displayErrorMessage(message)**

A screenshot of a computer screen

Description automatically generated**Purpose**: Displays an error message if fetching or displaying posts fails.

Figure 42. UML Class Diagram of UC 7&8

A diagram of a social media service

Description automatically generated

Figure 43. UML Sequence Diagram of UC8

1. **UC9: View Profile**

**Summary:**

The pseudo-code implements the process for displaying a user's profile information in the application:

1. **User Request**: The user initiates the profile display action.
2. **Backend Interaction**: The application retrieves the user's profile data via ClerkSDK.
3. **Data Transformation**: The backend maps the retrieved UserDTO to a UserModel.
4. **Frontend Update**: The ProfileScreen renders the user's profile information using the UserModel
5. **Error Handling**: If any errors occur, an error message is displayed.

A screen shot of a computer program

Description automatically generated

Figure 44. Pseudocode of UC9

**Explanation:**

**displayProfile()**

**Purpose**: Orchestrates the process of fetching and displaying a user’s profile information.

**Steps**:

1. Calls getUser(userId) to fetch user details.
2. Uses displayProfileInfo(user) to update the UI with the fetched user details.
3. If an exception occurs, calls displayError-Message(message) to display an error notification.

**getUser(userId)**

**Purpose**: Retrieves user information from the backend.

**Implementation**:

* Invokes the ClerkSDK with GET /user/{userID}.
* Maps the retrieved UserDTO to UserModel.

**displayProfileInfo(user)**

**Purpose**: Renders the user profile details on the screen.

**displayErrorMessage(message)**

**Purpose**: Displays an error message if fetching or rendering profile information fails.

A screenshot of a computer

Description automatically generated

Figure 45. UML Class Diagram of UC9

A diagram of a mobile application

Description automatically generated

Figure 46. UML Sequence Diagram of UC9

1. **UC10: Edit Profile**

**Summary:**

The pseudo-code outlines the process for managing user profile editing:

1. Validate user input (first name, last name, image, cooking level, preferences).
2. Upload the profile image if provided.
3. Update the backend with user preferences.
4. Display a success message upon successful update
5. Display an error message in case of any failures.

A screen shot of a computer screen

Description automatically generated**Explanation:**

Figure . Pseudocode of UC10

**handleEditProfileFormSubmit(firstName, lastName, profileImage, cookingLevel, favoriteMeals, dietaryRestrictions)**

**Purpose:** Manages the profile editing process.  
 **Steps:**

1. Validates input using validateInput.
2. If input is valid:
   * Uploads the image (if provided) via uploadImage.
   * Creates a preferences DTO via createPreferencesDTO.
   * Updates the backend using updatePreferences.
   * Displays a success message using displaySuccessMessage.
3. Displays an error message for any failures using displayErrorMessage.

**validateInput(args)**

**Purpose:** Ensures all inputs are valid.

**Implementation:** Uses Input.validate() for field checks.

**uploadImage(profileImage)**

**Purpose:** Handles the uploading of the profile image.

**Steps:**

1. Obtains a signed URL via ImageService.getSignedURL.
2. Uploads the image file using ImageUploader.upload.
3. Returns the image URL or displays an error message.

**createPreferencesDTO(cookingLevel,favoriteMeals,dietaryRestrictions, imageUrl)**

**Purpose:** Constructs a DTO for user preferences.

**Implementation:** Maps the input data to a PreferencesDTO.

**updatePreferences(userId, preferencesDTO)**

**Purpose:** Updates user preferences on the backend.

**Implementation:** Sends a request to UserService.update-Preferences.

**displaySuccessMessage(message)**

**Purpose:** Notifies the user of successful operations.

**Implementation:** Calls Notification.showSuccess(message).

**displayErrorMessage(message)**

**Purpose:** Notifies the user of any errors.

**A white paper with black text

Description automatically generatedImplementation:** Calls Notification.showError(message).

Figure . UML Class Diagram of UC10

A blueprint of a building

Description automatically generated

Figure . UML Sequence Diagram of UC10

1. **UC21: View AI Chat List**

**Summary:**

The pseudo-code describes the process for retrieving and displaying a user's chat list:

1. User Request: The user initiates a request to view their chat list.
2. Backend Interaction: The system sends a request to fetch the chat list using the ChatController.
3. Data Transformation: Backend entities (ChatEntity) are mapped to DTOs (ChatDTO) for frontend use.
4. Frontend Update: The chat list is displayed in the user interface using ChatListScreen.renderChatList.

Error Handling: If an error occurs at any stage, a user-friendly error message is displayed using ChatListScreen.display-ErrorMessage.

A screen shot of a computer program

Description automatically generated

Figure 50. Pseudocode of UC21

**Explanation:**

**displayChatList()**

**Purpose**: Orchestrates the entire process of fetching and displaying the chat list.

**Steps**:

1. Calls ChatController.getChatList(userID) to fetch the chat list from the backend.
2. Checks the response status code:
   * If the status code is 200, the body (chat list) is passed to ChatListScreen.renderChatList(chatList).
   * Otherwise, raises an exception with a failure message.
3. If any exception occurs, it is caught and passed to ChatListScreen.displayErrorMessage.

**renderChatList(chatList)**

**Purpose**: Displays the chat list in the user interface.

**Implementation**:

* Iterates through the chatList (a list of ChatDTO objects).
* For each chat in the list, call UI.displayChat(chat) to render it on the screen.

**displayErrorMessage(msg)**

**Purpose**: Displays error messages for the user.

**Implementation**: Calls UI.showError(msg) to show a user-friendly error message.

**ChatController.getChatList(userID)**

**Purpose**: Handles the request to retrieve the chat list from the service layer.

**Implementation**: Calls ChatService.getChatList(userID) to fetch the chat list.

**ChatService.getChatList(userID)**

**Purpose**: Retrieves chat list data from the repository and transforms it for frontend use.

**Implementation**:

1. Calls ChatRepository.getChatList(userID) to fetch raw chat entities (ChatEntity).
2. Passes the list of ChatEntity objects to ChatService-Impl.mapEntitiesToDTOs.

**ChatServiceImpl.mapEntitiesToDTOs(chatEntities)**

**Purpose**: Transforms backend chat entities into frontend-friendly DTOs.  
 **Implementation**: Maps each ChatEntity in the list to a ChatDTO object containing only relevant attributes.

**A screenshot of a computer

Description automatically generated**

Figure 51. UML Class Diagram of UC21

**A screenshot of a chat service

Description automatically generated**

Figure 52. UML Sequence Diagram of UC21

UC22 Manage AI Chats is considered in 2 parts. UC22.1 is create AI chat and UC22.2 is delete AI chat. These two shares the same class diagram. It is included at the end of the UC22.2.

1. **UC22.1: Manage AI Chats (Create AI Chat)**

**Summary:**

This use case allows a user to create a new chat session.

1. The **user** clicks the "Create New Chat" button.
2. The **ChatController** processes the request, delegating it to the **ChatService**.
3. The **ChatServiceImpl** interacts with the **ChatRepository** to create a new ChatEntity and map it to a ChatDTO.
4. The new chat (ChatDTO) is returned to the front end.
5. The frontend calls updateChatList to display the new chat.
6. If an error occurs, an error message is displayed using displayErrorMessage.

A screenshot of a computer program

Description automatically generated

Figure 53. Pseudocode of UC22.1

**Explanation:**

**handleCreateNewChatButtonClick**()

**Purpose**: Triggers the chat creation process.

**Steps**:

1. Sends the request to the backend for processing.
2. Updates the chat list with the newly created chat.
3. Displays error messages for any failures.

**updateChatList**(**chatModel**)

**Purpose**: Updates the UI with the newly created chat.

**ChatController.createChat(userID)**

**Purpose**: Forwards the chat creation request to the service layer.

**ChatService.createChat(userID)**

**Purpose**: Handles the business logic for chat creation.

**Steps**:

1. Interacts with the repository to store the chat.
2. Converts the stored chat entity into a ChatDTO.

**ChatServiceImpl.mapEntityToDTO(chatEntity)**

**Purpose**: Converts a backend ChatEntity to a frontend-friendly ChatDTO.

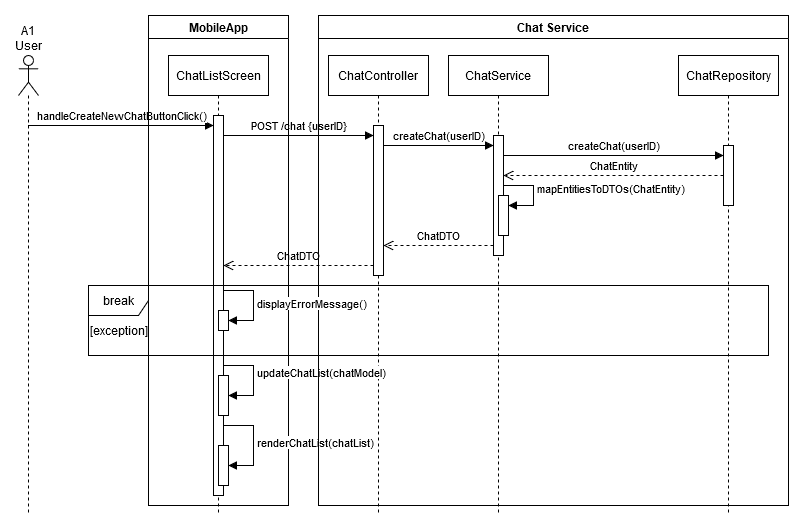
****

Figure 54. UML Sequence Diagram of UC22.1

1. **UC22.2: Manage AI Chats (Delete AI Chat)**

**Summary:**

This use case enables the user to delete an existing chat.

1. The user clicks the "Delete Chat" button.
2. The ChatController processes the delete request and delegates it to the ChatService.
3. The ChatServiceImpl interacts with the ChatRepository to remove the chat record.
4. Upon successful deletion, the frontend refreshes the chat list by invoking updateChatList.
5. If an error occurs (e.g., the chat does not exist or cannot be deleted), an error message is displayed using displayErrorMessage.

A screen shot of a computer program

Description automatically generated

Figure 55. Pseudocode of UC22.2

**Explanation:**

**handleDeleteChatButtonClick(chatID)**

**Purpose:** Initiates the deletion process for the selected chat.

**Steps:**

1. Sends a delete request to the backend.
2. Updates the frontend UI to reflect the latest state of the chat list.
3. Displays error messages for any failures.

**updateChatList**()

**Purpose:** Refreshes the chat list after a deletion to ensure the UI reflects the latest state.

**ChatController.deleteChat(chatID)**

**Purpose:** Forwards the delete request to the service layer.

**ChatService.deleteChat(chatID)**

**Purpose:** Contains the business logic for deleting a chat.

**ChatRepository.deleteChat(chatID)**

**Purpose:** Executes the database query to remove the chat record.

A screenshot of a chat application

Description automatically generated

Figure 56. UML Sequence Diagram of UC22.2

A diagram of a computer code

Description automatically generated with medium confidence

Figure . UML Class Diagram of UC22.2

# Discussions

## Limitations and Constraints

* **Time Constraints:** Balancing academic responsibilities and part-time jobs continued to limit the time we could dedicate to SDD preparation. Scheduling meetings was especially challenging, but through careful prioritization, we managed to convene several in-person sessions.
* **Infrastructure Issues:** During late-night work sessions in the A building, the water fountain was out of order, making it difficult to stay hydrated. This resulted in team members frequently leaving the workspace to find water, which disrupted workflow and extended preparation times

iç mekan, şişe, beyaz eşya, kişi, şahıs içeren bir resim

Açıklama otomatik olarak oluşturuldu

Figure 58. The broken water fountain in the A building

* **Limited Refreshment Options**: On Sunday night, December 22, 2024, the MA Starbucks closed early, leaving us without access to our preferred beverages. As a result, we had to rely on the Ankara Otomat vending machine's *sade çekirdek americano*, which was less enjoyable due to its taste and raised concerns about hygiene and the quality of the ground beans used. This decreased productivity and morale during critical work hours.

A black and white screen with white text

Description automatically generated

Figure 59. Receipt for vending machine coffee purchase at Ankara Otomat

* **Essential Meal Requirement:** On the night of December 22 to December 23, the team visited **ANKARA GECE DÖNERCİSİ ÇAYYOLU ŞUBESİ** to maintain energy levels. Group members Cemal, Batuhan, and Baturalp, who cannot work effectively without 200 grams of lamb döner daily, relied on this meal for physical and mental refreshment. This meal energized the team, allowing them to work with focus and dedication.

metin, makbuz içeren bir resim

Açıklama otomatik olarak oluşturuldu

Figure 60. Receipts for lamb döner meals from ANKARA GECE DÖNERCİSİ

## Health and Safety Issues

* **Prolonged Work Hours:** Preparing the SDD required extended hours of screen time, leading to neck and back pain, as well as general fatigue.
* **Hydration Challenges:** The broken water fountain in the A building resulted in health issues, as team members struggled to stay hydrated. Bottled water was occasionally purchased but added to overall costs and inconvenience.
* **Fast Food and Caffeine Dependency**: Similar to earlier deliverables, the reliance on fast food and energy drinks persisted, further impacting health.
* **Poor Beverages**: The reliance on vending machine coffee (*sade çekirdek americano*) due to the early closure of Starbucks added potential health concerns due to the unknown cleanliness of the machine and the quality of ingredients.
* **Weightlifting and Physical Discomfort:** Baturalp started a weight-lifting program at the Bilkent Dormitories Sports Hall during this period. As a result, he experienced significant muscle pain and soreness, which temporarily affected his performance and ability to contribute effectively during work sessions.

iç mekan, duvar, kişi, şahıs, mobilya içeren bir resim

Açıklama otomatik olarak oluşturuldu

Figure 61. Baturalp recovering from muscle soreness after a weight-lifting session

## Legal Issues

* No legal issues were encountered during the SDD preparation process.

## Economic Issues and Constraints

* **Commuting Costs:** Due to team members meeting at different locations, in-person meetings require spending on gas and transportation. Rising fuel prices further strained personal budgets.
* **Late-Night Food Costs:** Late-night work sessions often required trips to local döner shops, adding to travel and meal expenses.
* **Hydration Costs:** The lack of a functioning water fountain led to additional expenses for bottled water purchased during preparation sessions.

## Sustainability

* We continued to prioritize digital tools, such as Google Docs and Slack, to reduce paper waste and ensure efficient collaboration.
* For on-campus meetings, team members opted to walk instead of driving whenever feasible, minimizing carbon emissions from transportation.

## Ethical Issues

* Team members occasionally used employer-provided resources, such as laptops, to work on the SDD. While this improved efficiency, it raises minor ethical considerations regarding the use of work equipment for academic purposes.

## Multidisciplinary Collaboration

* No multidisciplinary collaborations were undertaken during the preparation of the SDD. All tasks were managed within the team's expertise in software design and development.

# References

|  |  |
| --- | --- |
| [1] | A. Çelik, C. F. Dağ, B. Duras, D. Durmaz and B. Sönmez, "Software Requirements Specification v2 of Team 11," Ankara, 2024. |
| [2] | A. Çelik, C. F. Dağ, B. Duras, D. Durmaz and B. Sönmez, "Software Project Management Plan v2 of Team 11," Ankara, 2024. |

1. GenAI tool: ChatGPT 4.o

   Prompt: “Formalize the given text which is the Executive Summary of our SDD”

   Rationale: To improve and make more formal the language of the text [↑](#footnote-ref-1)