M3 - Requirements and Design

1. Change History

Date	Modified Section(s)	Change Description	Rationale
2025-03-02	3.1 Use Case Diagram	Use Case diagram modified	Corrected diagram to combine Session use cases and added Authentication use case
2025-03-02	3.3 Functional Requirements	3.3.7 Authenticate added	To properly document the authentication workflow, the functional requirements for Authentication flow were added
2025-03-02	4.2 Databases	Modified databases and tables	We only need one database with multiple tables
2025-03-02	4.5 Dependencies Diagram	Changed component dependencies	Merged SessionsDB and UserDB into one DB for the entire system
2025-03-02	4.6 Functional Requirements Sequence Diagrams	Authentication sequence diagrams added	Log In and Sign Out sequence diagrams were added
2025-03-15	3.5 Non-Functional Requirements	Added new non-functional requirements	Added new non-functional requirements for performance, reliability, and error recovery
2025-03-30	3.1 Use Case Diagram	Use Case diagram modified	Corrected diagram to separate Manage Groups and Friends use case
2025-03-30	4.1 Main Components	Modified Interfaces	Updated the interfaces to better align with implementation
2025-03-30	4.5 Dependencies Diagram	Dependencies Diagram modified	Corrected diagram to only have Group Management
2025-03-30	4.6 Functional Requirements Sequence Diagrams	Modified a few sequence diagrams	Modified a few sequence diagrams for Manage Sessions, Manage Friends, and Manage Groups use cases to better align with the updated components and interfaces

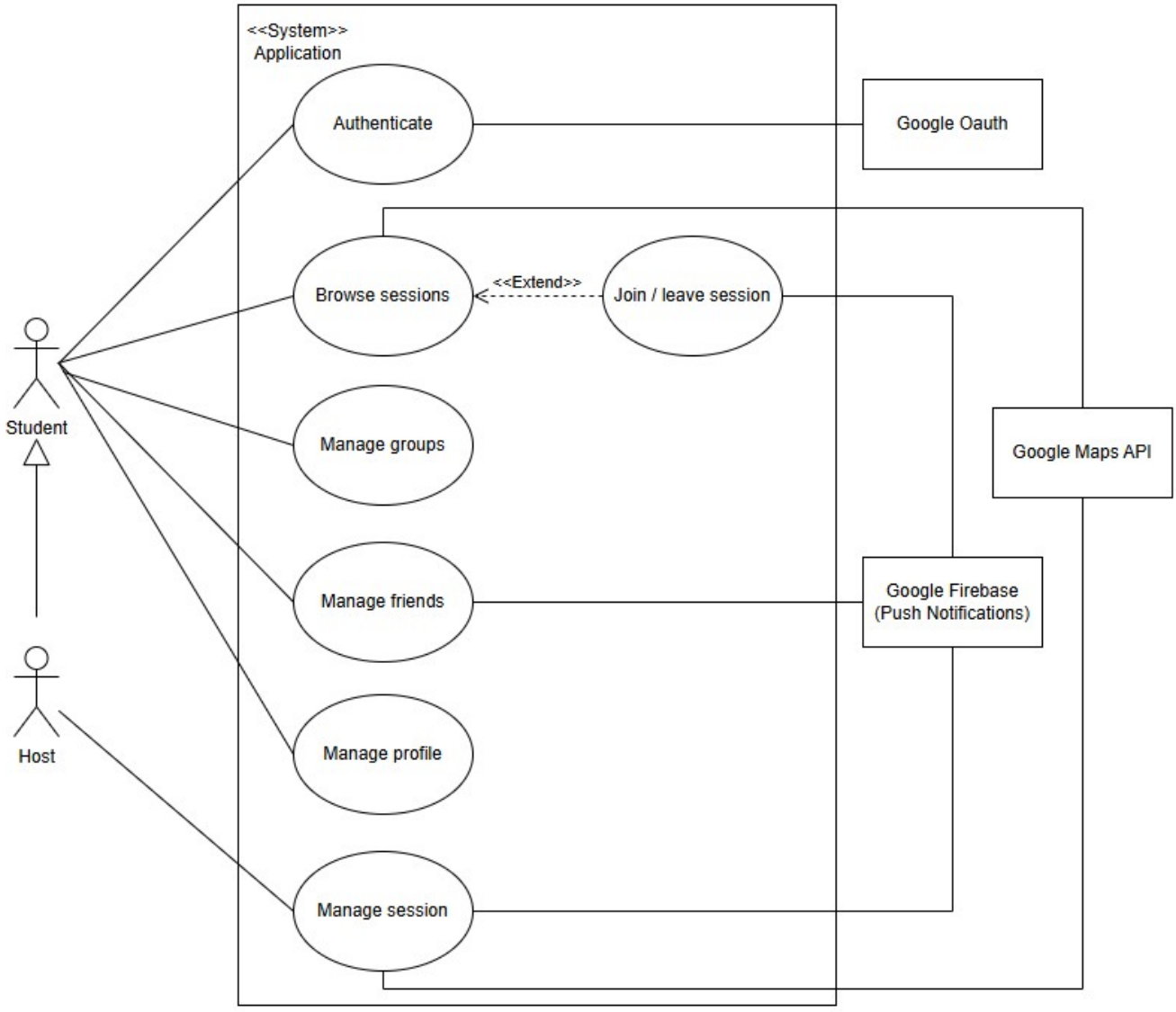
Date	Modified Section(s)	Change Description	Rationale
2025-03-31	4.8. Main Project Complexity Design	Elaborated on main complexity	Elaborated further on the main complexity, describing the algorithm in more detail

2. Project Description

Study Wimme targets university students who seek a collaborative study environment to feel motivated and accountable. Many students prefer studying with a companion to help them stay focused, organized, and accountable while studying. However, traditional study groups can be difficult to organize on a regular group chat and can create social barriers to organizing group study sessions. Study Wimme aims to bridge this gap by providing a social platform to efficiently organize study sessions with friends and like-minded peers.

3. Requirements Specification

3.1. Use-Case Diagram



3.2. Actors Description

1. **Student**: They are a student who uses the app to join a study session with their friends or with people nearby.
2. **Host**: A student can become a host. They are a student who wants to organize a study session with their friends or publicly.

3.3. Functional Requirements

- **General Failure Scenarios:**
    - Database failure scenarios exist in all functional requirements.
        - If any database operation fails due to database service not being available or encountering an unexpected error, a message will be displayed to the user informing them of the database error and to try again
    - Network failure scenario exists in all functional requirements.
        - If network access is lost at any point, an error message is displayed informing the user that an internet connection is required for app functionality

1. **Manage Profile**
    - **Overview**:
        1. Create profile
        2. Read profile
        3. Edit profile
        4. Delete profile
    - **Detailed Flow for Each Independent Scenario**:
        1. **Create Profile**:
            - **Description**: The actor can create their profile by specifying their username, year, and faculty
            - **Primary actor(s)**: Student
            - **Main success scenario**:
                1. The actor clicks the create profile button.
                2. The system displays empty, editable fields about their profile.
                3. The actor clicks the fields and enters the appropriate information
                4. The actor clicks the save button
                5. The inputted data gets populated in the database for that user.
                6. The system displays that the profile has been created successfully
            - **Failure scenario(s)**:
                - 3a. User enters invalid text for respective fields (string for year, number for major, etc..)
                    - 3a1. Message is displayed to user informing them that they have entered invalid characters for the given field(s) and to fix it before submitting is allowed

2. **Read Profile**:

    - **Description**: The actor can view their profile details.
    - **Primary actor(s)**: Student
    - **Main success scenario**:

        1. The actor clicks on their profile button
        2. The system retrieves the actor's profile information from the database
        3. The system displays information about their profile: username, year, and faculty

    - **Failure scenario(s)**:

        - See general failure requirements

3. **Edit profile**:

    - **Description**: The actor can change the details of their profile profile
    - **Primary actor(s)**: Student
    - **Main success scenario**:

        1. The actor clicks the edit button on their profile
        2. The system retrieves the actor's profile information from the database
        3. The system displays information about their profile, such as username, year, and faculty as editable fields
        4. The actor clicks on the field they want to modify and changes it
        5. The actor clicks the save button
        6. The inputted data gets updated in the database for that user
        7. The system displays that the changes have been saved successfully

    - **Failure scenario(s)**:

        - 4a. User enters invalid text for respective fields (string for year, number for major, etc..)
            - 4a1. Message is displayed to user informing them that they have entered invalid characters for the given field(s) and to fix before they can save their edited details

4. **Delete profile**:

    - **Description**: The actor can delete their profile
    - **Primary actor(s)**: Student
    - **Main success scenario**:

        1. The actor clicks the delete button on their profile
        2. The system displays a popup asking to confirm deletion of their profile
        3. The actor clicks the confirm deletion button

4. The system deletes the user profile entry from the database
5. The system displays that the profile has been deleted successfully

- **Failure scenario(s)**:
  - 2a. User cancels deletion:
    - 2a1. System closes confirmation popup
    - 2a2. Returns to profile page with no changes

2. **Manage friends**

- **Overview**:
  1. Add friends
  2. Read friends
  3. Delete friends
- **Detailed Flow for Each Independent Scenario**:

  1. **Add friends**:

  - **Description**: The actor can search for friends and add them to their friends list
  - **Primary actor(s)**: Student
  - **Main success scenario**:

    1. Student A clicks the friends button
    2. The system loads the friends list page that displays a list of the student's friends and a text input to add friends
    3. Student A enters the username for Student B and clicks the send request button to send a friend request to Student B
    4. Student B accepts the friend request
    5. The system updates the database, adding Student B to Student A's friend list and vice versa

  - **Failure scenario(s)**:
    - 3a. The user did not enter any search string
      - 3a1. System displays an error message that says empty search strings are not allowed
    - 3b. No users are found
      - 3b1. System displays a message saying no users with username matching the given search string was found, as well as a reminder to double-check input

  2. **Read friends**:

  - **Description**: The actor can view their friends list
  - **Primary actor(s)**: Student
  - **Main success scenario**:

1. The actor clicks the friends button
2. The system retrieves the friends list from the database and displays this information

- **Failure scenario(s)**:
  - 2a. The user has no friends
    - 2a1. Message is displayed to the user that they currently have no friends added

3. **Delete Friends**:

- **Description**: The actor can delete friends from their friends list
- **Primary actor(s)**: Student
- **Main success scenario**:

1. The actor clicks the friends button
2. The system retrieves the friends list from the database and displays this information
3. The actor clicks on a friend
4. The system retrieves the friend from the database and displays this information
5. The actor clicks the delete button
6. The system displays a popup asking to confirm deletion
7. The actor clicks the confirm deletion button
8. The system deletes the friend entry from list in the database
9. The system displays that the friend has been deleted successfully

- **Failure scenario(s)**:
  - 2a. User has no friends
    - 2a1. Popup informs user that they have no friends to delete
    - 2a2. Returns to friends list with no changes
  - 6a. User cancels deletion
    - 6a1. System closes confirmation popup
    - 6a2. Returns to friends list with no changes

3. **Manage groups**

- **Overview**:
  1. Add groups
  2. Read groups
  3. Edit groups
  4. Delete groups
- **Detailed Flow for Each Independent Scenario**:

1. **Add groups**:

- **Description**: The actor can create groups, which will consist of their friends

- **Primary actor(s)**: Student
- **Main success scenario**:
    1. Student clicks the groups button
    2. The system loads the groups page that displays Student's created groups
    3. Student creates a group with their friends
    4. The system adds the newly created group to the database
- **Failure scenario(s)**:
    - 3a. The user has no friends when creating the group
        - 3a1. Inform the user that they can only create groups with friends and to add some friends before trying again
    - 3b. The user already has a group with the same people
        - 3b1. Message pops up informing them that a preexisting group with the same people exist and tells them the name of the group

2. **Read groups**:
    - **Description**: The actor can view their groups list
    - **Primary actor(s)**: Student
    - **Main success scenario**:
        1. The actor clicks the groups button
        2. The system retrieves the groups list from the database and displays this information
    - **Failure scenario(s)**:
        - 2a. The user has no groups
            - 2a1. Message is displayed to the user that they currently have no groups created
3. **Edit groups**:
    - **Description**: The actor can edit which users are part of a group they created
    - **Primary actor(s)**: Student
    - **Main success scenario**:
        1. The actor clicks the groups button
        2. The system retrieves the groups list from the database and displays this information
        3. The actor clicks the edit button on the groups list
        4. The system displays a modal that allows the actor to add or remove members from the group
        5. The user clicks the save button on the modal
        6. The system updates the database with edited group

- **Failure scenario(s)**:
    - 2a. The user has no created groups
        - 2a1. Message is displayed to a user that they currently have no groups created
    - 5a. The user removes all members from a group
        - 5a1. Message is displayed to a user that they have removed everyone, and if they would like to just delete the group
    - 5b. The edited group matches a preexisting group
        - 5b1. Message pops up informing them that a preexisting group with the same people exist and tells them the name of the group

4. **Delete groups**:

- **Description**: The actor can delete groups from their groups list
- **Primary actor(s)**: Student
- **Main success scenario**:
    1. The actor clicks the groups button
    2. The system retrieves the groups list from the database and displays this information
    3. The actor clicks on a group
    4. The system retrieves the group from the database and displays this information
    5. The actor clicks the delete button
    6. The system displays a popup asking to confirm deletion
    7. The actor clicks the confirm deletion button
    8. The system deletes the group entry from list in the database
    9. The system displays that the group has been deleted successfully
- **Failure scenario(s)**:
    - 2a. User has no groups
        - 2a1. Popup informs user that they have no groups to delete
        - 2a2. Returns to groups list with no changes
    - 6a. User cancels deletion
        - 6a1. System closes confirmation popup
        - 6a2. Returns to groups list with no changes

4. **Browse sessions**

- **Overview**:
    1. Browse sessions
- **Detailed Flow for Each Independent Scenario**:
    1. **Browse sessions**:
        - **Description**: The actor can browse for sessions hosted by their friends or public sessions on a graphical map interface

- **Primary actor(s)**: Student
- **Main success scenario**:
    1. The actor clicks on the browse sessions button
    2. The system checks for location permissions and if not enabled, requests it with a modal
    3. The system retrieves all sessions available to the actor from the database within a specific timeframe and radius of the user's location
    4. The system displays the sessions on a map
- **Failure scenario(s)**:
    - 2a. Location services disabled
        - 2a1. System displays error that location access is required
        - 2a2. Prompts user to enable location services
    - 3a. There are no sessions available
        - 3a1. System displays message that no sessions are available at the moment, and prompts user to create a session

5. **Manage session**

- **Overview**:
    1. Create session
    2. Delete session
- **Detailed Flow for Each Independent Scenario**:
    1. **Create session**:
        - **Description**: The actor can create a private or public study session with a set time range, name, location, and optional description
        - **Primary actor(s)**: Host
        - **Main success scenario**:
            1. The actor clicks on the create session button
            2. The system displays empty, editable fields for time range, location, and description. It also includes a toggle that lets the actor choose if the session is private or public
            3. The actor clicks the fields, enters the appropriate information and specifies whether the session is public or private. If the session is private, the actor chooses which friends or groups they broadcast the session to
            4. The actor clicks the create button
            5. The inputted data gets populated in the database for the new session
            6. The system displays that the session has been created successfully. If the session is private, the selected friends/groups are notified
        - **Failure scenario(s)**:
            - 3a. User enters invalid information for session (letters for time, symbols for anything, date in the past, etc...)

        - 3a1. Message informing user that they have entered invalid information for a field and identifying that the field and expected format of input is

    2. **Delete session**:

        - **Description**: The host can end a session
        - **Primary actor(s)**: Host
        - **Main success scenario**:

            1. The actor clicks the session which they want to delete
            2. The system retrieves the session information from the database and displays the information
            3. The actor clicks the delete button
            4. The system displays a popup asking to confirm deletion of this session
            5. The actor clicks the confirm deletion button
            6. The system deletes the session entry from the database
            7. The system displays that the session has been deleted successfully

        - **Failure scenario(s)**:

            - 4a. User cancels deletion:
                - 4a1. System closes confirmation popup
                - 4a2. Returns to session page with no changes

6. **Join/Leave Session**

    - **Overview**:
        1. Join Session
        2. Leave Session
    - **Detailed Flow for Each Independent Scenario**:
        1. **Join Session**:

            - **Description**: The actor can join sessions which are available to them
            - **Primary actor(s)**: Student
            - **Main success scenario**:

                1. The actor clicks on the session which they want to join
                2. The system retrieves the session information from the database and displays the information
                3. The actor clicks the join button
                4. The system updates the database to include the actor as an attendee
                5. The system displays that the actor has joined the session

            - **Failure scenario(s)**:

                - 1a. User already joined
                    - 1a1. System displays message that user is already in session

        - 1a2. Returns to session page
    2. **Leave Session**:
        - **Description**: The actor can leave a session which they have joined.
        - **Primary actor(s)**: Student
        - **Main success scenario**:
            1. The actor clicks on the session which they have joined
            2. The system retrieves the session information from the database and displays the information
            3. The actor clicks the leave button
            4. The system updates the database to remove the actor as an attendee
            5. The system displays that the actor has left the session
        - **Failure scenario(s)**:
            - 3a. Session no longer exists:
                - 3a1. System displays error that session cannot be found
                - 3a2. Returns to sessions list

7. **Authenticate**
    - **Overview**:
        1. Sign in with Google
        2. Sign out
    - **Detailed Flow for Each Independent Scenario**:
        1. **Sign in with Google**:
            - **Description**: The actor can sign in using their Google account
            - **Primary actor(s)**: Student
            - **Main success scenario**:
                1. The actor clicks the Google sign-in button on the login screen
                2. The system displays a Google account selection dialog
                3. The actor selects their account
                4. The system verifies the Google authentication token
                5. The system checks if the user already exists in the database
                6. If the user exists and has completed their profile, the system navigates to the sessions screen
                7. If the user exists but hasn't completed their profile, the system navigates to the user settings screen
                8. If the user doesn't exist, the system creates a new user in the database and navigates to the user settings screen
            - **Failure scenario(s)**:
                - 3a. User cancels Google sign-in

        - 3a1. System returns to login screen without changes
    - 4a. Google authentication fails
        - 4a1. System displays error message about authentication failure
        - 4a2. Returns to login screen
    - 5a. Network connectivity issue
        - 5a1. System displays message about connection issue
        - 5a2. Prompts user to check their internet connection and try again

2. **Sign out**:

    - **Description**: The actor can sign out from their account
    - **Primary actor(s)**: Student
    - **Main success scenario**:

        1. The actor clicks the sign out button in the user settings screen
        2. The system removes user authentication tokens and session data from local storage
        3. The system unregisters device token from notification service
        4. The system redirects the user to the login screen

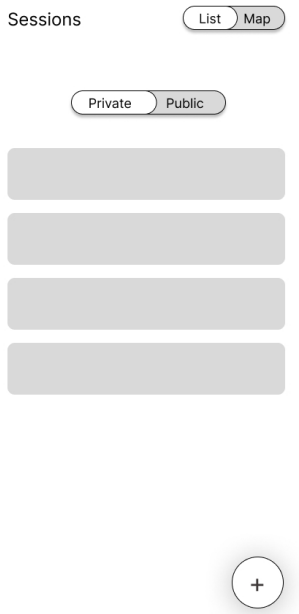
    - **Failure scenario(s)**:

        - 2a. Failed to clear local user data
            - 2a1. System attempts to sign out again
            - 2a2. If still unsuccessful, system displays error message
        - 3a. Failed to unregister device token
            - 3a1. System logs the error but continues with sign out process

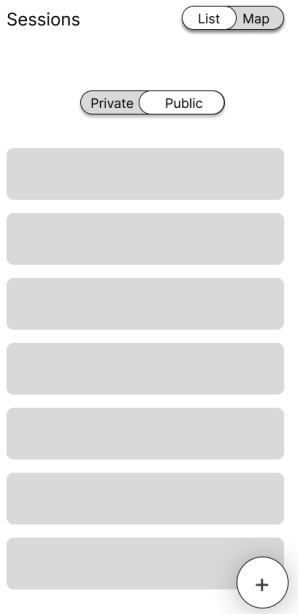
3.4. Screen Mockups

Sessions View List Private	Sessions View List Public	Sessions View Map	Manage Sessions

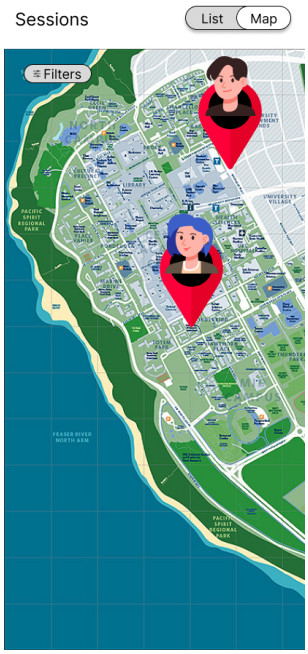
Sessions View List Private



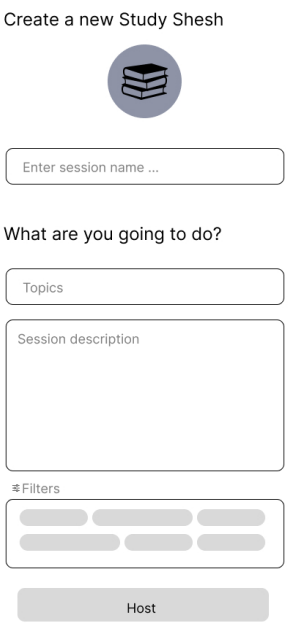
Sessions View List Public



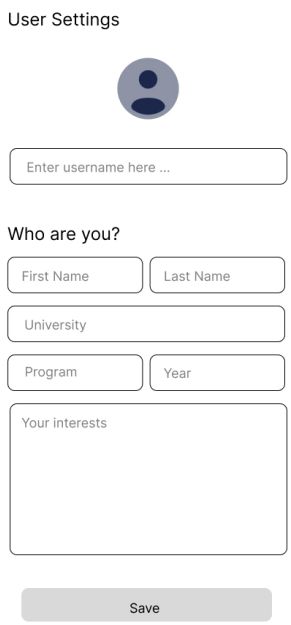
Sessions View Map



Manage Sessions



User Settings



Friends

Groups

Edit Groups

3.5. Non-Functional Requirements

1. **Real-time Updates**
    - **Description**: The system must update session information and notifications within 5 seconds
    - **Justification**: This is critical for maintaining accurate session information and participant coordination
2. **Location Accuracy**
    - **Description**: The system must maintain location accuracy within 10 meters for session locations
    - **Justification**: This is essential for students to find study locations efficiently
3. **Performance**

- **Description**: The server endpoints should have a response time of less than 300 milliseconds for 95% of the requests.
- **Justification**: Fast response times are crucial for providing a good user experience. Users are likely to abandon the application if it takes too long to respond.

4. **Reliability and Error Recovery**
   - **Description**: The application should be able to recover gracefully from errors. In the event of bad requests, the system should provide a clear error message to the user without crashing the app and loosing any user data.
   - **Justification**: This is essential for ensuring that user data is not lost and that the system is reliable.

4. Design Specification

4.1. Main Components

1. **User Management**
   - **Purpose**: Handles profile management and user's friends list.
   - **Interfaces**:
     1. `UserID createProfile(UserDTO)`
        - **Purpose**: Creates a new user profile
     2. `void updateProfile(UserID, UserDTO)`
        - **Purpose**: Updates user profile information
     3. `void deleteProfile(UserID)`
        - **Purpose**: Deletes the user's profile
     4. `User viewProfile(UserID)`
        - **Purpose**: Views the user's profile
     5. `void sendFriendRequest(SenderID, ReceiverID)`
        - **Purpose**: Send friend request to another user
     6. `void handleFriend(Decision, SenderID, ReceiverID)`
        - **Purpose**: Adds user to friends list if accepted, otherwise remove the friend request
     7. `List<User> getFriends(UserID)`
        - **Purpose**: Get all friends for user
     8. `List<User> getFriendRequests(UserID)`
        - **Purpose**: Get all friend requests for user
     9. `void removeFriend(UserID, FriendID)`
        - **Purpose**: Remove a friend that the user has selected

2. **Group Management**
   - **Purpose**: Handles the creation, modification, and deletion of groups.
   - **Interfaces**:
     1. `GroupID createGroup(UserID, name)`
        - **Purpose**: Create a group using the given group name
     2. `void editGroup(GroupID, List<UserID> members)`
        - **Purpose**: Make changes to an existing group
     3. `void deleteGroup(GroupID)`
        - **Purpose**: Delete a preexisting group

4. `List<Group> getGroups(UserID)`
    - **Purpose**: Get all groups made by that user

3. **Session**

- **Purpose**: Handles the creation, modification, and deletion of study sessions
- **Interfaces**:
    1. `SessionID createSession(SessionDTO)`
        - **Purpose**: Creates new study session
    2. `void deleteSession(SessionID)`
        - **Purpose**: Deletes existing study session
    3. `void joinSession(UserID, SessionID)`
        - **Purpose**: Adds a user to the joined attribute of a session
    4. `void leaveSession(UserID, SessionID)`
        - **Purpose**: Removes from a user from the joined attribute of a session
    5. `void sendJoinNotificationToCreator(SessionId)`
        - **Purpose**: Send notification to creator of session notifying that someone joins
    6. `List<Session> getFilteredSessions(filters: FilterOptions)`
        - **Purpose**: Retrieves sessions based on applied filters

4. **Authentication**

- **Purpose**: Handles user authentication, verification, and account management
- **Interfaces**:
    1. `UserID signIn(GoogleToken)`
        - **Purpose**: Authenticates a user using their Google account
    2. `boolean verifyUserProfile(UserID)`
        - **Purpose**: Checks if a user has completed their profile
    3. `void signOut(UserID)`
        - **Purpose**: Signs out a user from the application
    4. `DeviceToken registerDeviceToken(UserID, Token)`
        - **Purpose**: Registers a device for push notifications
    5. `void unregisterDeviceToken(DeviceToken)`
        - **Purpose**: Removes a device from receiving push notifications

4.2. Databases

1. **DB**
    - **Purpose**:
        - `User table` - Stores user profile information (name, faculty, year, friends).
        - `Group table`- Stores the group that have been made by users along with the group members.
        - `Device table` - Stores the association between a user and their device token for push notifications.
        - `Sessions table` - Stores session data including session ID, creator ID, invitee ID, and session details.

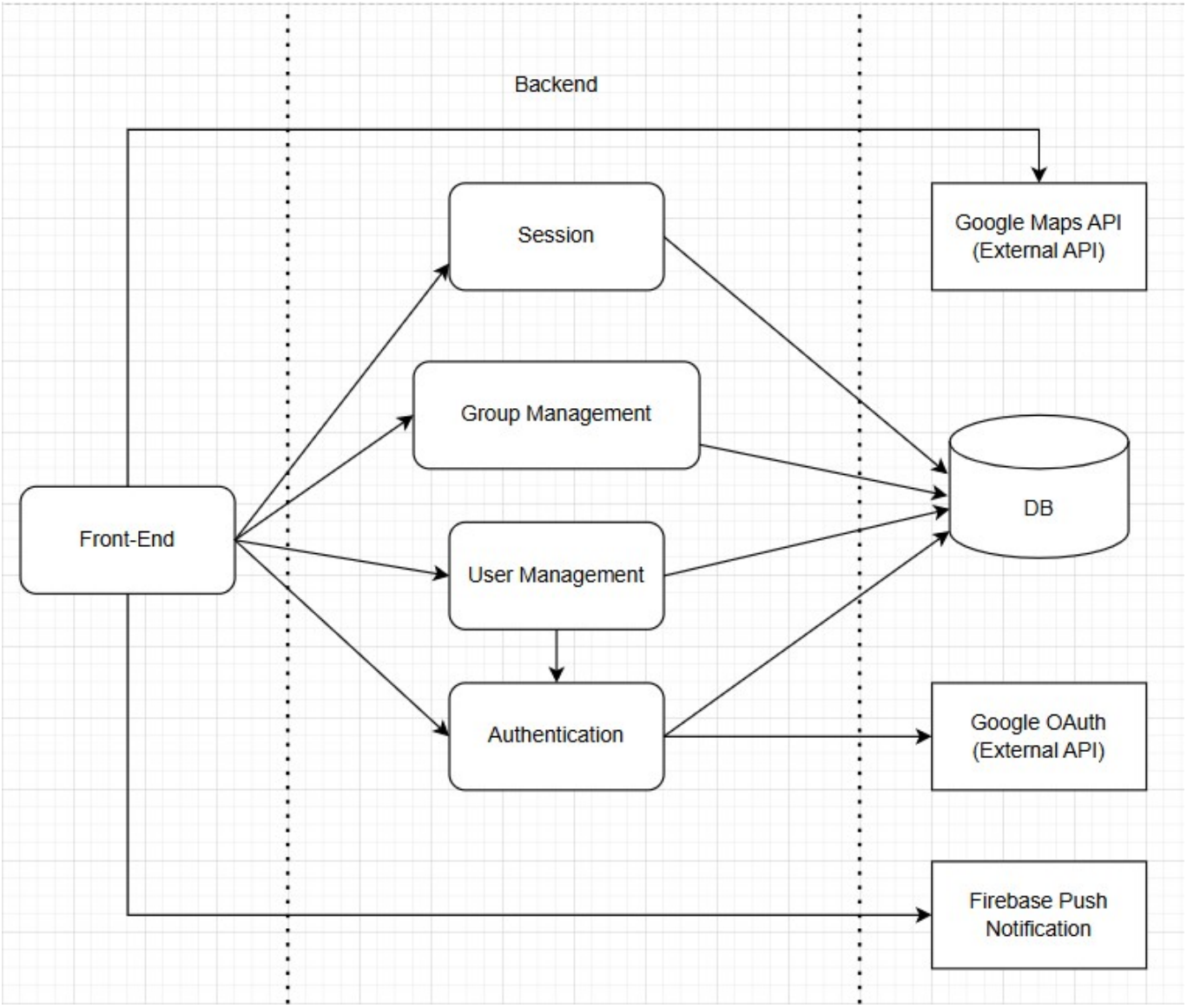
4.3. External Modules

1. **Google Maps API**
    - **Purpose**: Provides location services and map visualization
2. **Google OAuth**
    - **Purpose**: Handles user authentication and security
3. **Google Firebase Push Notifications**
    - **Purpose**: Provides push notifications to the users as a way to relay information

4.4. Frameworks

1. **Amazon Web Services (AWS)**
    - **Purpose**: Using AWS EC2, we are able to deploy our server backend
    - **Reason**: It was covered in the tutorials and we have the technical knowledge from M1. It is simple and easy to deploy with good performance and it has a generous free tier
2. **Kotlin/Android**
    - **Purpose**: Native Android mobile application development
    - **Reason**: Required as per course syllabus
3. **Node.js with Express**
    - **Purpose**: Backend REST API server
    - **Reason**: It is lightweight and enables fast development and easy deployment

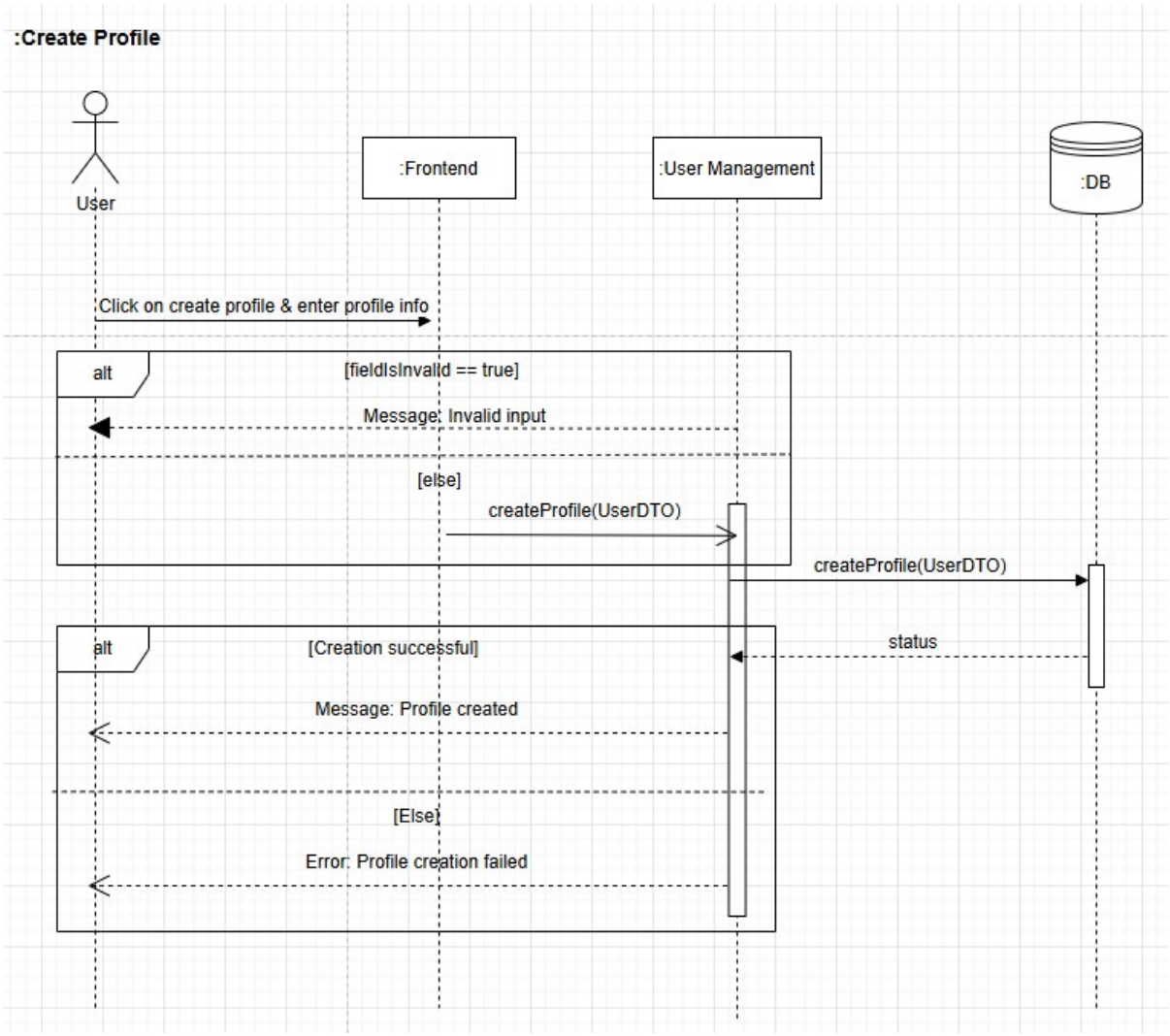
4.5. Dependencies Diagram



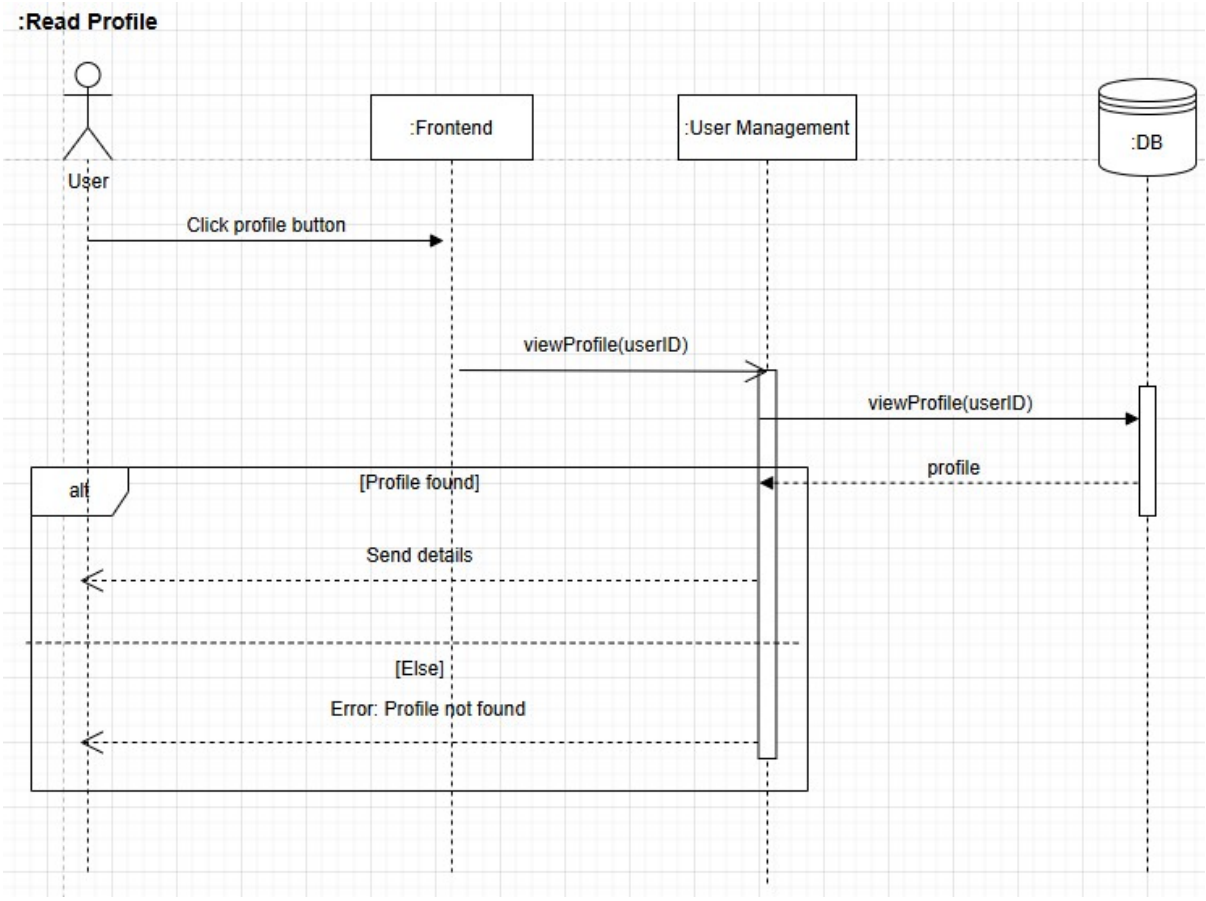
4.6. Functional Requirements Sequence Diagram

1. **Manage Profile**

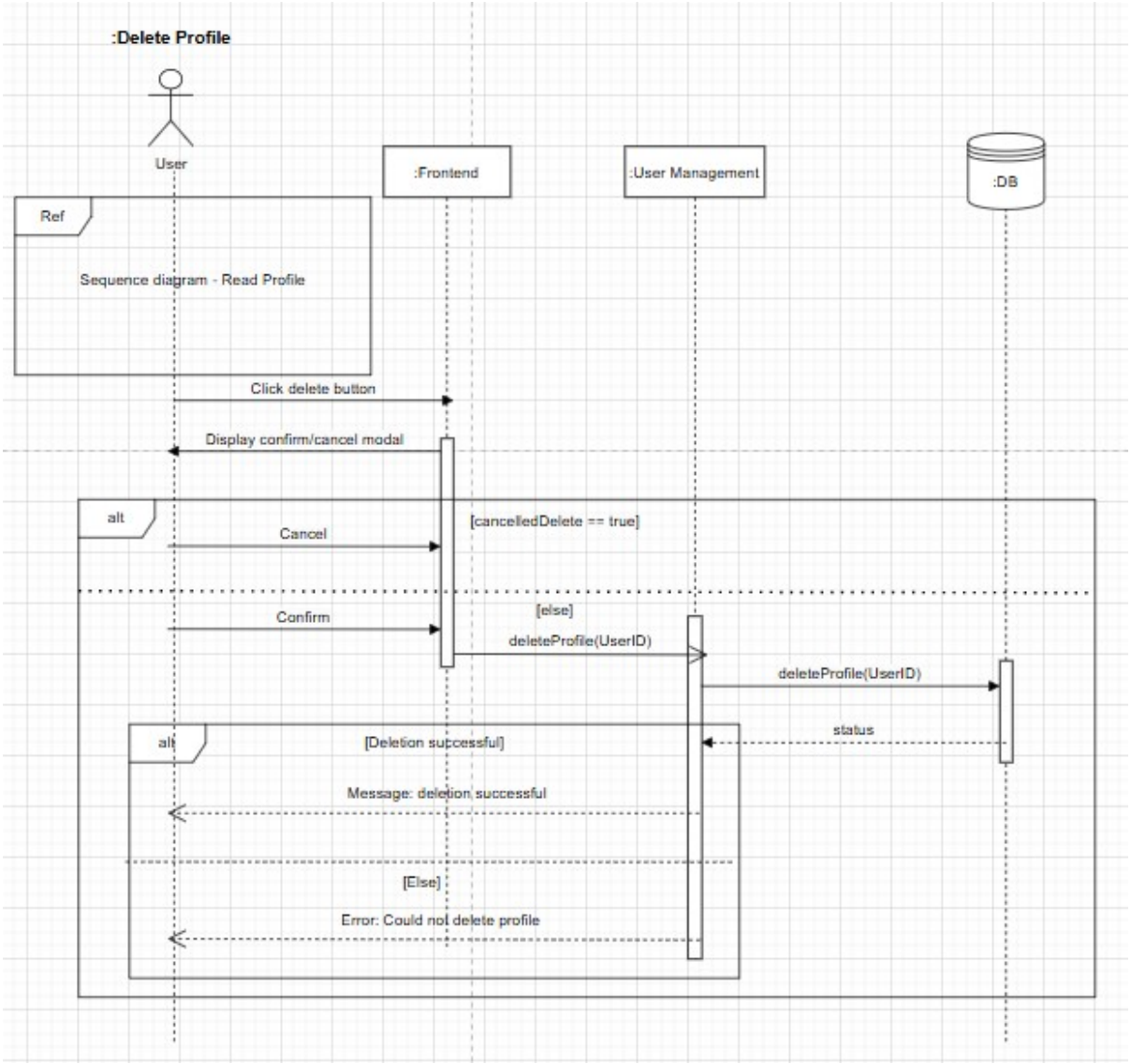
- **Create Profile**



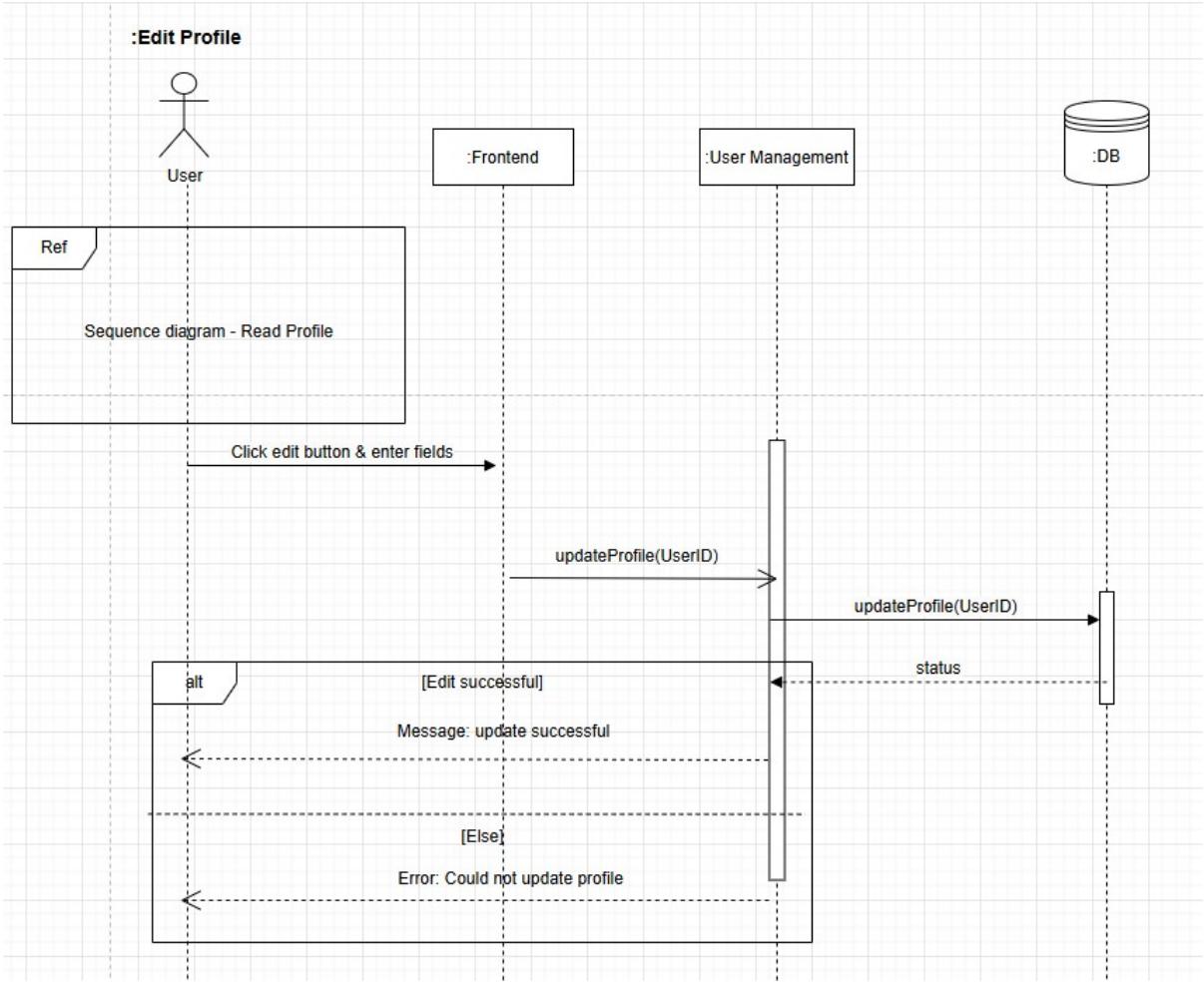
- **Read Profile**



- **Delete Profile**

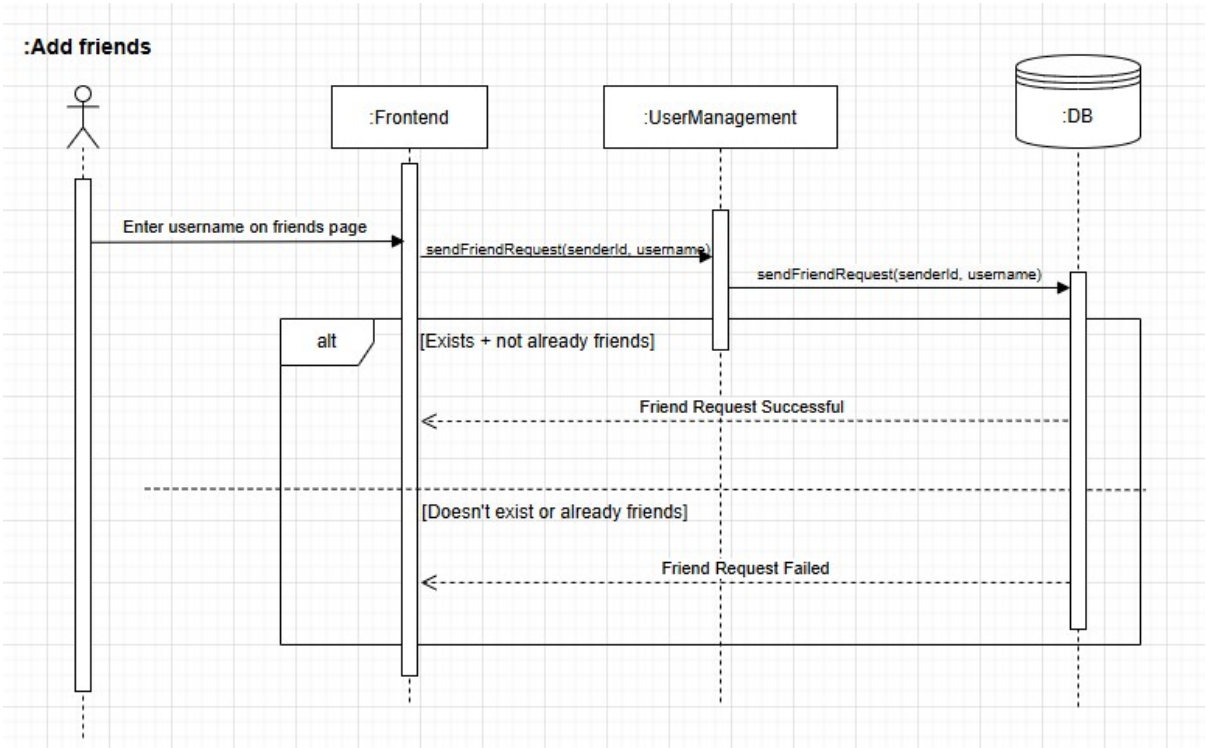


- **Edit Profile**

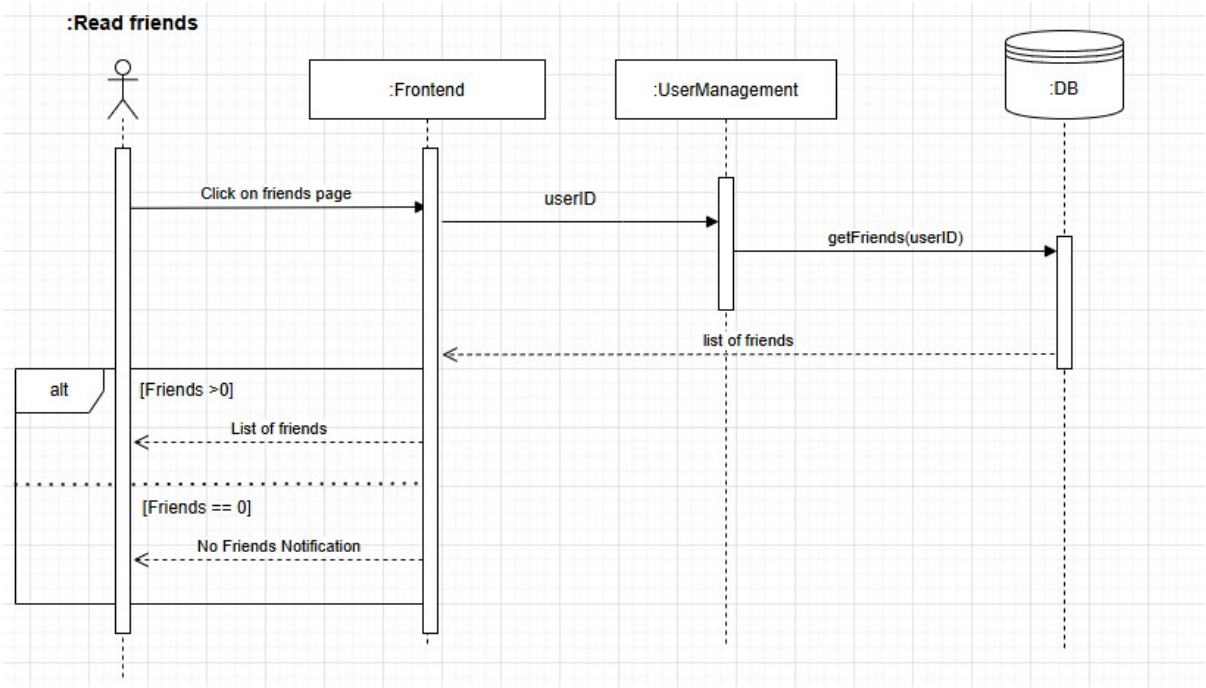


2. **Manage Friends**

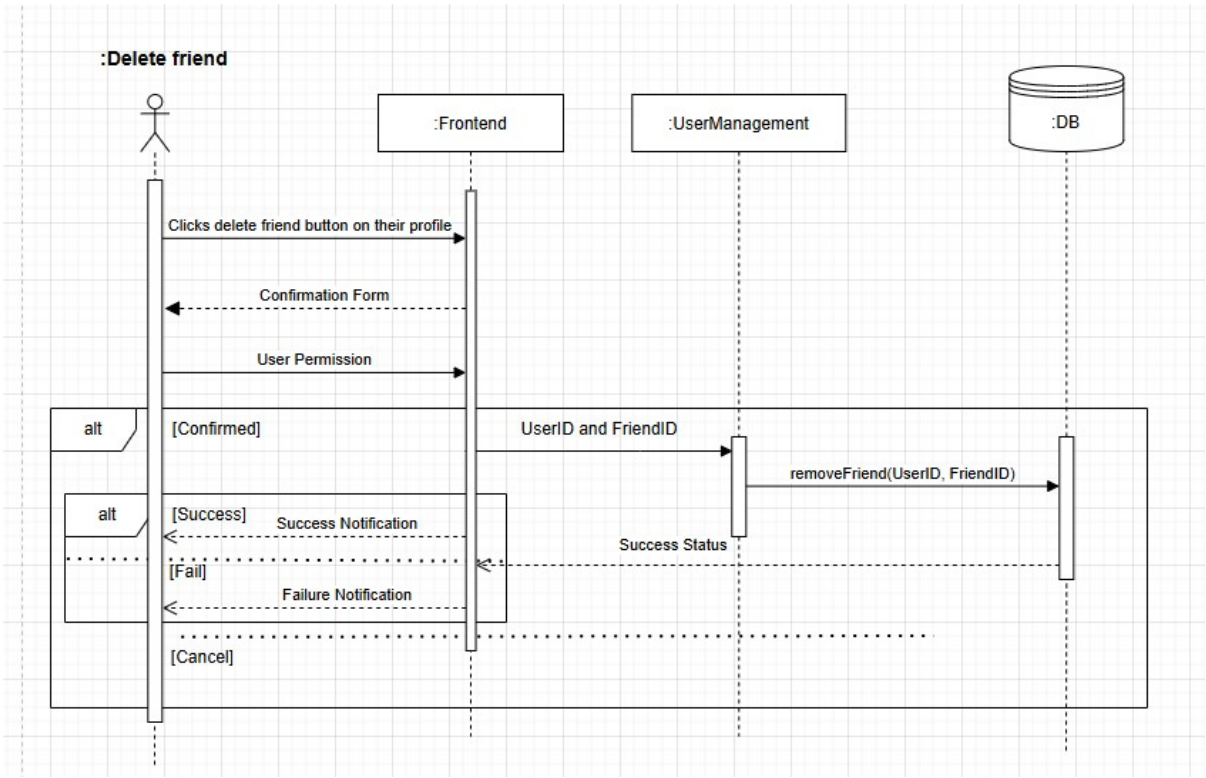
- **Add Friends**



- **Read Friends**

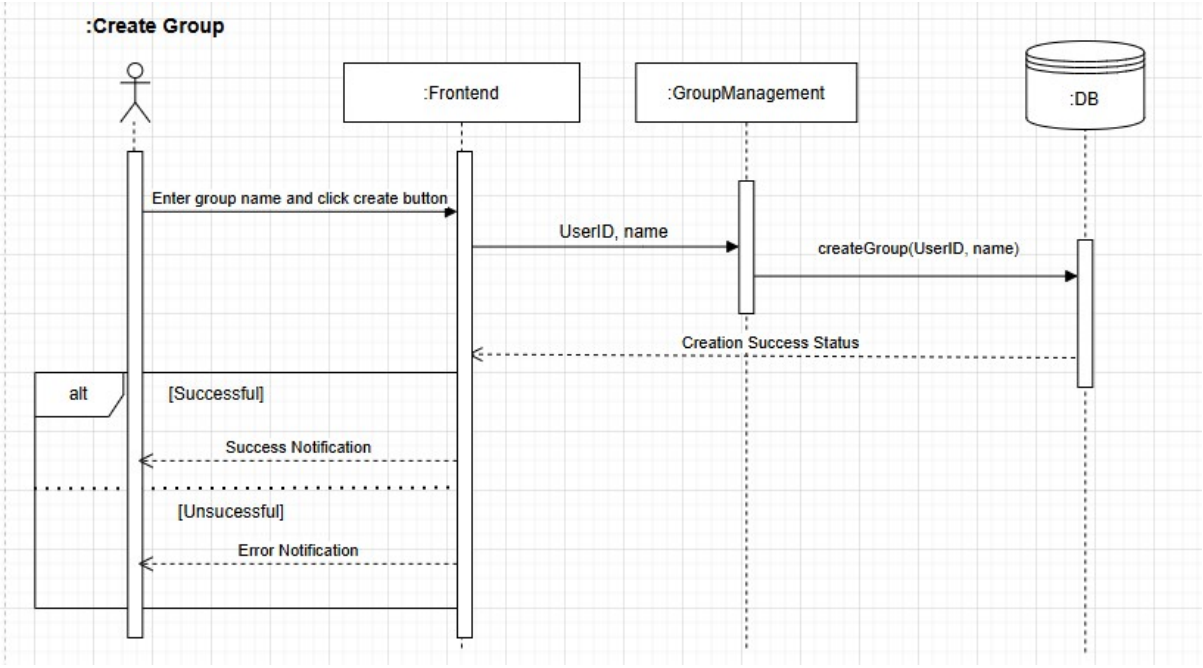


- **Delete Friend**

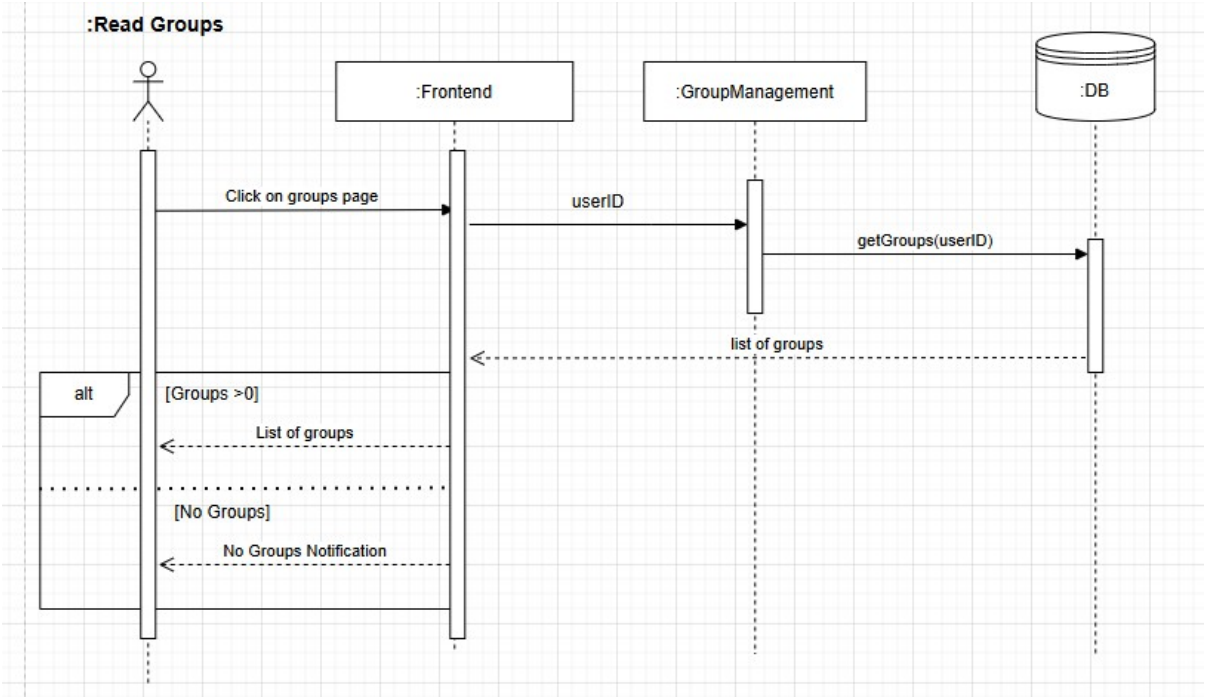


3. **Manage Groups**

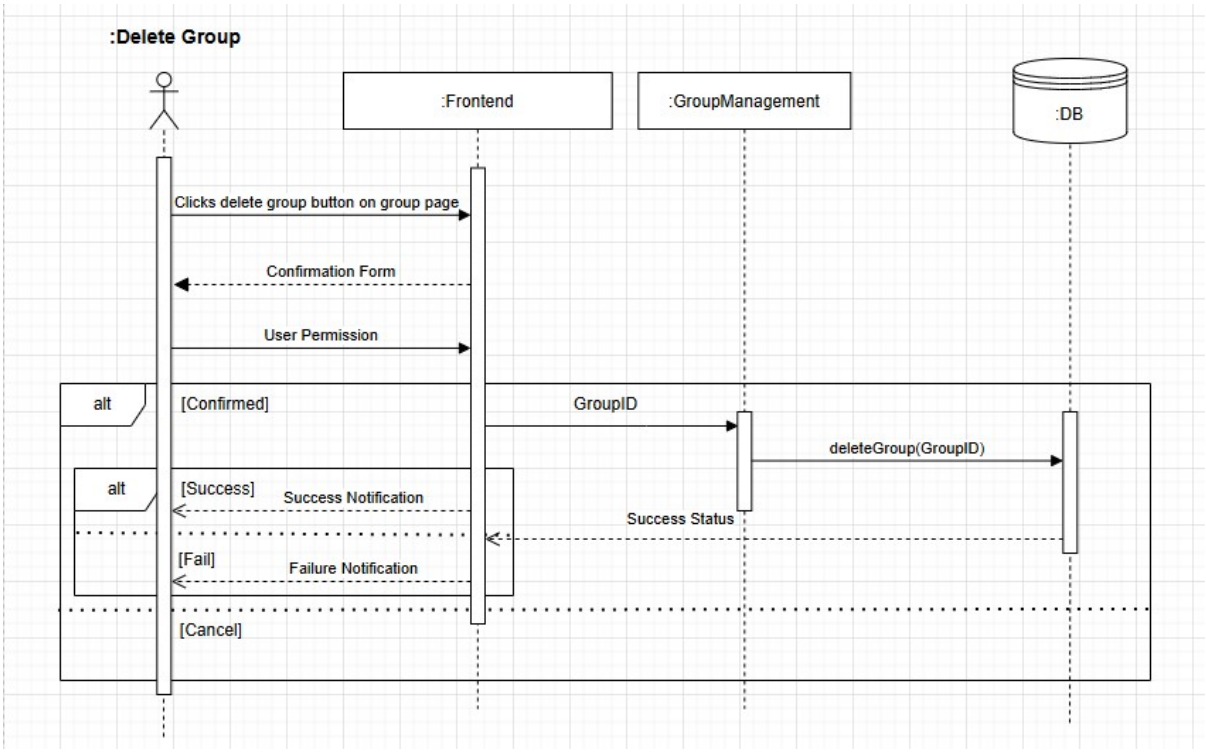
- **Create Group**



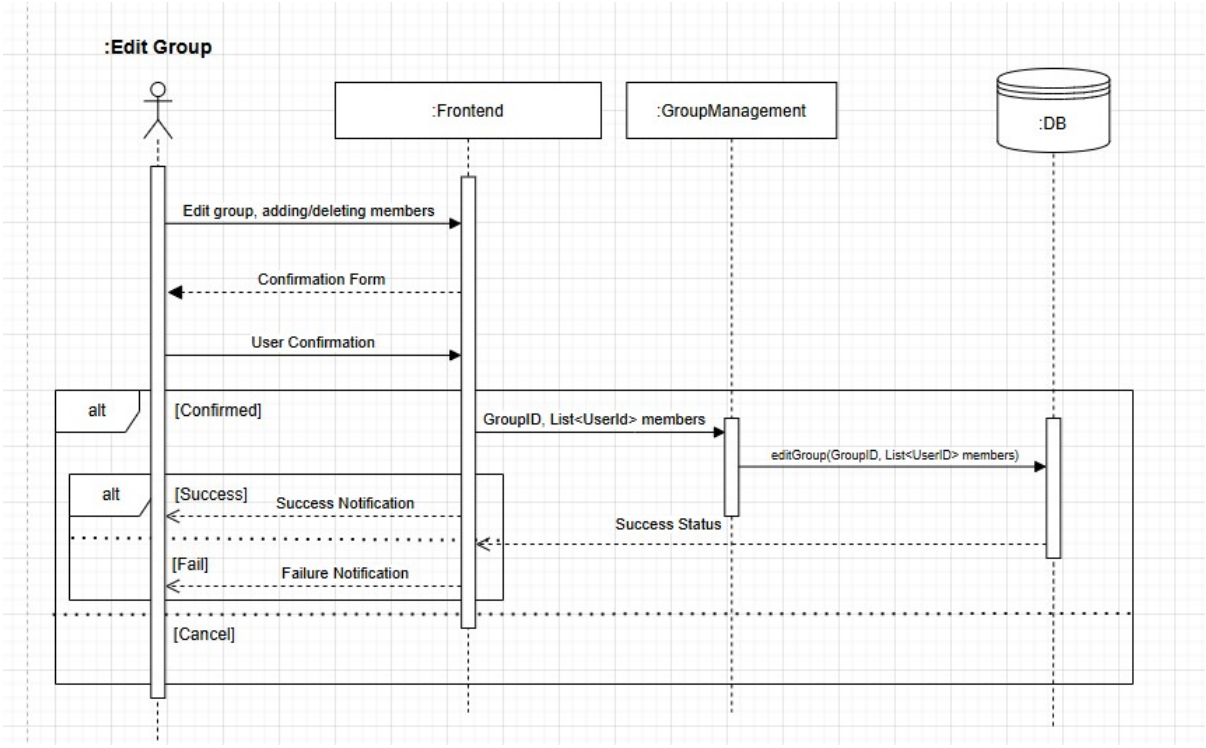
- **Read Groups**



- **Delete Group**

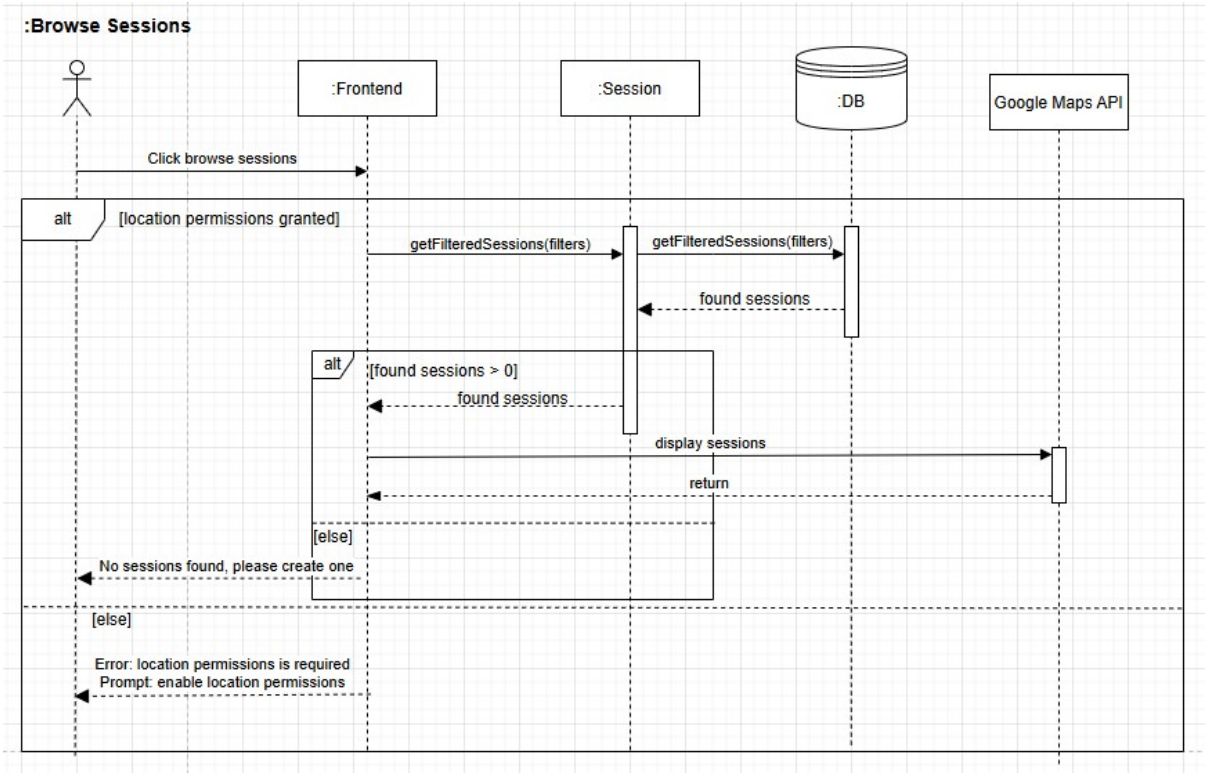


- **Edit Group**



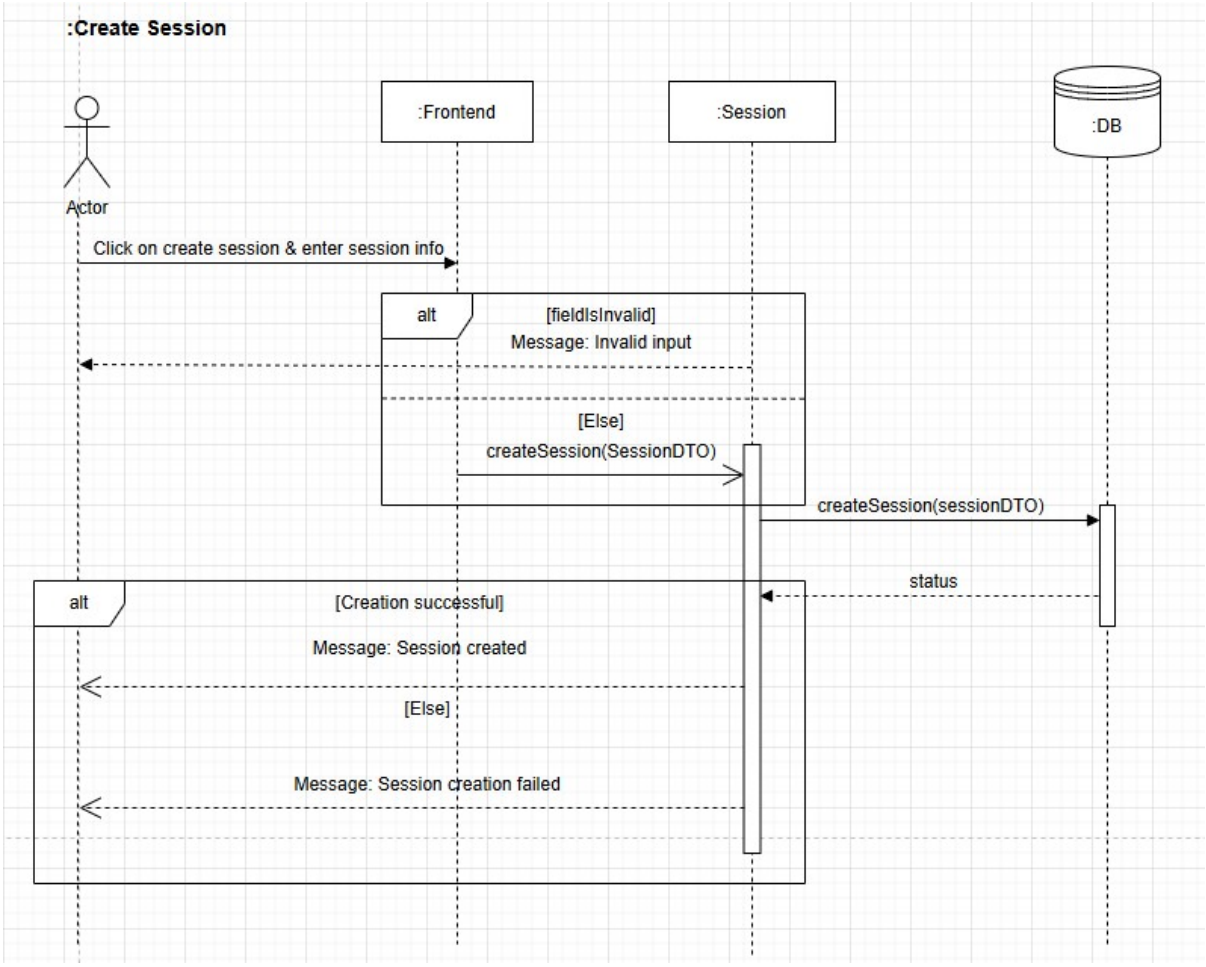
4. **Browse Sessions**

- **Browse Sessions**

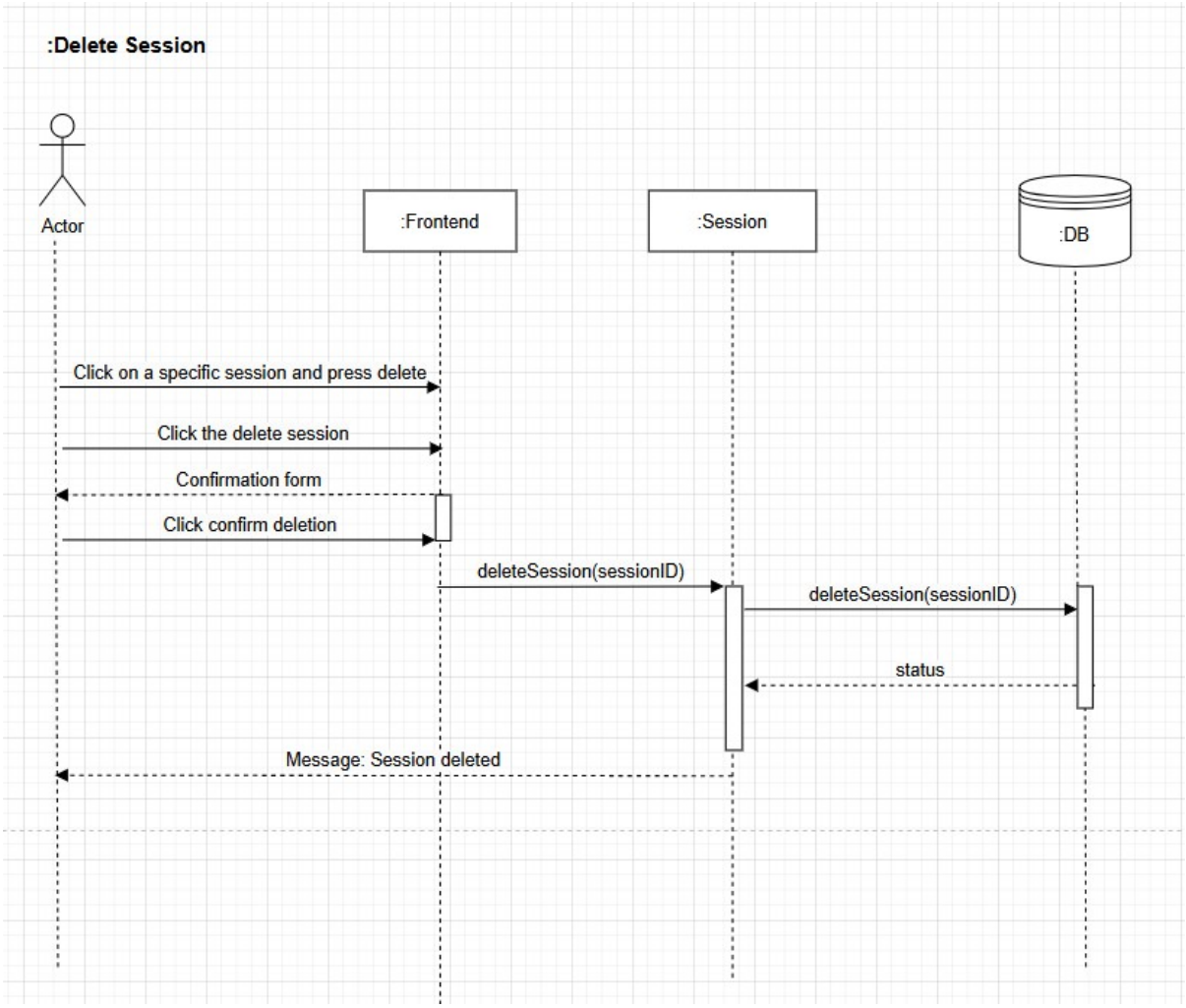


5. **Manage Session**

- **Create Session**

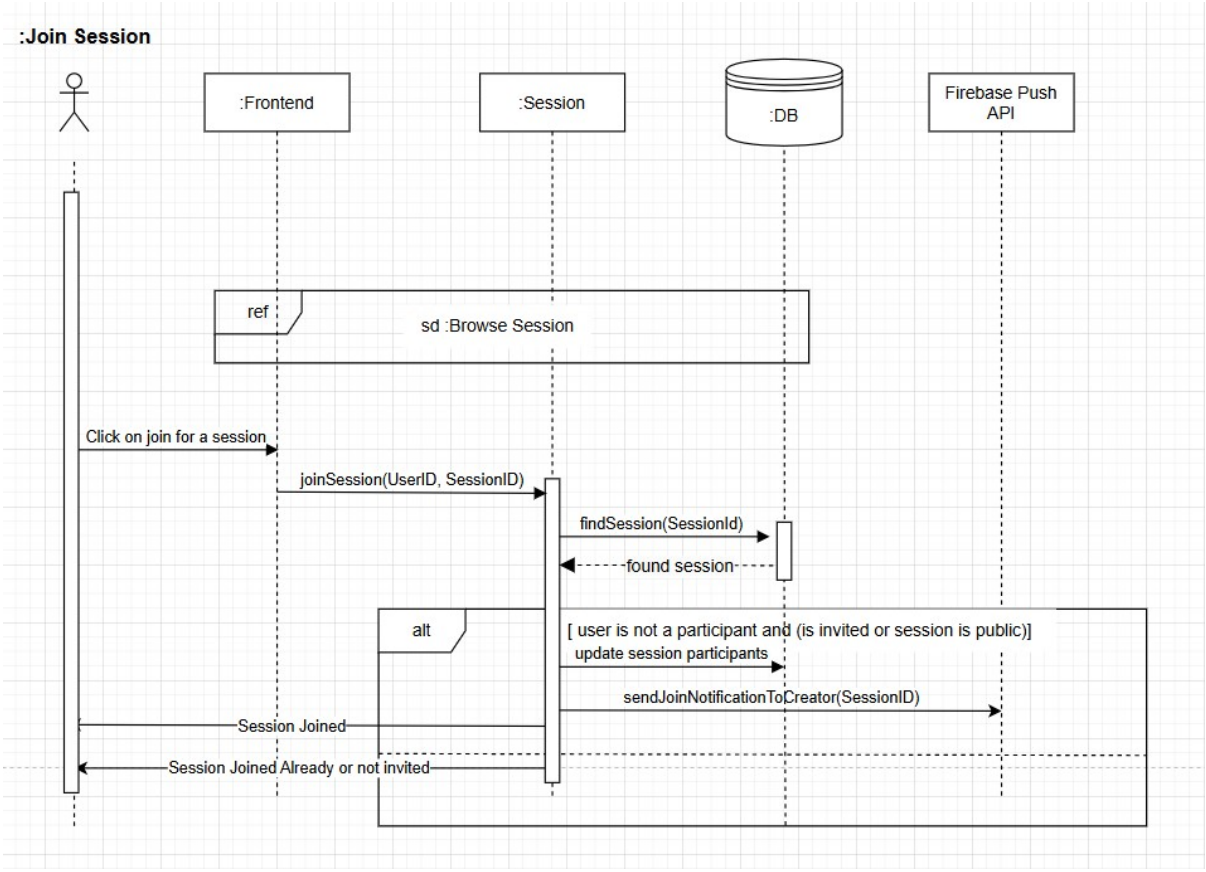


- **Delete Session**

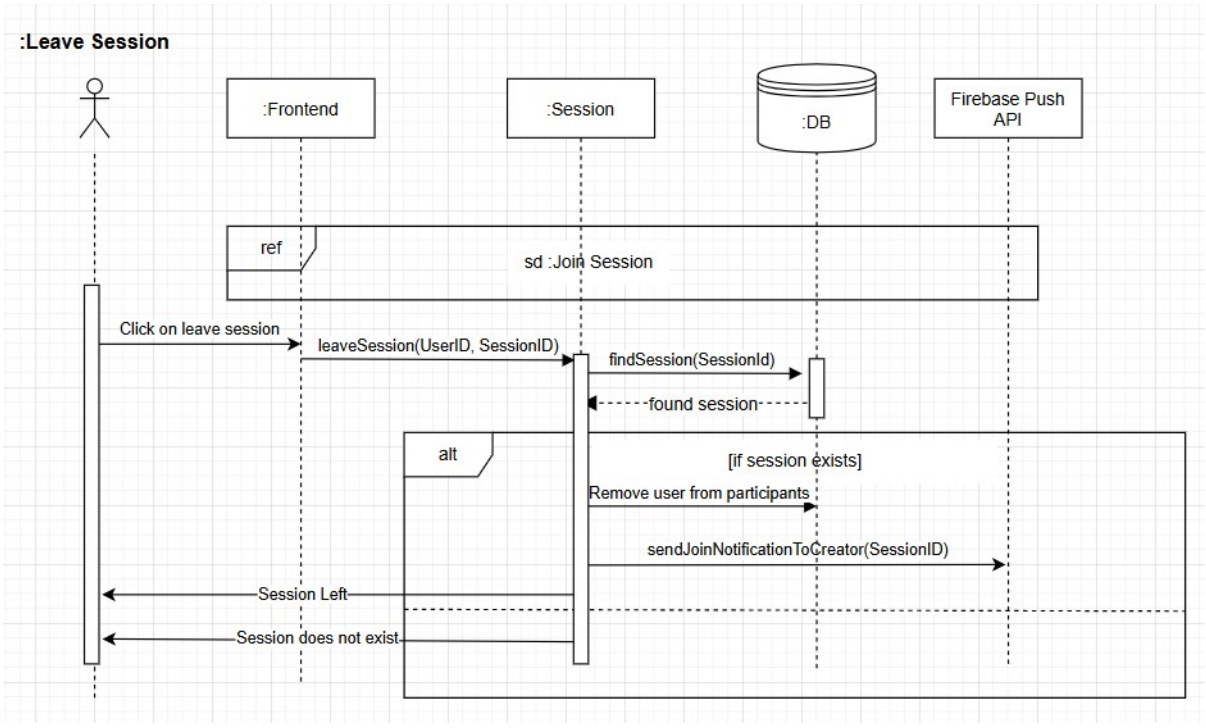


6. **Join/Leave Sessions**

- **Join Session**



- **Leave Session**



4.7. Non-Functional Requirements Design

1. **Real-time Updates**

- **Validation**:
  - Measure time between server update and client reflection
  - Run automated tests to verify notification delivery time < 2 seconds

2. **Location Accuracy**

    - **Validation**:
        - Compare reported location with actual physical location using multiple devices
        - Measure deviation from true coordinates using reference points
        - Verify accuracy remains within 10-meter threshold in 95% of test cases

3. **Performance**

    - **Validation**: - Measure response time for server endpoints using automated tests - Monitor server performance under load testing - Verify response time is < 300ms for 95% of requests

4. **Reliability and Error Recovery**

    - **Validation**: - Simulate bad requests and network failures to test error recovery - Monitor server logs for error messages and exceptions - Verify system can recover gracefully from errors without data loss

4.8. Main Project Complexity Design

Study Session Recommendation Algorithm

- **Description**: Algorithm to recommend compatible study sessions to students using their profile information

- **Why complex?**: Considers multiple factors including the user's year, faculty, session time, the user's friends and the user's interests. A user can enter any response for their interests so these responses need to be normalized and compared for similarities. Thus we first tokenize the interest strings into pieces of semantic meaning, then create embeddings from them (convert into vectors) using google's universal-sentence-encoder model, and compute their cosine similarities using the following formula:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

- **Design**:

    - **Input**: User's profile information and a list of available public study sessions
    - **Output**: Ranked list of recommended study sessions
    - **Main computational logic**: Scoring system based 5 criteria:
        1. If the user's program matches that of the session's host
        2. If the user's year matches that of the sessions "Year level to invite"
        3. If user's friends list contains any of the participants of the session
        4. If the hosted session starts soon (less than 24 hours)
        5. Interest score calculated using Cosine Similarity (**See Note 2**)
    - Add individual criteria scores up and divide by 5 to compute total score for each session in the list. Recommend the top 3 sessions

- **Code**:
  - **Cosine Similarity**

```
  export const sentenceSimilarity = async (
    sentence1: string,
    sentence2: string
  ): Promise<number> => {
    const model = await loadModel();
    const embeddings = await model.embed([sentence1, sentence2]);

    return tf.tidy(() => {
      const vecs = embeddings.arraySync() as number[][];
      const [vec1, vec2] = vecs;

      const dotProduct = vec1.reduce((sum, value, i) => sum +
value * vec2[i], 0);
      const magnitude1 = Math.sqrt(
        vec1.reduce((sum, value) => sum + value * value, 0)
      );
      const magnitude2 = Math.sqrt(
        vec2.reduce((sum, value) => sum + value * value, 0)
      );

      return dotProduct / (magnitude1 * magnitude2);
    });
  };
```

Note 2: Cosine similarity did not work on our T2 Micro EC2 instance as the model uses too much memory on the server despite working when hosting locally. Thus, instead, we use Jaccard similarity when deploying as seen in the below code:

  - **Jaccard Similarity**

```
  export const jaccardSimilarity = (str1: string, str2: string):
number => {
    const stopwords = new Set(["and", "or"]);

    const tokenize = (text: string): Set<string> => {
        return new Set(
            text
                .replace(/,/g, " ")
                .split(/\s+/)
                .filter(word => word.toLowerCase() &&
!stopwords.has(word.toLowerCase()))
                .map(word => word.toLowerCase())
        );
    };

    const set1 = tokenize(str1);
    const set2 = tokenize(str2);
```

```
    if (set1.size === 0 || set2.size === 0) return 0.0;

    const intersectionSize = [...set1].filter(word =>
set2.has(word)).length;
    const unionSize = new Set([...set1, ...set2]).size;

    return intersectionSize / unionSize;
  };
```

The Jaccard Similarity function above calculates the similarity between 2 string sets by dividing the size of their intersection by their union. We create sets of strings by using regex to separate out individual words from the user's interests using spaces and commas, taking away any stop words such as "or" and "and". Jaccard similarity is less powerful as it does not take into account semantically similar words (ex. milk and dairy) but our algoritm still accounts for some edge cases such as capital letters and out of order lists. Jaccard similarity is less resource intensive to compute and will run on our instance.

- **Ranking Algorithm** (Utilizing similarity to score interests)

```
export const scoreSessions = async (user: IUser, sessionsArray:
ISession[]) => {
  const scoredSessions: { session: ISession; score: number }[] =
[];

  for (const session of sessionsArray) {
    const host = await User.findById(session.hostId);
    if (host) {
      const facultyScore = session.faculty == user.faculty ? 1 :
0;
      const participantsScore =
session.participants.some((participant) =>
        user.friends.includes(participant)
      )
        ? 1
        : 0;
      const yearScore = session.year === user.year ? 1 : 0;

      const sessionStartDateMillis = new Date(
        session.dateRange.startDate
      ).getTime();

      const dateScore =
        sessionStartDateMillis - Date.now() <= 24 * 60 * 60 * 1000
&&
        sessionStartDateMillis > Date.now()
          ? 1
          : 0;

      let interestsScore;
      if (user.interests && host.interests) {
        // determine the cosine similarity between the user's
```

```
interests and the host's interests
        // interestsScore = await sentenceSimilarity(
        //     user.interests,
        //     host.interests
        // );

        // use Jaccard similarity in place of cosine similarity
        interestsScore = jaccardSimilarity(user.interests,
host.interests);
      } else {
        interestsScore = 0;
      }

      const finalScore =
        (facultyScore +
          participantsScore +
          yearScore +
          dateScore +
          interestsScore) /
        5;
      scoredSessions.push({ session, score: finalScore });
    }
  }
```

5. Contributions

- Yibo Chen - All members discussed and worked on all parts of the assignment, contributing equally to all the parts.
- David Deng - All members discussed and worked on all parts of the assignment, contributing equally to all the parts.
- Simran Garcha - All members discussed and worked on all parts of the assignment, contributing equally to all the parts.
- Mayank Rastogi - All members discussed and worked on all parts of the assignment, contributing equally to all the parts.