M5 - Requirements and Design

1. Change History

Change History

March 1st 2025 Changes

3.3.1 Sign In

- **Change**: Users must register with our app after successfully signing in with Discord. If they haven't registered yet, they will be taken to the account creation screen, where they must enter their registration preferences, including Spoken Language, Region, Timezone, Skill Level, and Game.
- **Modified Sections**: Functional Requirements: Sign In.
- **Rationale**: Added the additional requirement for users to register with our app. The updated preferences better represent what a user should provide.

4.1 Admin Routes

- **Change**: Added interfaces for retrieving, creating, updating, and deleting admins.
- **Modified Sections**: Admin Routes section.
- **Rationale**: Provides a detailed breakdown of admin-related operations, ensuring clarity and completeness in managing admin users.

4.1 Auth Routes

- **Change**: Added interfaces for login, callback, registration, logout, and redirect.
- **Modified Sections**: Auth Routes section.
- **Rationale**: Ensures secure user authentication and authorization using Discord's OAuth2, providing a clear flow for user login and registration.

4.1 Game Routes

- **Change**: Added interfaces for retrieving, creating, updating, and deleting games.
- **Modified Sections**: Game Routes section.
- **Rationale**: Provides a centralized way to manage games, ensuring consistency and ease of access.

4.1 Group Routes

- **Change**: Added interfaces for retrieving, creating, updating, and deleting groups, as well as joining and leaving groups.
- **Modified Sections**: Group Routes section.
- **Rationale**: Provides a centralized way to manage groups, ensuring consistency and ease of access.

4.1 Matchmaking Routes

- **Change**: Added interfaces for initiating matchmaking and checking matchmaking status.

- **Modified Sections**: Matchmaking Routes section.
- **Rationale**: Provides a centralized way to handle matchmaking requests, ensuring consistency and ease of access.

4.1 Preferences Routes

- **Change**: Added interfaces for retrieving, creating, updating, and deleting preferences.
- **Modified Sections**: Preferences Routes section.
- **Rationale**: Provides a centralized way to manage user preferences, ensuring consistency and ease of access.

4.1 Report Routes

- **Change**: Added interfaces for retrieving, creating, resolving, and deleting reports.
- **Modified Sections**: Report Routes section.
- **Rationale**: Provides a centralized way to handle user reports, ensuring consistency and ease of access.

4.1 User Routes

- **Change**: Added interfaces for retrieving, creating, updating, and deleting users, as well as managing user groups and banning users.
- **Modified Sections**: User Routes section.
- **Rationale**: Provides a centralized way to manage users, ensuring consistency and ease of access.

4.1 Authentication

- **Change**: Documentation format change
- **Modified Sections**: Authentication Routes section.
- **Rationale**: The original purpose and interfaces for authentication remain relevant and accurate. We lost marks for formatting however.

4.1 Sign In

- **Change**: Documentation format change
- **Modified Sections**: Sign In section.
- **Rationale**: The original purpose and interfaces for sign-in remain relevant and accurate. We lost marks for formatting however.

4.1 User Management

- **Change**: Documentation format change
- **Modified Sections**: User Management section.
- **Rationale**: The original purpose and interfaces for user management remain relevant and accurate. We lost marks for formatting however.

4.1 Matchmaking

- **Change**: Documentation format change

- **Modified Sections**: Matchmaking section.
- **Rationale**: The original purpose and interfaces for matchmaking remain relevant and accurate. We lost marks for formatting however.

4.1 Group Management

- **Change**: Documentation format change
- **Modified Sections**: Group Management section.
- **Rationale**: The original purpose and interfaces for group management remain relevant and accurate. We lost marks for formatting however.

4.1 Report Management

- **Change**: Documentation format change
- **Modified Sections**: Report Management section.
- **Rationale**: The original purpose and interfaces for report management remain relevant and accurate. We lost marks for formatting however.

4.1 Admin Management

- **Change**: Documentation format change
- **Modified Sections**: Admin Management section.
- **Rationale**: The original purpose and interfaces for admin management remain relevant and accurate. We lost marks for formatting however.

4.2 Databases

- **Change**: Updated the purpose and details of the GameOnDB (MySQL) and Redis databases.
- **Modified Sections**: Databases section.
- **Rationale**: Expanded the description to include more specific details about the tables and the use of Redis for session management, caching, and matchmaking queues.

4.3 External Modules

- **Change**: Added a new section for External Modules.
- **Modified Sections**: External Modules section.
- **Rationale**: Provides clarity on the use of the Discord API for authentication and creating matchmaking groups, ensuring a comprehensive understanding of external dependencies.

4.6 Functional Requirements Sequence Diagram

- **Change**: Updated UML diagram for Signin.
- **Modified Sections**: Functional Requirements Sequence Diagram.
- **Rationale**: Design was updated.

March 1st 2025 Changes

3.1 Use Case Diagram

- **Change**: Updated use case diagram.
- **Modified Sections**: Use case diagram.
- **Rationale**: Updated based on M3 TA feedback.

4.4 Frameworks

- **Change**: Remove android studio.
- **Modified Sections**: Frameworks.
- **Rationale**: Updated based on M3 TA feedback.

5. Contributions

- **Change**: Add contributions.
- **Modified Sections**: Contributions
- **Rationale**: Updated based on M3 TA feedback.

3.3 Functional Requirements

- **Change**: Fix use case scenarios for all functional requirements.
- **Modified Sections**: Functional Requirements
- **Rationale**: Updated based on M3 TA feedback.

3.3 Non Functional Requirements

- **Change**: Fix non functional requirements.
- **Modified Sections**: Non Functional Requirements
- **Rationale**: Updated based on M3 TA feedback.

March 12th 2025 Changes

3.5 Non Functional Requirements

- **Change**: Change non functional requirements.
- **Modified Sections**: Non Functional Requirements
- **Rationale**: Based on the feedback received from Milestone 3, we revised and updated our non-functional requirements. The previous requirements were untestable and lacked measurable criteria. We have now replaced them with two new non-functional requirements that are clear, measurable, and testable.

4.7 Non Functional Requirements Design

- **Change**: Updated validation.
- **Modified Sections**: Non Functional Requirements Design
- **Rationale**: As mentioned above we have changed our non-functional requirements. Hence, we have also changed the validation required.

March 14th 2025 Changes

3.3 Functional Requirements - Find Group

- **Change**: Update use case.
- **Modified Sections**: Functional Requirements
- **Rationale**: Updated Find Group use case based on actual implementation.

3.3 Functional Requirements - View Existing Group

- **Change**: Update use case.
- **Modified Sections**: Functional Requirements
- **Rationale**: Updated View Existing Group use case based on actual implementation.

3.3 Functional Requirements - Report User

- **Change**: Update use case.
- **Modified Sections**: Functional Requirements
- **Rationale**: Updated Submit Report use case based on actual implementation.

March 27 2025 Changes

4.5 Dependencies Diagram

- **Change**: Change dependendencies diagram to more accurately reflect the implementation and KISS principle.
- **Modified Sections**: Dependency diagram
- **Rationale**: We previously lost marks for violating the KISS principle.

3.1 Use Case Diagram

- **Change**:Added the add-game use case for admin
- **Modified Sections**: Use case diagram
- **Rationale**: Reflect added functionality for admin user

3.1 Use Case Diagram

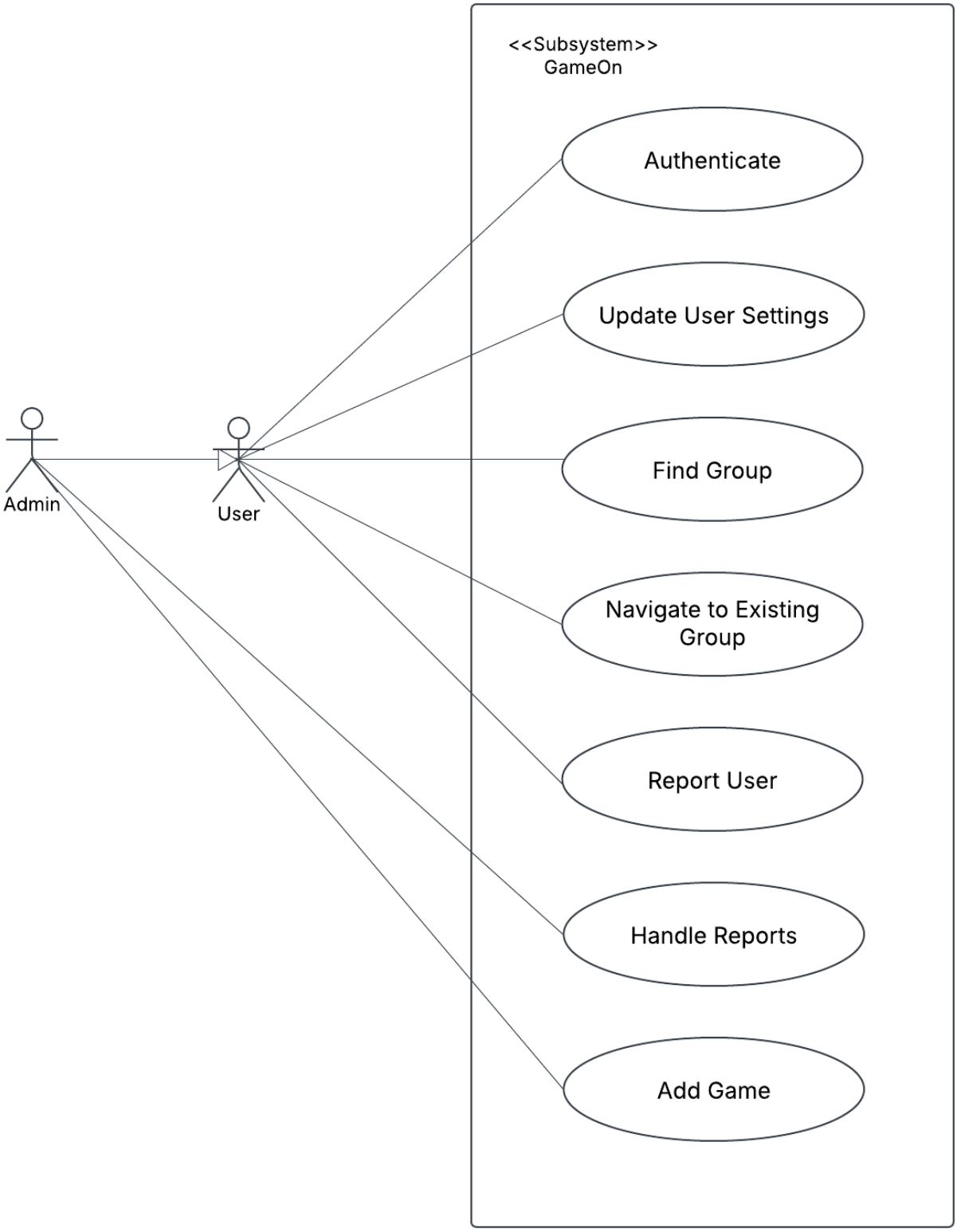
- **Change**: Changed every use case diagram to more accurately reflect the application functionality
- **Modified Sections**: Use case diagram
- **Rationale**: Reflect correct functionality

2. Project Description

GameOn is a social matchmaking platform designed for gamers to find ideal teammates and build lasting connections. By authenticating through Discord, players create personalized profiles, sharing details like preferred games, skill levels, communication styles, and playstyles. The app intelligently matches players based on their preferences, instantly creating a dedicated Discord group for seamless in-game coordination and ongoing communication. With integrated feedback systems, including reviews and ratings, GameOn fosters a supportive and positive gaming community.

3. Requirements Specification

3.1. Use-Case Diagram



3.2. Actors Description

1. **User**: A player who uses Discord authentication to access the app. They can set preferences, join groups via group matching, interact with other users, and report users if needed.
2. **Admin**: A system administrator who monitors user reports and has the authority to ban users if necessary, ensuring a safe and fair community environment. They can also add additional games for users to play.

3.3. Functional Requirements

1. **Authenticate**

   - **Overview**:

     1. Login
     2. Register

   - **Detailed Flow for Each Independent Scenario**:

     1. **Login**:
        - **Description**: Users are prompted to sign in with Discord. If they have already signed in with Discord and registered with our app they are taken to the home screen.
        - **Primary actor(s)**: User
        - **Main success scenario**:
          1. User opens the app.
          2. System displays the login screen with a "Sign in with Discord" button.
          3. User clicks "Sign in with Discord."
          4. System redirects the user to Discord’s authentication page.
          5. User enters Discord credentials and submits the form.
          6. System validates the credentials with Discord API.
          7. System checks if the user has a registered profile in the app.
          8. If registered, system redirects the user to the home screen.
          9. User successfully accesses the app.
        - **Failure scenario(s)**:
          - 1a. Unsuccessful sign-in via Discord authentication.
            - 1a1. User enters incorrect Discord credentials.
            - 1a2. System displays an "Invalid credentials" error message.
            - 1a3. User is prompted to re-enter credentials.
     2. **Register**:
        - **Description**: Users are prompted to sign in with Discord. If they haven't registered with our app, they are taken to the account creation screen, where they must enter their registration preferences, including Spoken Language, Region, Timezone, Skill Level, and Game.
        - **Primary actor(s)**: User
        - **Main success scenario(s)**:
          1. User clicks "Sign in with Discord."
          2. System redirects to Discord authentication.
          3. User enters credentials, and system verifies them.
          4. If new user, system prompts for preferences.
          5. User fills in and submits the form.
          6. System saves profile and redirects to the home screen.
        - **Failure scenario(s)**:
          - 1a. Unsuccessful sign-in via Discord authentication.
            - 1a1. Incorrect credentials are provided.
          - 2a. User fails to provide all required fields.
            - 2a1. User must be prompted to fill in missing fields.
          - 2b: User leaves and closes application before all imformation is filled. Registration does not occur.

2. **Update User Settings**

    - **Overview**:

        1. View and Change Settings

    - **Detailed Flow for Each Independent Scenario**:

        1. **View and Change Settings**:
            - **Description**: When clicking their avatar in the top right of the screen, users are taken to their account settings page, where they can change their spoken language preferences, and time zone.
            - **Primary actor(s)**: User
            - **Main success scenario**:
                1. User clicks their avatar to open settings.
                2. System displays the user's current settings.
                3. User modifies settings (e.g., avatar, language, timezone).
                4. User presses "Confirm," and system updates the database.
                5. System confirms the update and redirects the user to the previous screen.
            - **Failure scenario(s)**:
                - 1a. Database rejects update, actor remains on this screen and receives an error

3. **Find Group**

    - **Overview**:

        1. Looking for Group

    - **Detailed Flow for Each Independent Scenario**:

        1. **Looking for Group**:

            - **Description**: When a user clicks the "Find Group" button on the main page of the app, they are entered into the matchmaking queue. The Find Group button is then disabled and the user can continue to use other features in the app. When a group is found, or the timeout of 1 minute is reached, the user receives a live update in the form of a popup notification, informing them whether matchmaking was successful or not.

            - **Primary actor(s)**: User

            - **Main success scenario**:

                1. User is on the main page of GameOn.
                2. User clicks the "Find Group" button.
                3. The "Find Group" button becomes disabled and says "Finding".
                4. A matchmaking request is submitted, and the user is added to the matchmaking queue.
                5. A group is found based on the user's preferences.
                6. A live update appears stating "You have been matched with a group!"

7. The new group appears in the "My Groups" section of the main page.

- **Failure scenario(s)**:
  - 1a. Not enough users in the matchmaking queue, After a timeout, the system removes the user from matchmaking queue and displays a timeout popup.

4. **Navigate to Existing Group**

- **Overview**:
  1. Navigate to Existing Group Page
- **Detailed Flow for Each Independent Scenario**:
  1. Navigate to Existing Group Page:
     - **Description**: When a user clicks on an existing group from the “My Groups” banner, they are taken to the existing group page.
     - **Primary actor(s)**: User
     - **Main success scenario**:
       1. User is on the main page of GameOn.
       2. User clicks on an existing group from the "My Groups" section.
       3. The View Existing Groups page opens, displaying the group name, group members, and a "Go to Discord Group" button.
       4. User clicks the "Go to Discord Group."
       5. Chrome opens the Discord web version and displays the newly created group with the added members.
     - **Failure scenario(s)**:
       - 1a. The group is not available from the database, the user receives a 500 server error and stays on the main screen.

5. **Report User**

- **Overview**:
  1. Submit Report
- **Detailed Flow for Each Independent Scenario**:
  1. **Submit Report**:
     - **Description**: When a user presses "Submit a Report" on the main page, a new page appears where a user can write a summary on why the user is being reported. Upon clicking “Submit Report”, the report is submitted to the administrators.
     - **Primary actor(s)**: User, Admin
     - **Main success scenario**:
       1. User selects the "Submit a Report" button on the main page.
       2. User selects a group from the dropdown
       3. User selects a user to report from the dropdown
       4. User writes a reason for the report
       5. User selects "Submit report"
       6. System successfully sends the report to the database

- **Failure scenario(s)**:
  - 6a. User presses "Submit" and an error pops up. (See **3.5.2**)
    - 6a1 The "Reason" textbox turns red and an error message pops up warning the user that the input must be less than 500 characters.
    - 6a2. The user can shorten their input and attempt to submit again.

6. **Handle Reports**

- **Overview**:

  1. Ban User
  2. Acquit User

- **Detailed Flow for Each Independent Scenario**:

  1. **Ban User**:

     - **Description**: Admins view and action user reports from the view reports screen. Admin has the option to ban a user based on the summary they've been provided.
     - **Primary actor(s)**: Admin
     - **Main success scenario**:
       1. Admin clicks "View reports"
       2. Admin selects the desired report from the list
       3. Admin selects "Ban user"
       4. Once the user logs out, they will hit the banned user screen when attempting to log back in.
     - **Failure scenario(s)**:
       - 1a. System error prevents banning the user. Admin selects "Ban User," but a database or server error occurs, preventing the ban.

  2. **Acquit User**:

     - **Description**: Admins view and action user reports from the view reports screen. Admin has the option to *not* ban the user based on the summary they've been provided.
     - **Primary actor(s)**: Admin
     - **Main success scenario**:
       1. Admin clicks "View reports"
       2. Admin selects the desired report from the list
       3. Admin selects "Acquit User"
       4. User can continue using the app normally.
     - **Failure scenario(s)**:
       - 1a. System error prevents acquitting the user. Admin selects "Acquit User," but a database or server error occurs, preventing the action.

7. **Add Game**

- **Overview**:

  1. Add New Game to System

- **Detailed Flow for Each Independent Scenario**:

    1. **Add New Game**:
        - **Description**: Admins can add new games to the system from the admin panel. These games will be available for users to select in their preferences.
        - **Primary actor(s)**: Admin
        - **Main success scenario**:
            1. Admin clicks their avatar to access the admin panel.
            2. Admin selects "Manage Games" from the admin panel.
            3. Admin clicks "Add New Game" button.
            4. Admin enters the game name and description.
            5. Admin clicks "Submit" to add the game.
            6. System validates and stores the new game in the database.
            7. System displays a success message and the game appears in the games list.
        - **Failure scenario(s)**:
            - 5a. Admin enters a game name that already exists.
                - 5a1. System displays an error message indicating the game already exists.
                - 5a2. Admin can modify the game name and try again.
            - 6a. Database error prevents adding the game.
                - 6a1. System displays an error message about the database issue.
                - 6a2. Admin can try again later.

3.4. Screen Mockups

3.5. Non-Functional Requirements

1. **Session-Based Endpoint Security**
    - **Description**: All critical API endpoints must be secured through session validation. Only users authenticated via Discord OAuth should have access to these endpoints.
    - **Justification**: Ensures platform security by restricting access to authenticated users. This prevents unauthorized access and protects user data.

2. **Report Reason Character Limit**
    - **Description**: User-submitted report reasons are limited to 500 characters.
    - **Justification**: Prevents database bloat and mitigates potential security risks such as memory exhaustion or abuse through overly large inputs. This ensures system stability and efficient storage usage.

4. Designs Specification

4.1. Main Components

Admin Routes

- **Purpose**: Manages admin-related operations such as retrieving, creating, updating, and deleting admins.

- **Rationale**: Provides a centralized way to manage admin users, ensuring only authorized users have access to admin functionalities.
- **Interfaces**:
    1. **GET /admins**
        - **Parameters**: None
        - **Return Value**: JSON array of admin objects
        - **Description**: Fetches all admin users from the database.
    2. **GET /admins/:id**
        - **Parameters**: admin ID (URL parameter)
        - **Return Value**: JSON object of the admin
        - **Description**: Fetches a specific admin user based on the provided ID.
    3. **POST /admins**
        - **Parameters**: JSON object with discord_id and permissions
        - **Return Value**: JSON object of the created admin
        - **Description**: Adds a new admin user to the database.
    4. **PUT /admins/:id**
        - **Parameters**: admin ID (URL parameter), JSON object with updated permissions
        - **Return Value**: JSON object of the updated admin
        - **Description**: Updates the permissions of an existing admin user.
    5. **DELETE /admins/:id**
        - **Parameters**: admin ID (URL parameter)
        - **Return Value**: JSON object with a success message
        - **Description**: Removes an admin user from the database.

Auth Routes

- **Purpose**: Handles authentication and authorization processes, including login, registration, and logout.
- **Rationale**: Ensures secure user authentication and authorization using Discord's OAuth2.
- **Interfaces**:
    1. **GET /auth/login**
        - **Parameters**: None
        - **Return Value**: Redirect to Discord's OAuth2 login page
        - **Description**: Starts the OAuth2 login process.
    2. **GET /auth/callback_discord**
        - **Parameters**: None
        - **Return Value**: JSON object with user information
        - **Description**: Processes the OAuth2 callback and retrieves user information.
    3. **POST /auth/register**
        - **Parameters**: JSON object with user details
        - **Return Value**: JSON object of the registered user
        - **Description**: Registers a new user in the system.
    4. **POST /auth/logout**
        - **Parameters**: None
        - **Return Value**: JSON object with a success message
        - **Description**: Logs out the current user and ends their session.
    5. **GET /auth/redirect**
        - **Parameters**: None

        - **Return Value**: Redirect to the frontend with the authorization code
        - **Description**: Redirects the user to the frontend application with the authorization code.

Game Routes

- **Purpose**: Manages game-related operations such as retrieving, creating, updating, and deleting games.
- **Rationale**: Provides a centralized way to manage games, ensuring consistency and ease of access.
- **Interfaces**:
    1. **GET /games**
        - **Parameters**: None
        - **Return Value**: JSON array of game objects
        - **Description**: Fetches all games from the database.
    2. **GET /games/:id**
        - **Parameters**: game ID (URL parameter)
        - **Return Value**: JSON object of the game
        - **Description**: Fetches a specific game based on the provided ID.
    3. **POST /games**
        - **Parameters**: JSON object with game_name and description
        - **Return Value**: JSON object of the created game
        - **Description**: Adds a new game to the database.
    4. **PUT /games/:id**
        - **Parameters**: game ID (URL parameter), JSON object with updated game_name and description
        - **Return Value**: JSON object of the updated game
        - **Description**: Updates the details of an existing game.
    5. **DELETE /games/:id**
        - **Parameters**: game ID (URL parameter)
        - **Return Value**: JSON object with a success message
        - **Description**: Removes a game from the database.

Group Routes

- **Purpose**: Manages group-related operations such as retrieving, creating, updating, and deleting groups, as well as joining and leaving groups.
- **Rationale**: Provides a centralized way to manage groups, ensuring consistency and ease of access.
- **Interfaces**:
    1. **GET /groups**
        - **Parameters**: None
        - **Return Value**: JSON array of group objects
        - **Description**: Fetches all groups from the database.
    2. **GET /groups/:id**
        - **Parameters**: group ID (URL parameter)
        - **Return Value**: JSON object of the group
        - **Description**: Fetches a specific group based on the provided ID.
    3. **POST /groups**
        - **Parameters**: JSON object with game_id, group_name, and max_players
        - **Return Value**: JSON object of the created group

    - **Description**: Adds a new group to the database.
4. **PUT /groups/:id**
    - **Parameters**: group ID (URL parameter), JSON object with updated group_name and max_players
    - **Return Value**: JSON object of the updated group
    - **Description**: Updates the details of an existing group.
5. **DELETE /groups/:id**
    - **Parameters**: group ID (URL parameter)
    - **Return Value**: JSON object with a success message
    - **Description**: Removes a group from the database.
6. **POST /groups/:id/join**
    - **Parameters**: group ID (URL parameter)
    - **Return Value**: JSON object with a success message
    - **Description**: Adds the current user to the specified group.
7. **DELETE /groups/:id/leave**
    - **Parameters**: group ID (URL parameter)
    - **Return Value**: JSON object with a success message
    - **Description**: Removes the current user from the specified group.
8. **GET /groups/:id/members**
    - **Parameters**: group ID (URL parameter)
    - **Return Value**: JSON array of group member objects
    - **Description**: Fetches all members of the specified group.
9. **GET /groups/:id/url**
    - **Parameters**: group ID (URL parameter)
    - **Return Value**: JSON object with the group's URL
    - **Description**: Fetches the URL of the specified group.

Matchmaking Routes

- **Purpose**: Manages matchmaking-related operations such as initiating matchmaking and checking matchmaking status.
- **Rationale**: Provides a centralized way to handle matchmaking requests, ensuring consistency and ease of access.
- **Interfaces**:
    1. **POST /matchmaking/initiate**
        - **Parameters**: JSON object with preference_id
        - **Return Value**: JSON object with a success message
        - **Description**: Starts the matchmaking process based on the provided preferences.
    2. **GET /matchmaking/status/:discord_id**
        - **Parameters**: discord ID (URL parameter)
        - **Return Value**: JSON object with the matchmaking status
        - **Description**: Retrieves the matchmaking status for the specified user.

Preferences Routes

- **Purpose**: Manages preference-related operations such as retrieving, creating, updating, and deleting preferences.

- **Rationale**: Provides a centralized way to manage user preferences, ensuring consistency and ease of access.
- **Interfaces**:
    1. **GET /preferences**
        - **Parameters**: None
        - **Return Value**: JSON array of preference objects
        - **Description**: Fetches all preferences from the database.
    2. **GET /preferences/:id**
        - **Parameters**: preference ID (URL parameter)
        - **Return Value**: JSON object of the preferences
        - **Description**: Fetches specific preferences based on the provided ID.
    3. **POST /preferences**
        - **Parameters**: JSON object with discord_id, spoken_language, time_zone, skill_level, and game_id
        - **Return Value**: JSON object of the created preferences
        - **Description**: Adds new preferences to the database.
    4. **PUT /preferences/:id**
        - **Parameters**: preference ID (URL parameter), JSON object with updated spoken_language, time_zone, and skill_level
        - **Return Value**: JSON object of the updated preferences
        - **Description**: Updates the details of existing preferences.
    5. **DELETE /preferences/:id**
        - **Parameters**: preference ID (URL parameter)
        - **Return Value**: JSON object with a success message
        - **Description**: Removes preferences from the database.

Report Routes

- **Purpose**: Manages report-related operations such as retrieving, creating, resolving, and deleting reports.
- **Rationale**: Provides a centralized way to handle user reports, ensuring consistency and ease of access.
- **Interfaces**:
    1. **GET /reports**
        - **Parameters**: None
        - **Return Value**: JSON array of report objects
        - **Description**: Fetches all reports from the database.
    2. **GET /reports/:id**
        - **Parameters**: report ID (URL parameter)
        - **Return Value**: JSON object of the report
        - **Description**: Fetches a specific report based on the provided ID.
    3. **POST /reports**
        - **Parameters**: JSON object with reported_discord_id, group_id, and reason
        - **Return Value**: JSON object of the created report
        - **Description**: Adds a new report to the database.
    4. **PUT /reports/:id/resolve**
        - **Parameters**: report ID (URL parameter), JSON object with resolved status
        - **Return Value**: JSON object of the resolved report

        - **Description**: Updates the status of a report to resolved.
    5. **DELETE /reports/:id**
        - **Parameters**: report ID (URL parameter)
        - **Return Value**: JSON object with a success message
        - **Description**: Removes a report from the database.

User Routes

- **Purpose**: Manages user-related operations such as retrieving, creating, updating, and deleting users, as well as managing user groups and banning users.
- **Rationale**: Provides a centralized way to manage users, ensuring consistency and ease of access.
- **Interfaces**:
    1. **GET /users**
        - **Parameters**: None
        - **Return Value**: JSON array of user objects
        - **Description**: Fetches all users from the database.
    2. **GET /users/:id**
        - **Parameters**: user ID (URL parameter)
        - **Return Value**: JSON object of the user
        - **Description**: Fetches a specific user based on the provided ID.
    3. **POST /users**
        - **Parameters**: JSON object with discord_id, username, and email
        - **Return Value**: JSON object of the created user
        - **Description**: Adds a new user to the database.
    4. **PUT /users/:id**
        - **Parameters**: user ID (URL parameter), JSON object with updated username and email
        - **Return Value**: JSON object of the updated user
        - **Description**: Updates the details of an existing user.
    5. **DELETE /users/:id**
        - **Parameters**: user ID (URL parameter)
        - **Return Value**: JSON object with a success message
        - **Description**: Removes a user from the database.
    6. **GET /users/:id/groups**
        - **Parameters**: user ID (URL parameter)
        - **Return Value**: JSON array of group objects
        - **Description**: Fetches all groups that the specified user is a member of.
    7. **PUT /users/:id/ban**
        - **Parameters**: user ID (URL parameter), JSON object with banned status
        - **Return Value**: JSON object of the banned user
        - **Description**: Updates the banned status of a user.

4.2. Databases

1. **GameOnDB (MySQL)**
    - **Purpose**: It will store several main tables: user profile information, admin details, game details, group details, group member details, matchmaking preferences, and reports on users who have

violated application guidelines. We will use SQL since this data is relational and well-suited to our needs.

2. **Redis**

    - **Purpose**: It will be used for session management, storing session data such as user authentication tokens and temporary session information. Redis is chosen for its speed and efficiency in handling in-memory data storage, which is crucial for managing user sessions effectively. Additionally, Redis will be used for caching frequently accessed data to improve performance and reduce database load. Redis will also be used for storing and managing matchmaking queues and preferences temporarily, ensuring efficient and timely matching of users based on their preferences.

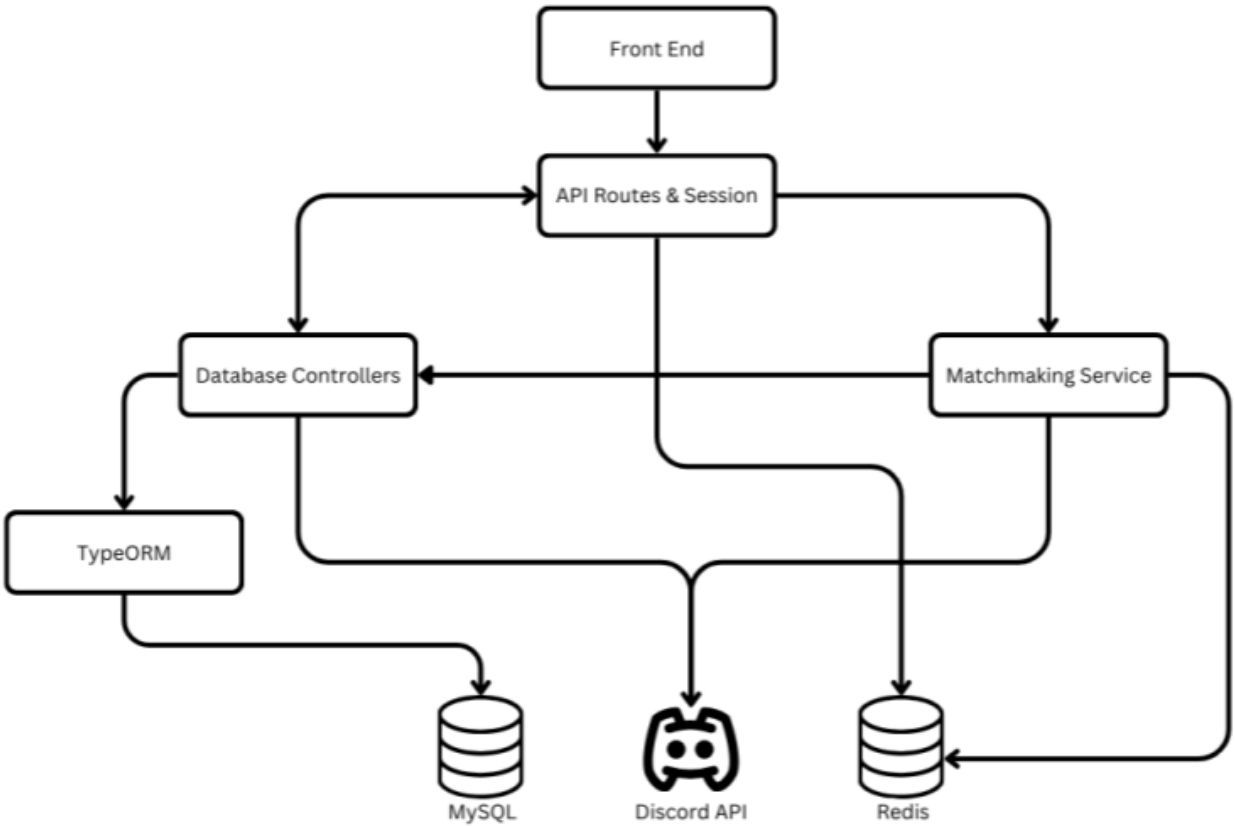
4.3. External Modules

1. **Discord API**
    - **Purpose**: We will use the Discord API for authentication within the app and for creating matchmaking groups. The groups will be created on Discord, as gamers widely use the platform and have a large, active community.

4.4. Frameworks

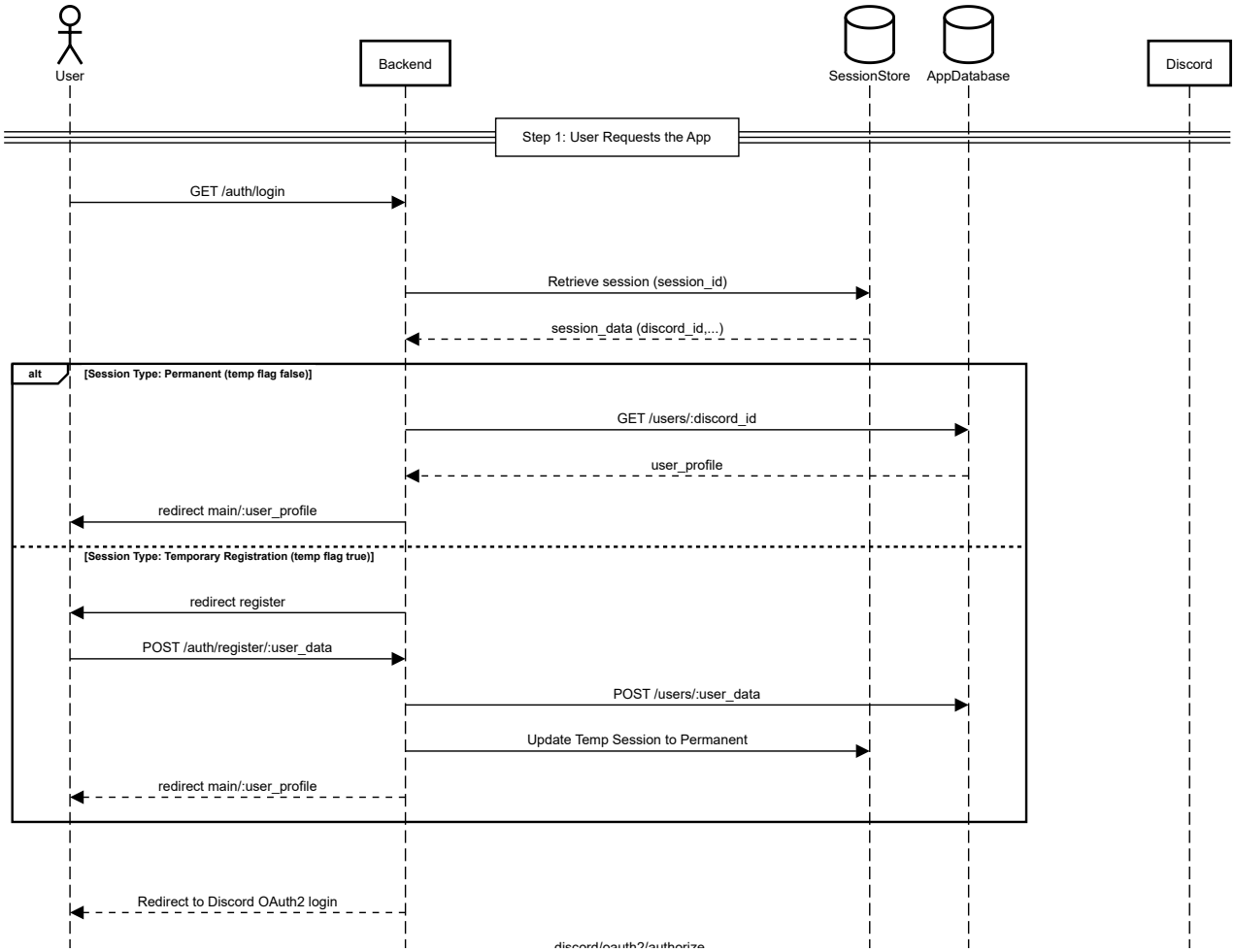
1. **Node.js**
    - **Purpose**: Developing the back-end server of the application.
    - **Reason**: Course requirement.
2. **Azure Virtual Machine**
    - **Purpose**: We are using a VM for hosting the back-end server.
    - **Reason**: Azure is the most financially viable option with a student account.
3. **Azure Database for MySQL**
    - **Purpose**: We will use a MySQL server to store our data.
    - **Reason**: We will use a SQL database since this data is relational and well-suited to our needs. It remains in the same cloud service as our VM.

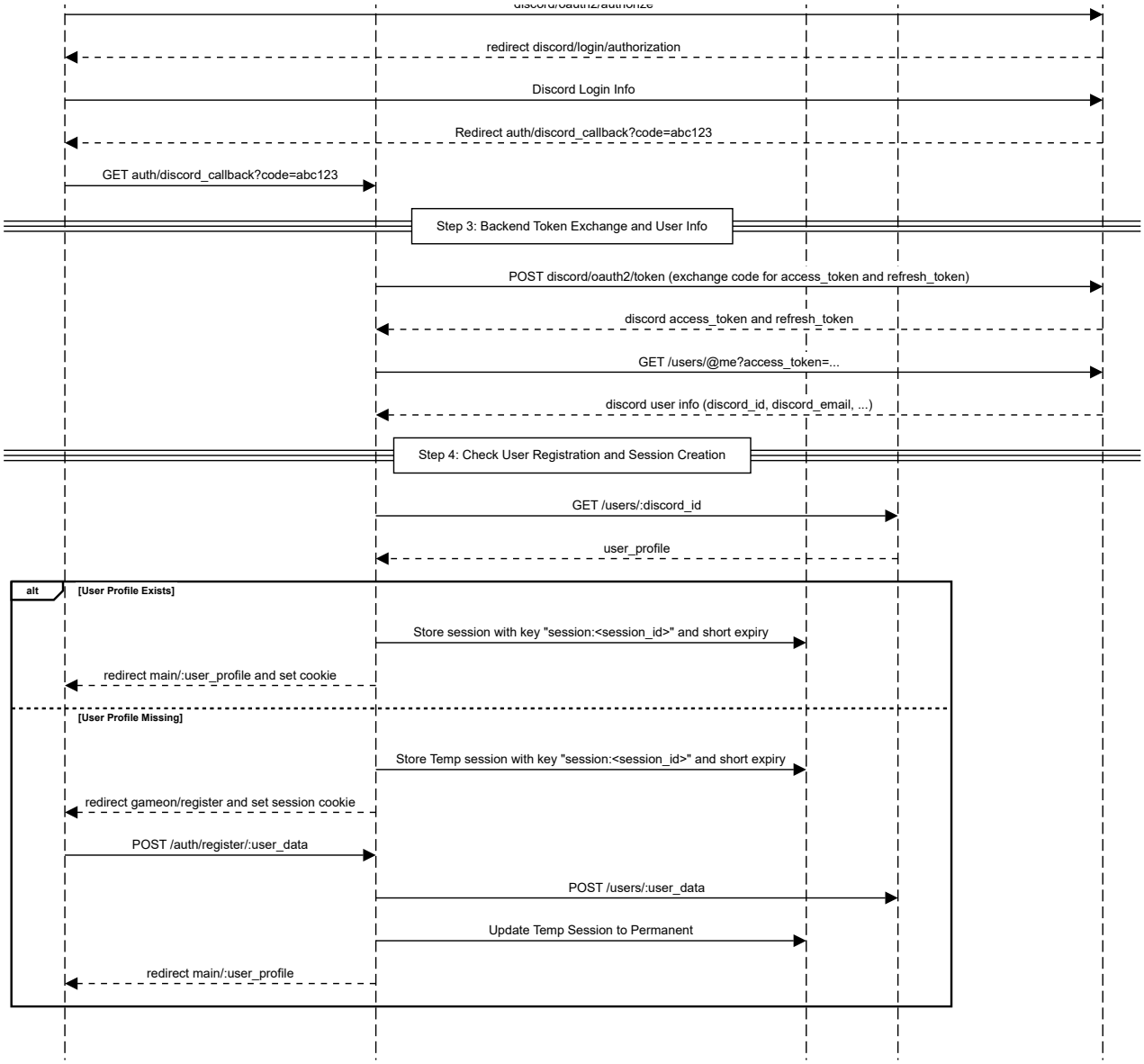
4.5. Dependencies Diagram



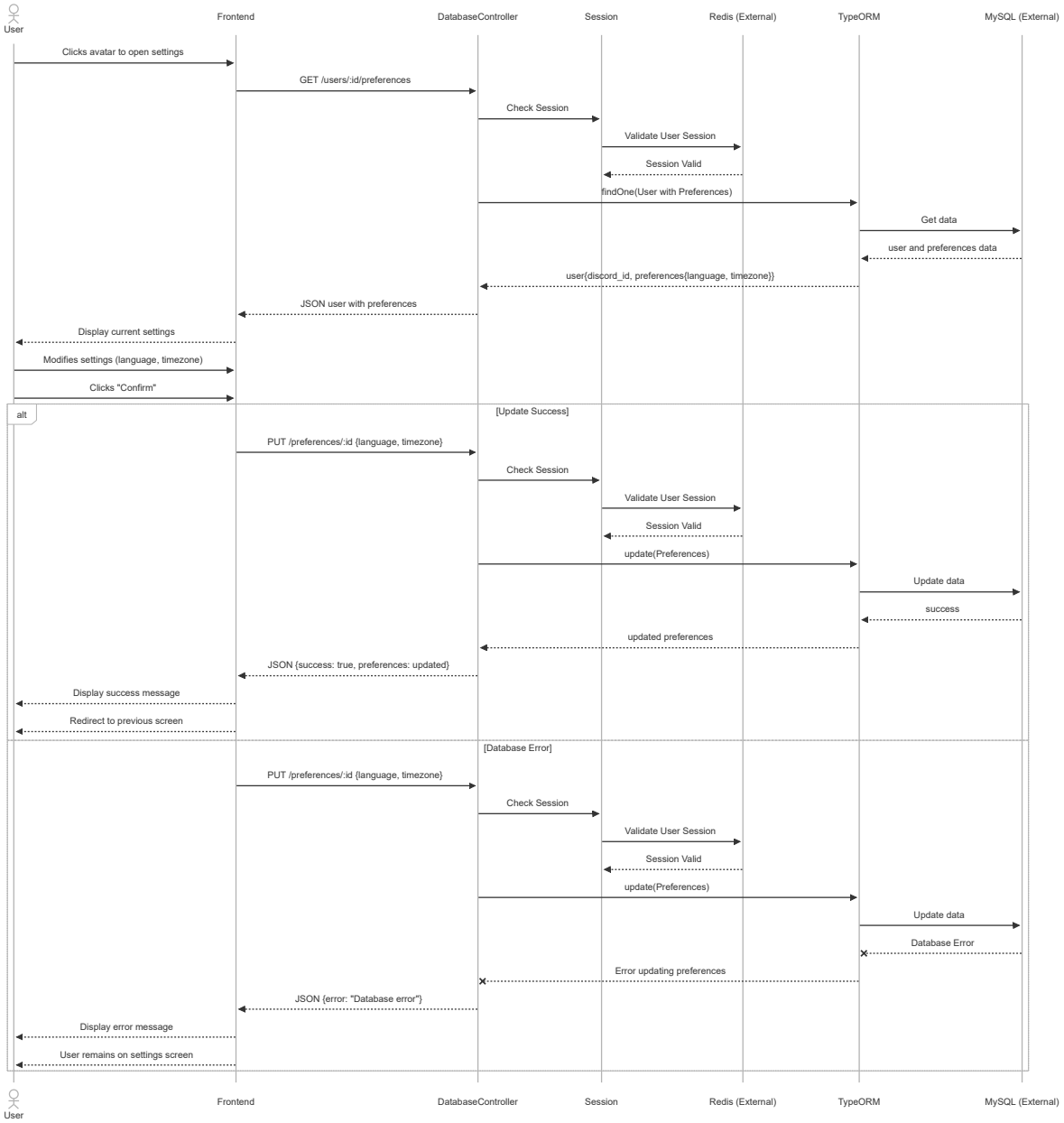
4.6. Functional Requirements Sequence Diagram

1. **Sign In**

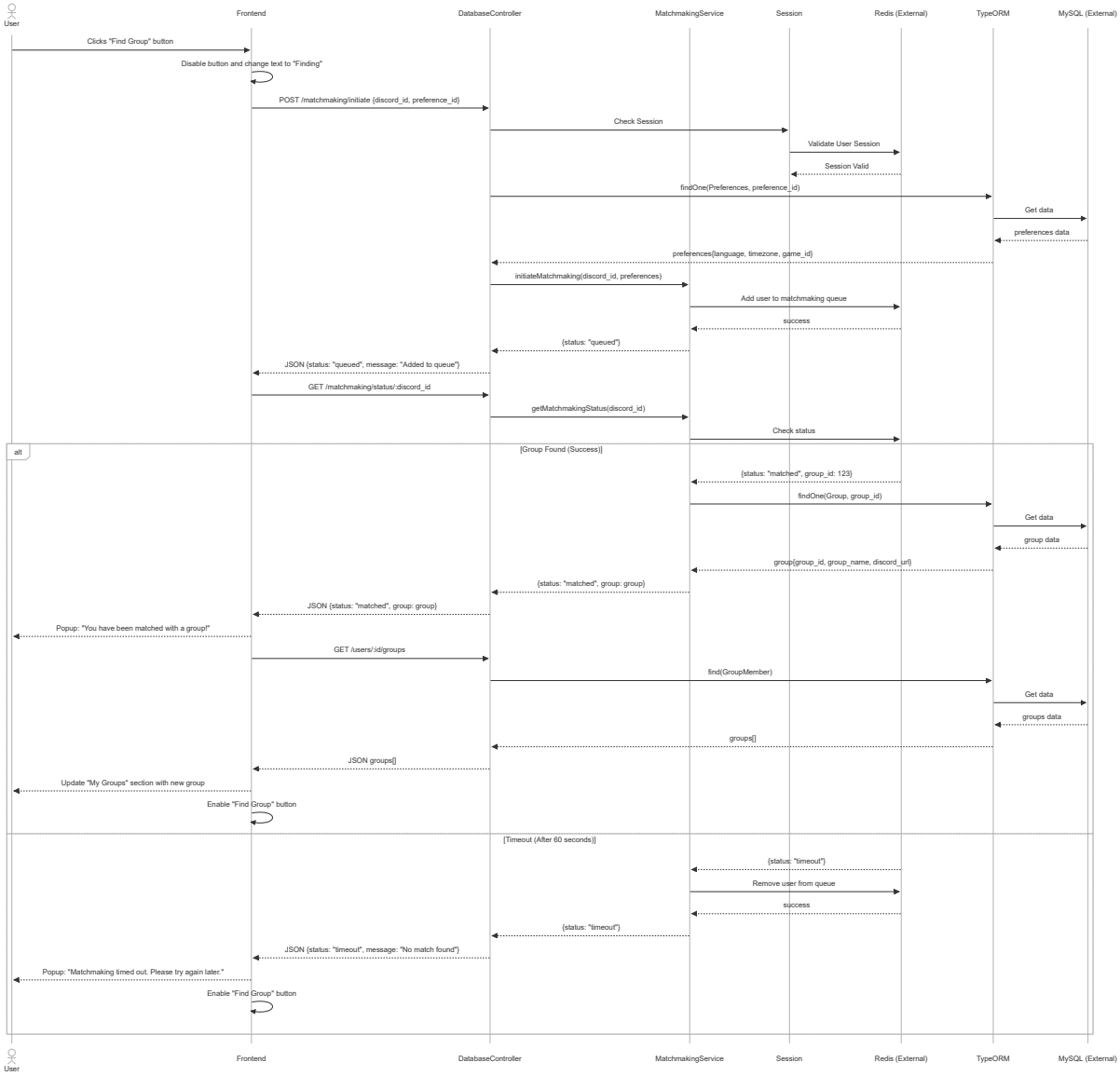



redirect discord/login/authorization
Discord Login Info
Redirect auth/discord_callback?code=abc123
GET auth/discord_callback?code=abc123
Step 3: Backend Token Exchange and User Info
POST discord/oauth2/token (exchange code for access_token and refresh_token)
discord access_token and refresh_token
GET /users/@me?access_token=...
discord user info (discord_id, discord_email, ...)
Step 4: Check User Registration and Session Creation
GET /users/:discord_id
user_profile
alt
[User Profile Exists]
Store session with key "session:<session_id>" and short expiry
redirect main/:user_profile and set cookie
[User Profile Missing]
Store Temp session with key "session:<session_id>" and short expiry
redirect gameon/register and set session cookie
POST /auth/register/:user_data
POST /users/:user_data
Update Temp Session to Permanent
redirect main/:user_profile

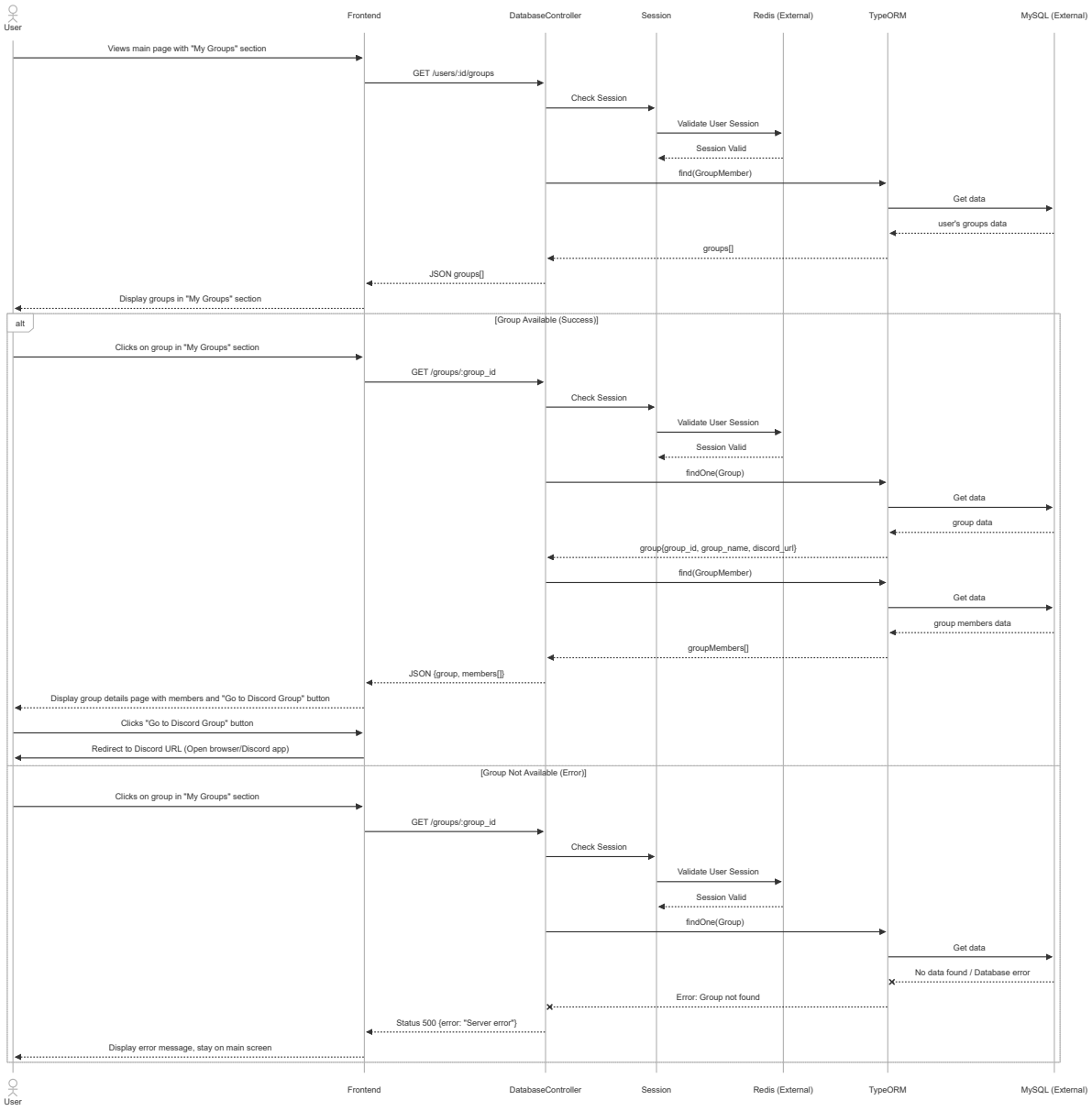
2. **User Settings**



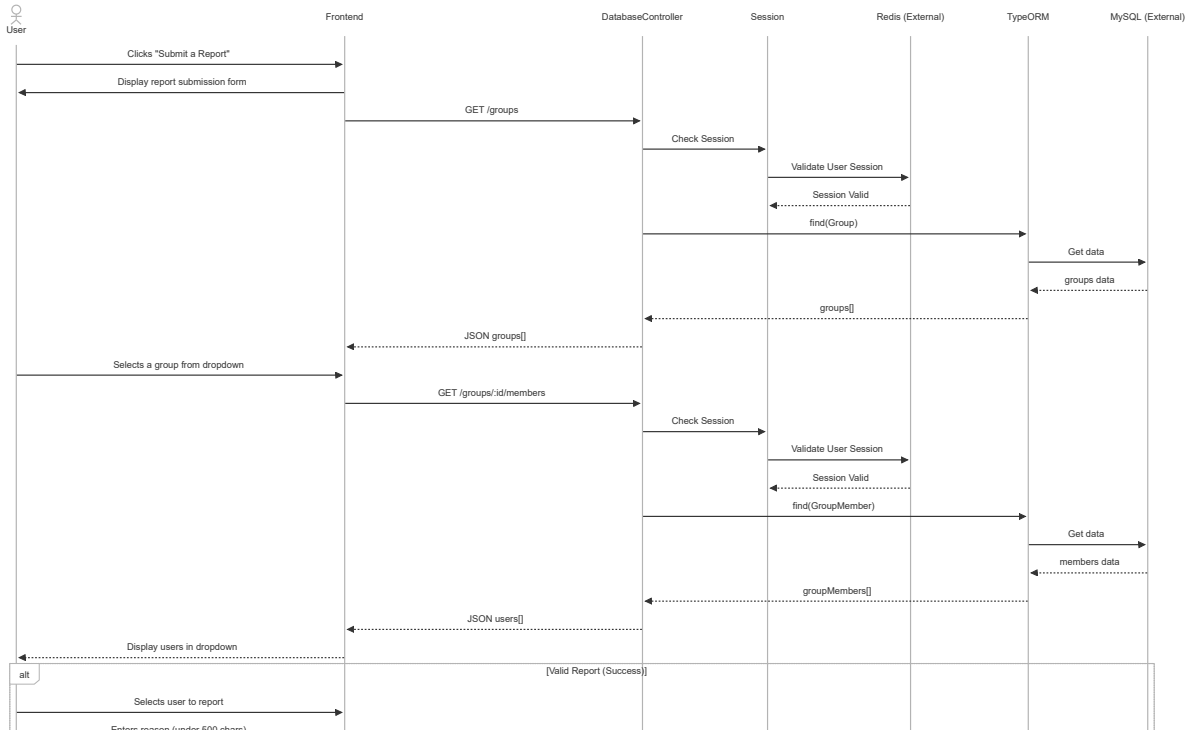
3. **Find Group**

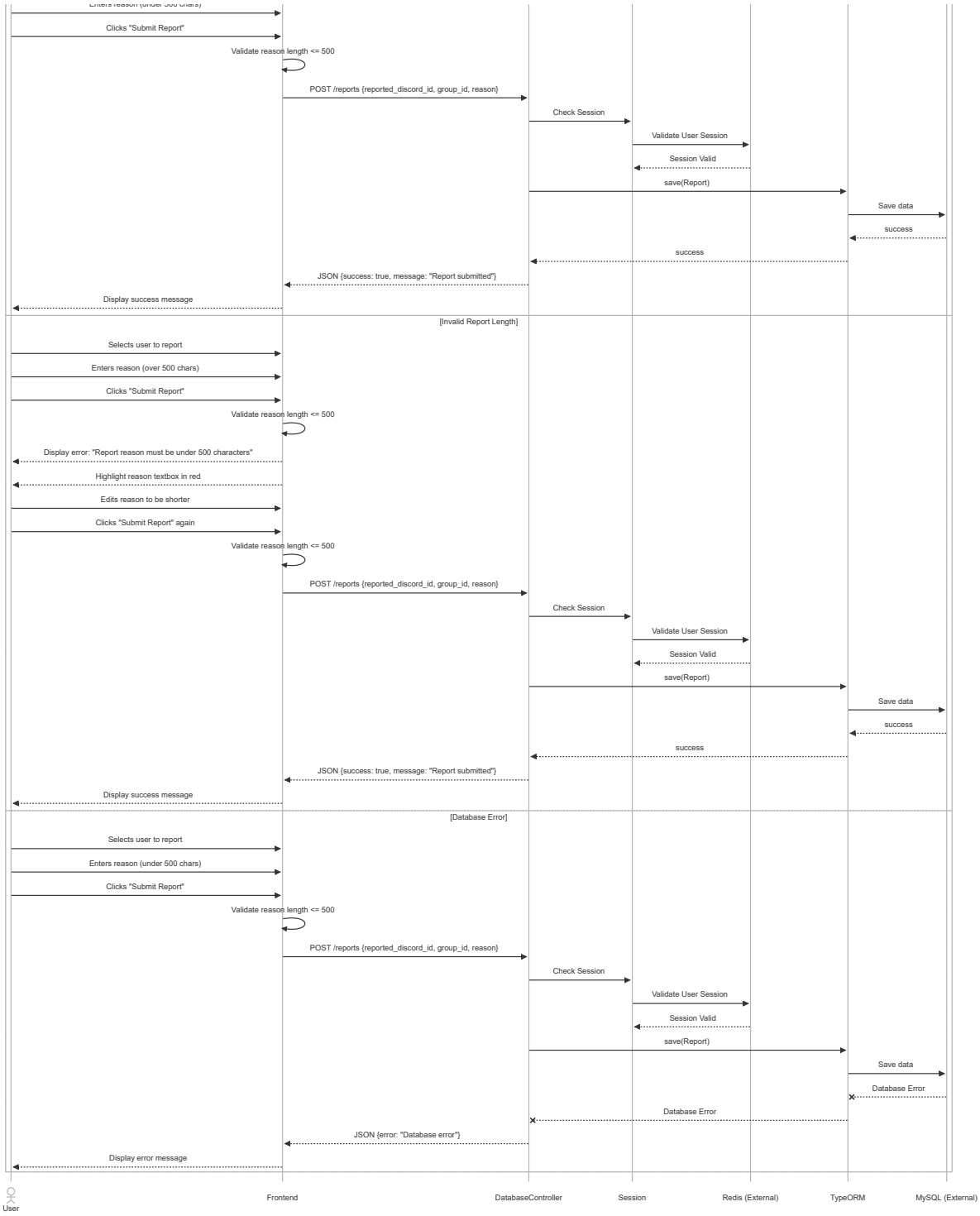

User
Frontend
DatabaseController
MatchmakingService
Session
Redis (External)
TypeORM
MySQL (External)
Clicks "Find Group" button
Disable button and change text to "Finding"
POST /matchmaking/initiate {discord_id, preference_id}
Check Session
Validate User Session
Session Valid
findOne(Preferences, preference_id)
Get data
preferences data
preferences{language, timezone, game_id}
initiateMatchmaking(discord_id, preferences)
Add user to matchmaking queue
success
{status: "queued"}
JSON {status: "queued", message: "Added to queue"}
GET /matchmaking/status/:discord_id
getMatchmakingStatus(discord_id)
Check status
alt
[Group Found (Success)]
{status: "matched", group_id: 123}
findOne(Group, group_id)
Get data
group data
group{group_id, group_name, discord_url}
{status: "matched", group: group}
JSON {status: "matched", group: group}
Popup: "You have been matched with a group!"
GET /users/:id/groups
find(GroupMember)
Get data
groups data
groups[]
JSON groups[]
Update "My Groups" section with new group
Enable "Find Group" button
[Timeout (After 60 seconds)]
{status: "timeout"}
Remove user from queue
success
{status: "timeout"}
JSON {status: "timeout", message: "No match found"}
Popup: "Matchmaking timed out. Please try again later."
Enable "Find Group" button

4. Navigate to Existing Group

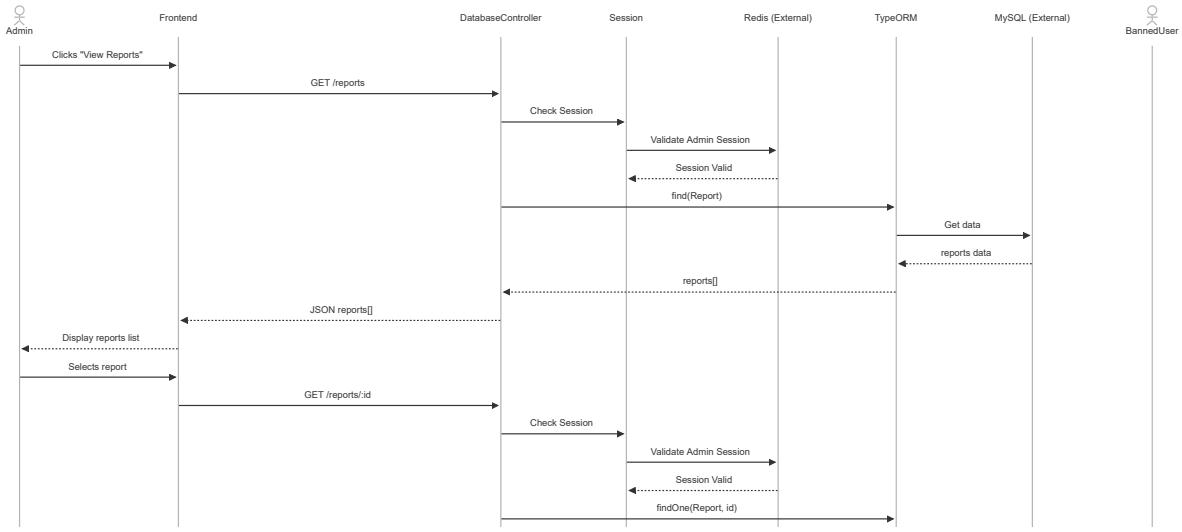


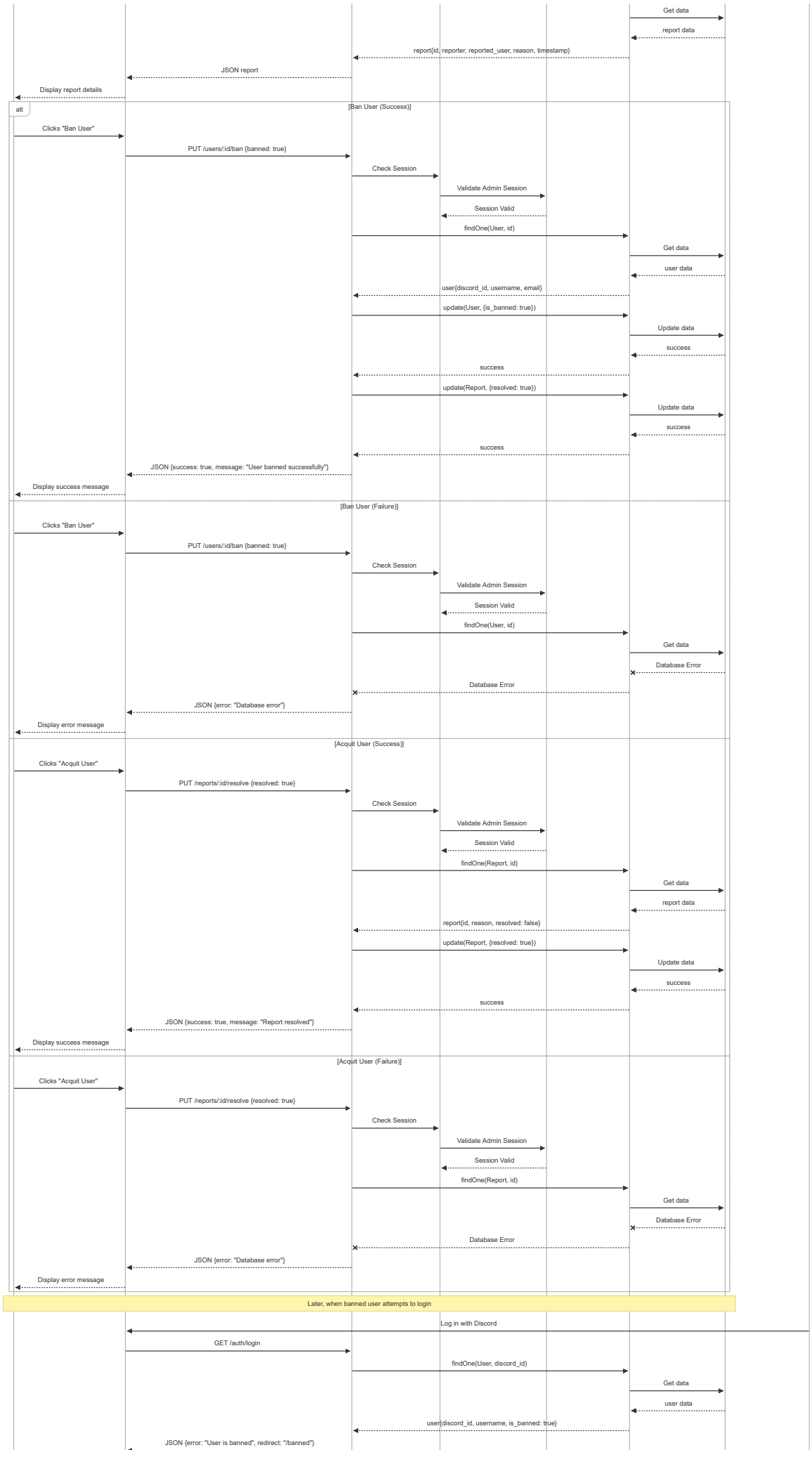
5. Report User





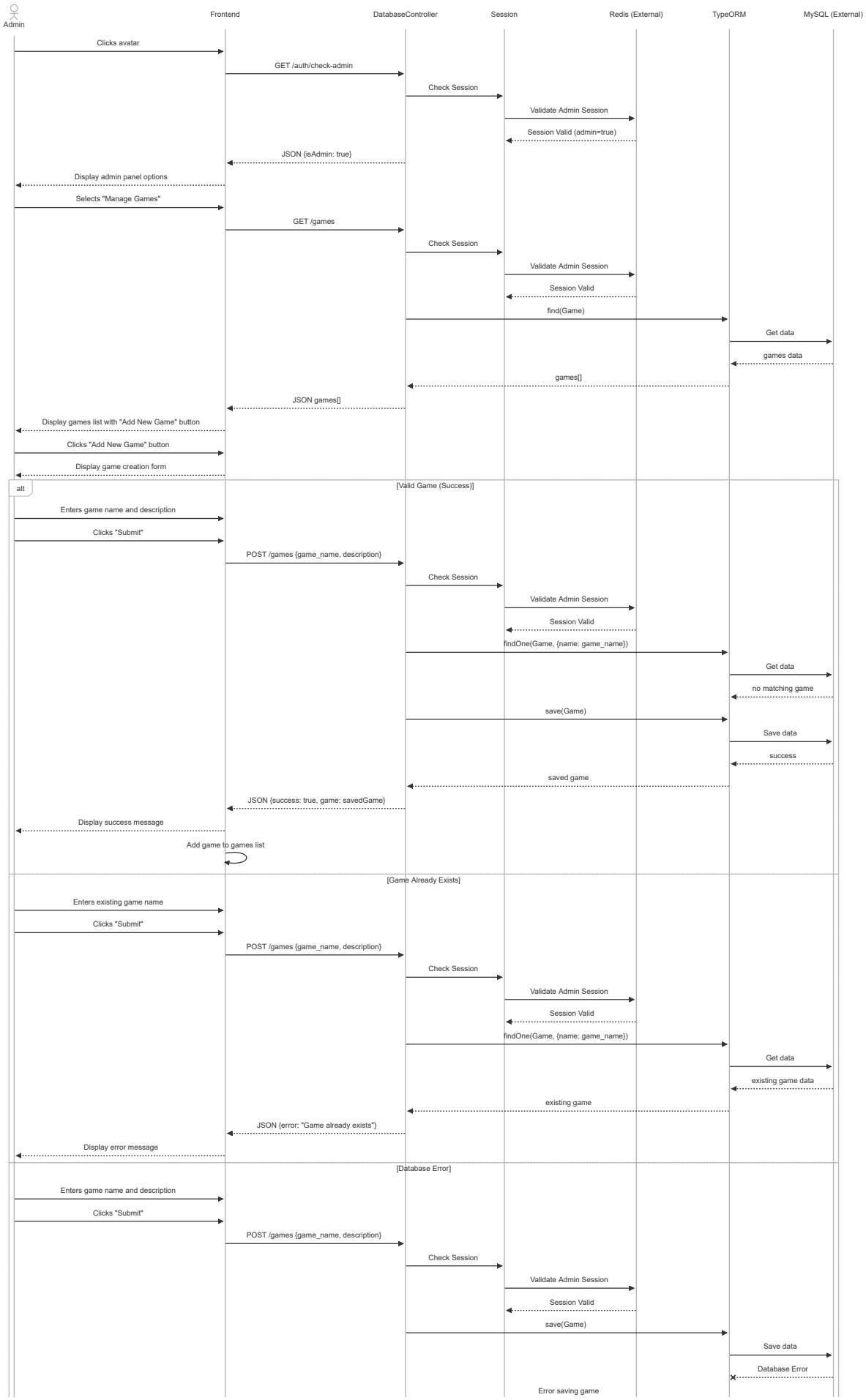
6. Handle Reports

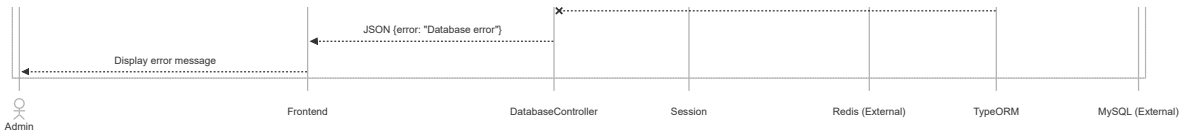



Get data
report data
report{id, reporter, reported_user, reason, timestamp}
JSON report
Display report details
alt
[Ban User (Success)]
Clicks "Ban User"
PUT /users/:id/ban {banned: true}
Check Session
Validate Admin Session
Session Valid
findOne(User, id)
Get data
user data
user{discord_id, username, email}
update(User, {is_banned: true})
Update data
success
success
update(Report, {resolved: true})
Update data
success
success
JSON {success: true, message: "User banned successfully"}
Display success message
[Ban User (Failure)]
Clicks "Ban User"
PUT /users/:id/ban {banned: true}
Check Session
Validate Admin Session
Session Valid
findOne(User, id)
Get data
Database Error
Database Error
JSON {error: "Database error"}
Display error message
[Acquit User (Success)]
Clicks "Acquit User"
PUT /reports/:id/resolve {resolved: true}
Check Session
Validate Admin Session
Session Valid
findOne(Report, id)
Get data
report data
report{id, reason, resolved: false}
update(Report, {resolved: true})
Update data
success
success
JSON {success: true, message: "Report resolved"}
Display success message
[Acquit User (Failure)]
Clicks "Acquit User"
PUT /reports/:id/resolve {resolved: true}
Check Session
Validate Admin Session
Session Valid
findOne(Report, id)
Get data
Database Error
Database Error
JSON {error: "Database error"}
Display error message
Later, when banned user attempts to login
Log in with Discord
GET /auth/login
findOne(User, discord_id)
Get data
user data
user{discord_id, username, is_banned: true}
JSON {error: "User is banned", redirect: "/banned"}


Admin
Frontend
DatabaseController
Session
Redis (External)
TypeORM
MySQL (External)
BannedUser
Display "Account Banned" screen

7. **Add Game**


Admin
Frontend
DatabaseController
Session
Redis (External)
TypeORM
MySQL (External)
Clicks avatar
GET /auth/check-admin
Check Session
Validate Admin Session
Session Valid (admin=true)
JSON {isAdmin: true}
Display admin panel options
Selects "Manage Games"
GET /games
Check Session
Validate Admin Session
Session Valid
find(Game)
Get data
games data
games[]
JSON games[]
Display games list with "Add New Game" button
Clicks "Add New Game" button
Display game creation form
alt
[Valid Game (Success)]
Enters game name and description
Clicks "Submit"
POST /games {game_name, description}
Check Session
Validate Admin Session
Session Valid
findOne(Game, {name: game_name})
Get data
no matching game
save(Game)
Save data
success
saved game
JSON {success: true, game: savedGame}
Display success message
Add game to games list
[Game Already Exists]
Enters existing game name
Clicks "Submit"
POST /games {game_name, description}
Check Session
Validate Admin Session
Session Valid
findOne(Game, {name: game_name})
Get data
existing game data
existing game
JSON {error: "Game already exists"}
Display error message
[Database Error]
Enters game name and description
Clicks "Submit"
POST /games {game_name, description}
Check Session
Validate Admin Session
Session Valid
save(Game)
Save data
Database Error
Error saving game



4.7. Non-Functional Requirements Design

1. **Session-Based Endpoint Security**
    - **Validation**: Critical endpoints will use session validation middleware to restrict access to authenticated users. Manual and automated tests will verify that unauthenticated requests are rejected with a 401 Unauthorized response.
2. **Report Reason Character Limit**
    - **Validation**: A 500-character limit will be enforced through input validation. Manual and automated tests will confirm that inputs exceeding the limit are rejected with appropriate error messages.

4.8. Main Project Complexity Design

Real-Time Gaming Group Matchmaking with Redis Queue

- **Description**: Our matchmaking system creates gaming groups based on user preferences including **language**, **time zone**, and **game choice**. It uses **Redis** as a real-time queue to efficiently manage matchmaking requests and **TypeORM** to persist data once matches are made.
- **Why complex?**:
    - **Concurrent real-time matchmaking**: System handles multiple users joining and leaving the queue simultaneously.
    - **Preference-based matching**: Users are matched based on multiple criteria rather than simple FIFO queuing.
    - **Distributed state management**: Redis maintains matchmaking state separately from the persistent database.
    - **Timeout handling**: Users are automatically removed from the queue after a timeout period if no match is found.
    - **Integration with external systems**: Successfully matched groups trigger Discord group creation.
- **Design**:
    - **Input**:
        1. **User matchmaking requests** containing:
            - User's `discord_id`
            - User's preferences (`language`, `time_zone`, `game_id`)
        2. **System parameters**:
            - Max **group size** (typically 4-6 players)
            - Timeout duration (60 seconds)
    - **Output**:
        - Successfully created gaming groups in both the database and Discord.

    - Timeout notifications when matches can't be found.

- **Main computational logic**:

    1. **Queue management**: When users initiate matchmaking, they're added to Redis queues organized by game preference.
    2. **Preference matching**: The system searches queues for users with compatible language and timezone preferences.
    3. **Group formation**: When enough compatible users are found, they're removed from the queue and a group is created.
    4. **Discord integration**: A Discord group is created for the newly matched users.
    5. **Persistence**: The group and its members are stored in the MySQL database via TypeORM.
    6. **Timeout handling**: Users who remain in the queue longer than the timeout period are removed and notified.

- **Pseudo-code**:

```
function matchmake(users, max_group_size, timeout):
    redis_queues = initializeRedisQueues(users)  // Organize users by
game preference
    while redis_queues are not empty:
        for queue in redis_queues:
            compatible_users = findCompatibleUsers(queue,
max_group_size)
            if compatible_users.length == max_group_size:
                createGroup(compatible_users)  // Create group in
Discord and persist in DB
                removeUsersFromQueue(compatible_users, queue)
            else:
                handleTimeouts(queue, timeout)  // Remove users
exceeding timeout
    return "Matchmaking complete"
```