M3 - Requirements and Design

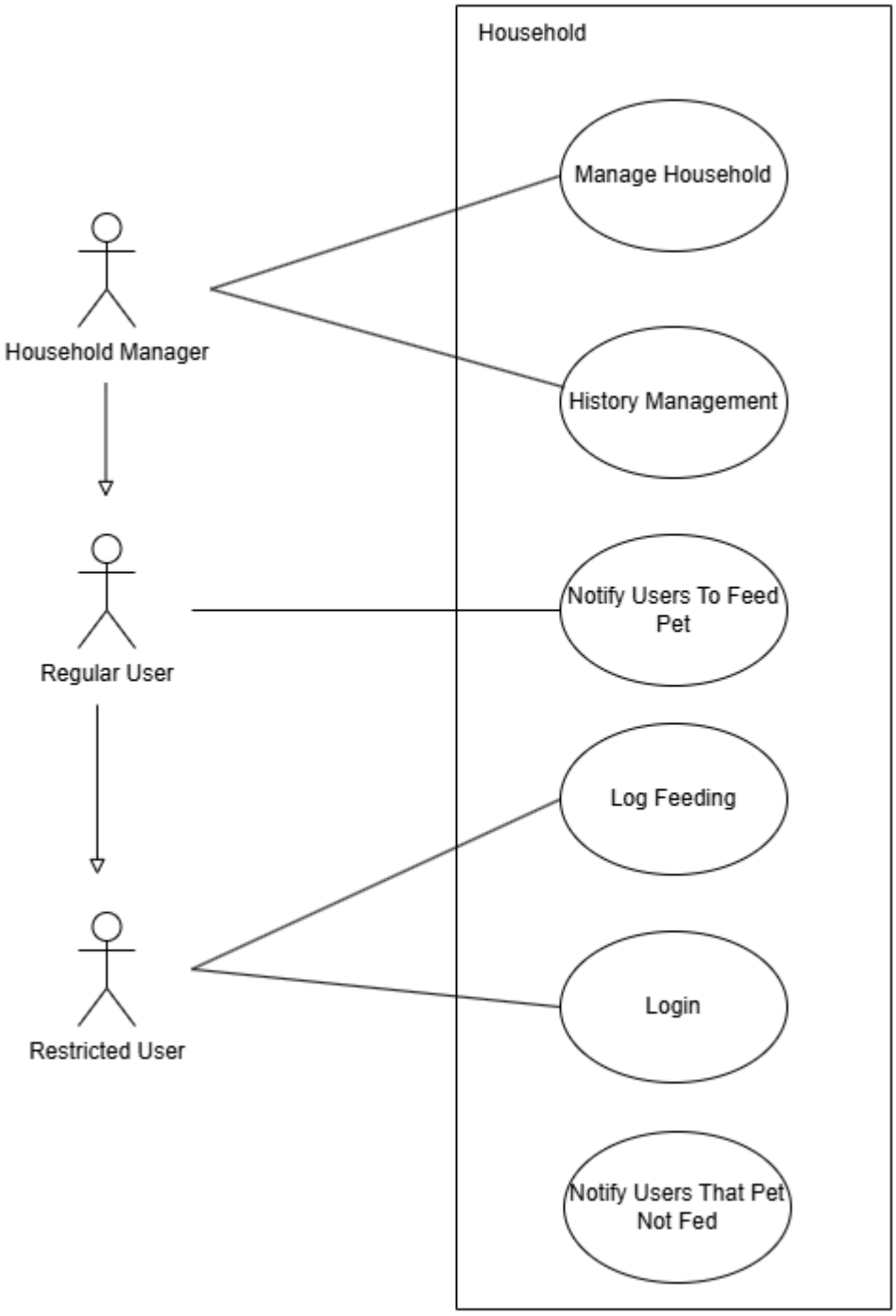
1. Change History

2. Project Description

“Did you feed the animals?” “Are the animals fed?” “Anyone feed the animals?” These texts flood the family group chat every day. The kids don’t respond. Mom is still at work. No one knows if the beasts have been fed. The beasts also lie, as though they are starving and wasting away even if they already got dinner. Many task managing apps out there are WAY too complicated. TAS is straightforward app that indicates whether or not your beloved household pet has been fed. To go even further, there will be a minimal mode, for those living with grandparents, small children, or a grumpy father that does not want to be bothered with a new app. Simple, easy, convenient.

3. Requirements Specification

3.1. Use-Case Diagram



3.2. Actors Description

1. **Household Manager**:
    - Most technically capable user in the household
    - Initiates the household
    - Will control the users, permissions, pets, and feeding schedule
    - Also has access to all of the below
2. **Household Member/Regular User**:
    - Average household member that will be able to feed the pet.
    - Can request that another regular user feed the pet
    - Receives notifications and feeding requests
3. **Restricted User**:
    - Household member who struggles with technology and needs/wants a limited UI

- Can submit to the app when they've fed the pet
- Can check the app to see whether they need to feed the pet or not

3.3. Functional Requirements

1. **Log Feeding**

   - **Overview**:

     1. Users must be able to log that a pet has been fed, ensuring accurate feeding records for tracking purposes.

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

     1. **Log Feeding Event**:
        - **Description**: Users log a feeding event by selecting the pet they've fed and confirming the action.
        - **Primary actor(s)**: All household member
        - **Main success scenario**:
          1. User scrolls through the list of pets on the base page to find the pet being fed.
          2. User presses the corresponding "Feed Pet" button to confirm that the pet has been fed.
          3. System updates the feeding log with the pet's ID, user ID, date, and amount of food.
          4. User is prompted with a success message indicating that the log has been updated successfully.
        - **Failure scenario(s)**:
          - 1a. App is unable retrieve the log data.
            - 1a1. User is redirected back to the home page and system displays an error message prompting the user to try again later.
          - 1c. App is unable to update the system
            - 1c1. User is redirected back to the pet's page after the failure and prompted that the logging was unsuccesful and should try again.
            - 1c2. User presses the "Feed Pet" button again to attempt logging the feeding once more

2. **Requesting Others to do Feeding**

   - **Overview**:

     1. Regular users must be able to notify other regular users to feed the pet. The receiving user must be notified that they are responsible for feeding the pet.

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

     1. **Feeding Request Notification**:
        - **Description**: Users can send a notification to another user, requesting them to feed the pet. The receiving user is notified of their responsibility to feed the pet.
        - **Primary actor(s)**: Household Manager, Regular User

- **Main success scenario**:
    1. Sender scrolls the main page to find the pet they want fed and selects the recipient in a droplist
    2. The sender then presses the “Notify” button
    3. App sends a request to the server to notify the recipient.
    4. User 2 (the recipient) receives a notification indicating that they are responsible for feeding the pet.
- **Failure scenario(s)**:
    - 1c. App is unable to send the request to the server.
        - 1c1. System displays and error message prompting the user to try again later
        - 1c2. User dismisses the message and is sent back to the list of pets to try again

3. **Managing The Household**

    - **Overview**:

        1. Household Manager must be able to set up and edit the household users
        2. Household Manager must be abe to manage the pets and set their feeding schedules.

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

        1. **Add Users**:
            - **Description**: Household manager can add users to the household
            - **Primary actor(s)**: Household Manager
            - **Main success scenario**:
                1. Manager clicks the "Manage Household" button on the top of the page and is directed to a page with the "Add User" button
                2. Manager is prompted to enter the email of the person they are inviting, with the corresponding restriction level
                3. Manger clicks "ok"
                4. The server adds the user to the household
                5. A toast message is displayed confirming that the user has been added
            - **Failure scenario(s)**:
                - 1d. The app is unable to update the server
                    - 1d1. The app displays an error message and prompts the user with a message to try again
                    - 1d2. The input form is re-displayed, and the user types in the code again
                - 2d. The user does not exist
                    - 2d1. The app displays an error message saying that the user does not exist and should make sure they typed in the ID correct and the person has an account on the app
                    - 2d2. The user dismisses the message and is redirected back to the manage household page
        2. **Manage Pets**:
            - **Description**: Household manager can add pets and change their feeding schedules

- **Primary actor(s)**: Household manager
- **Main success scenario**:
    1. Manager clicks the "Manage Household" button on the top of the page and is directed to a page with the "Manage Pets" button
    2. Manager clicks the "Add Pet" button or selects the edit icon on the pet they want to update, and the fields becomes editable
    3. Manger clicks "ok"
    4. The app updates the database with these changes
    5. A toast message is displayed confirming that the changes have been made
- **Failure scenario(s)**:
    - 1b. Pets could not be retrieved
        - 1b1. If there are no pets yet, display text saying "Add Your Pets!")in the text display. Otherwise, display "error getting pets, try again later"
        - 1b2. The user backs out to the manage household screen and clicks manage pets to try again
    - 1d. The changes could not be made to the server
        - 1d1. An error message is displayed and the user is prompted to try again
        - 1d2. The user is directed back to the manage pets screen

4. **History Management**

    - **Overview**:

        1. Household Manager must be able to view the history of logs, including log time and log member.

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

        1. **Manage History**:
            - **Description**:
            - **Primary actor(s)**:
            - **Main success scenario**:
                1. Household Manager clicks the “View History” button on the main screen
                2. The history is retrieved and the user is directed to a new screen displaying the feeding history in the household
            - **Failure scenario(s)**:
                - 1b. Server is unable to retrieve the history
                    - 1b1. App displays an error message saying the history could not be retrieved and the user should try again later
                    - 1b2. The user backs out to the main screen and clicks the view history button again

5. **Feeding Time Notification**

    - **Overview**:

        1. User must receive a notification if it is time to feed the pet and the pet has not been fed.

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

1. **Feeding Notification**:
    - **Description**: The system sends a time-based notification to users when it's time to feed the pet, ensuring that pets are fed on schedule.
    - **Primary actor(s)**: Household Manager, Regular Users
    - **Main success scenario**:
        1. The feeding schedule for the pet is triggered based on the time set by the user.
        2. The system sends a push notification to the user indicating that it is time to feed the pet.
        3. User receives the push notification on their device
    - **Failure scenario(s)**:
        - 1b. The push notification fails to send
            - 1b1. The system fails to deliver the notification due to connectivity or server issues.
            - 1b2. The next time the app is opened, the user is told the request was failed and is prompted to check their settings and permissions.

6. **Login Authentication**

    - **Overview**:

        1. Users must be able to log in using Google authentication.

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

        1. **Login/Signup**:
            - **Description**: Users access the in-app account via their google account to gain access to the app
            - **Primary actor(s)**: All users
            - **Main success scenario**:
                1. User clicks the "Login Button"
                2. User is directed to the google authentication page
                3. If the user has an account, their household information is retrieved and they are sent to the main screen. Otherwise, if the user does not have an account, they are added to the database and displayed a welcome message asking them to join or initiate a household.
            - **Failure scenario(s)**:
                - 1b. User is unable to authenticate through Google.
                    - 1b1. The app encounters an issue during the authentication process (e.g., incorrect credentials, server error)
                    - 1b2. The system displays an error message prompting the user to try again.
                    - 1b3. The user dismisses the message and is redirected to the login screen
                - 1c. Household could not be initiated for a new user
                    - 1c1. The system displays an error message prompting the user to try again.

- 1c2. The user dismisses the message and is redirected back to the welcome page

3.5. Non-Functional Requirements

1. **Accessibility**
   - **Description**: The app should be usable by all members of the household, including those with impaired vision, language barriers, and of all mental faculties. Further, users should be able to complete the core actions (logging a feeding, viewing history) in 3 clicks or less.
   - **Justification**: This app is designed for people who aren't tech-savvy. By making it simple and intuitive, we ensure it's easy for all users, regardless of ability, to track pet feedings with minimal effort. Core actions are streamlined to 3 clicks or less, eliminating complexity and making pet care accessible for everyone in the household.
2. **Maintainability**
   - **Description**: Logs should persist for at least a year
   - **Justification**: A key secondary function of the app is to store feeding history. This data can be valuable for pet owners who wish to track their pets' diets, portion sizes, and feeding patterns over time. Retaining this information is useful for various purposes, such as monitoring health trends, managing pet budgets, and identifying changes in feeding behavior. By ensuring that logs are kept for a full year, the app provides lasting value and insight for users

4. Designs Specification

4.1. Main Components

1. **User & Household Management Service**

   - **Purpose**: Manages user authentication (via Google API), household creation, and user roles. Ensures that users belong to a household before accessing pets and feeding logs. Separating user management from pet and feeding history ensures scalability and maintainability. Alternative being embedding user management in pet tracking would lead to tight coupling and harder role-based access control.
   - **Interfaces**:

     1. `User login(String email);`
        - **Purpose**: Verifies email and returns user details.
     2. `Household createHousehold(String householdName, String ownerUserId);`
        - **Purpose**: Creates a new household with an owner.
     3. `boolean addUser(String email);`
        - **Purpose**: Adds an existing user to a household. Returns `true` on success.
     4. `boolean removeUserFromHousehold(String householdId, String userId);`
        - **Purpose**: Removes a user from a household. Returns `true` on success.
     5. `List<User> getUnrestrictedUsers();`

- **Purpose**: Gets all the users that are not in restricted mode.

2. **Pet Management Service**

- **Purpose**: Tracks pets in a household and their feeding schedules. Keeping pet data separate from user management allows for future expansions like health tracking. Merging it with user management would complicate database queries.
- **Interfaces**:

1. `Pet addPet(String householdId, String petName, String species, String feedingSchedule);`

   - **Purpose**: Adds a pet to a household.

2. `boolean removePet(String petId);`

   - **Purpose**: Removes a pet from the system. Returns `true` on success.

3. `List<Pet> getPetsByHousehold(String householdId);`

   - **Purpose**: Retrieves all pets for a given household.

3. **Logging**

- **Purpose**: Records and retrieves feeding history, ensuring users can check when and who last fed a pet. Keeping a dedicated feeding log service prevents bloating the pet management component. Embedding it in pet service would create unnecessary dependencies between pet data and feeding logs, and may lead to complications like calculations tied to the household rather than a specific pet.
- **Interfaces**:
  1. Pet addPet(String householdId, String petName, String species, String feedingSchedule);
     - **Purpose**: Adds a pet to a household.
  2. boolean removePet(String petId);
     - **Purpose**: Removes a pet from a household.
  3. List getPetsByHousehold(String householdId);
     - **Purpose**: Retrieves all pets for a given household.

4. **Notifications**

- **Purpose**: Sends reminders to users when pets need to be fed and alerts them when a feeding is logged, also users to send requests to other users. Implementing notifications with users directly would bloat the user component and unnecessarily tie mass notifications to individual users.
- **Interfaces**:
  1. sendNotifcations(message, recipients[])
     - **Purpose**: interacts with the notification service. Sent by application
  2. sendNotifcations(message, senderID, recipients[])
     - **Purpose**: interacts with the notification service. Used when sending request to feed.

4.2. Databases

1. **MongoDB (PetTrackerDB)**

- **Purpose**: Stores household information, users, pets, feeding history, and notification preferences.
- **Reason**: A NoSQL database provides flexibility for dynamic data structures (e.g., pets with different feeding schedules) and supports fast reads/writes, making it ideal for tracking events like feedings.

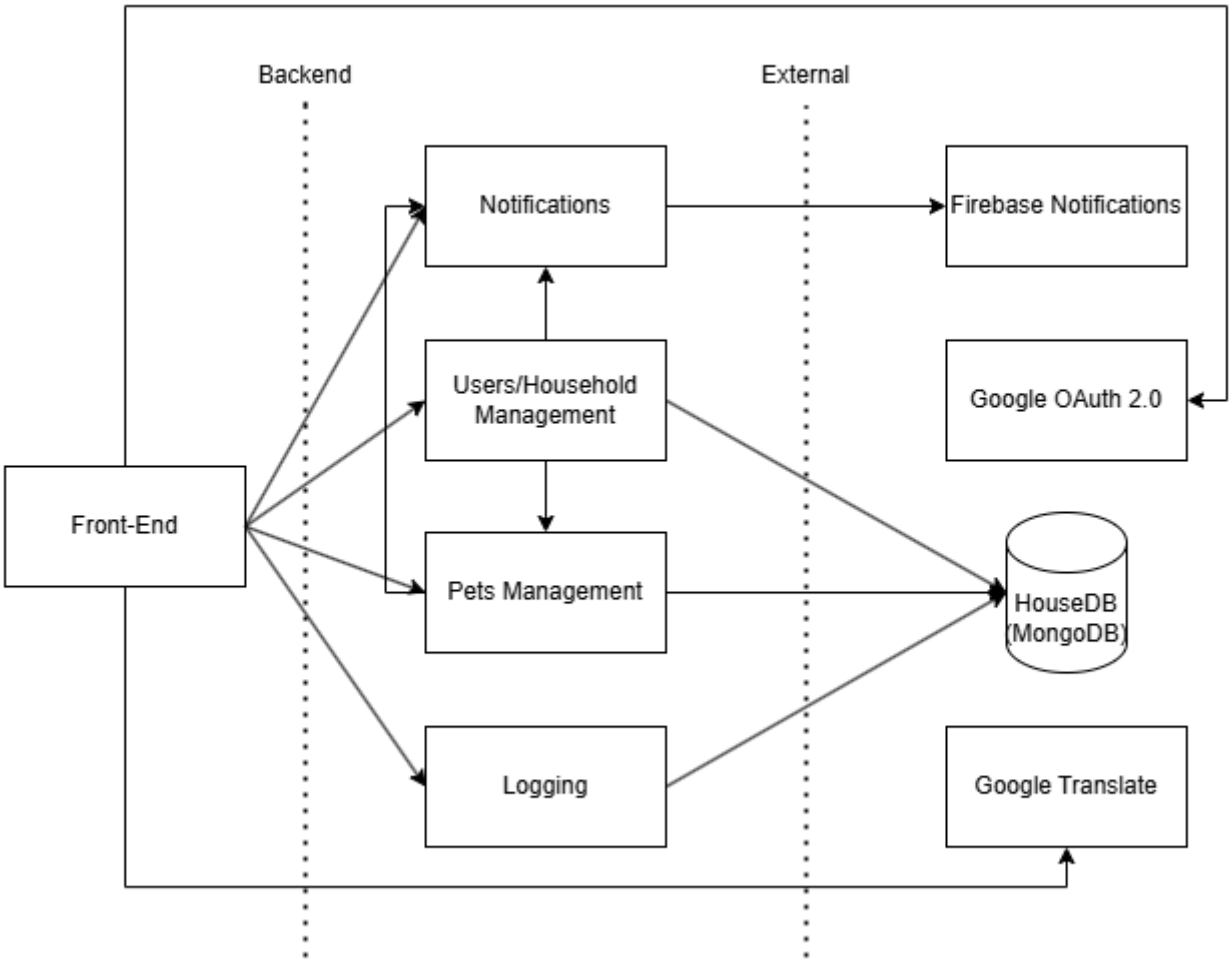
4.3. External Modules

1. **Google OAuth 2.0**
    - **Purpose**: Handles user authentication, allowing users to log in with their Google accounts securely.
    - **Reason**: Reduces the need for custom authentication logic, ensuring security and ease of use.
2. **Firebase Cloud Messaging (FCM)**
    - **Purpose**: Sends push notifications to users for feeding reminders, missed feedings, and confirmations.
    - **Reason**: Free for basic use, reliable, and integrates seamlessly with mobile apps.
3. **Google Translate API**
    - **Purpose**: Provides translation services for notifications and feeding logs to accommodate multilingual households.
    - **Reason**: Supports real-time translations with a simple API call, making the app more accessible to families with non-English speakers.

4.4. Frameworks

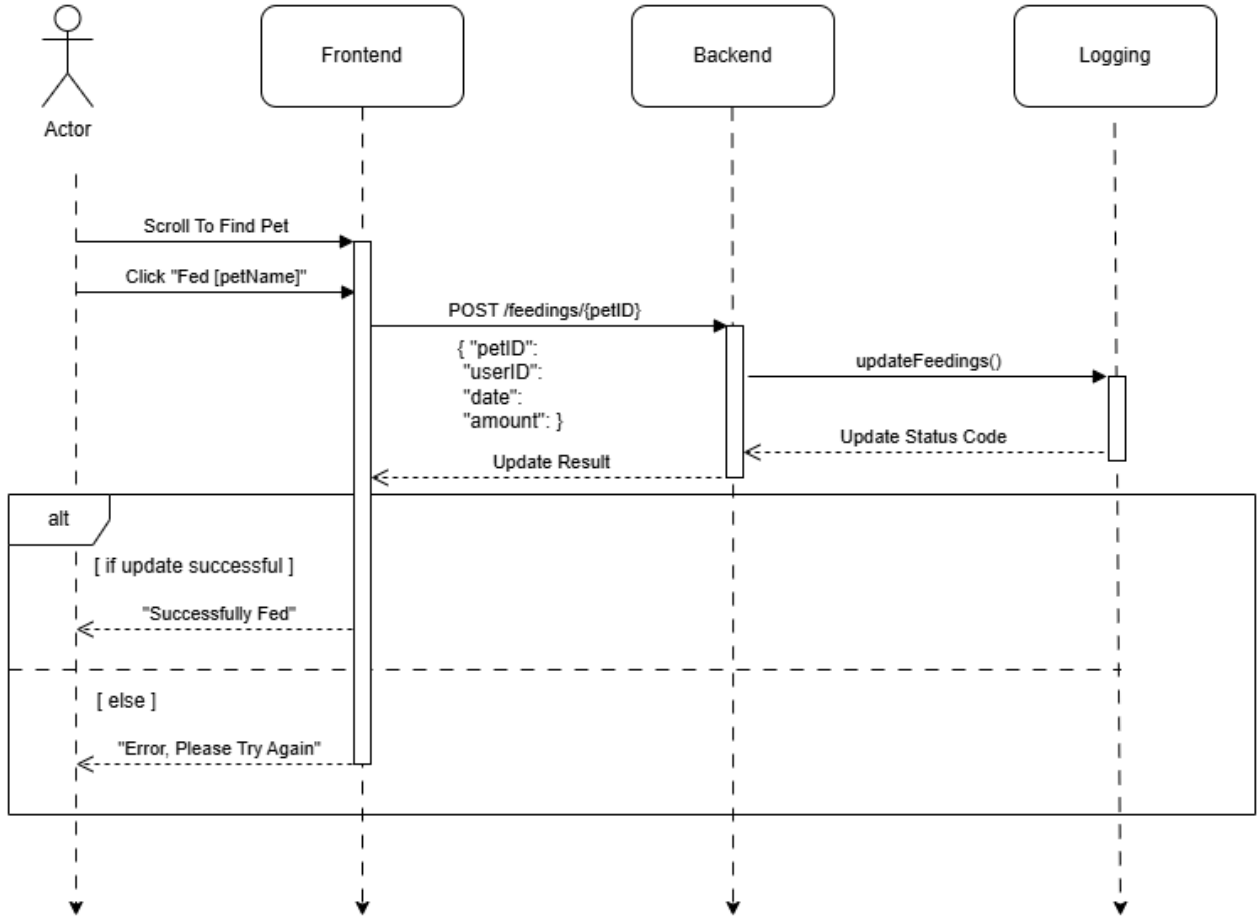
1. **Express.js (Node.js)**
    - **Purpose**: Backend framework for handling API logic, database operations, and service orchestration.
    - **Reason**: Express.js is lightweight and fast for handling RESTful APIs.
2. **Docker**
    - **Purpose**: Containerization of the backend and frontend services for consistent deployments.
    - **Reason**: Ensures that the application runs the same way across different environments, making development and deployment easier.
3. **AWS EC2**
    - **Purpose**: Hosts the backend API and database services.
    - **Reason**: Provides scalable infrastructure, integrates well with MongoDB and AWS services, and allows auto-scaling based on demand.

4.5. Dependencies Diagram

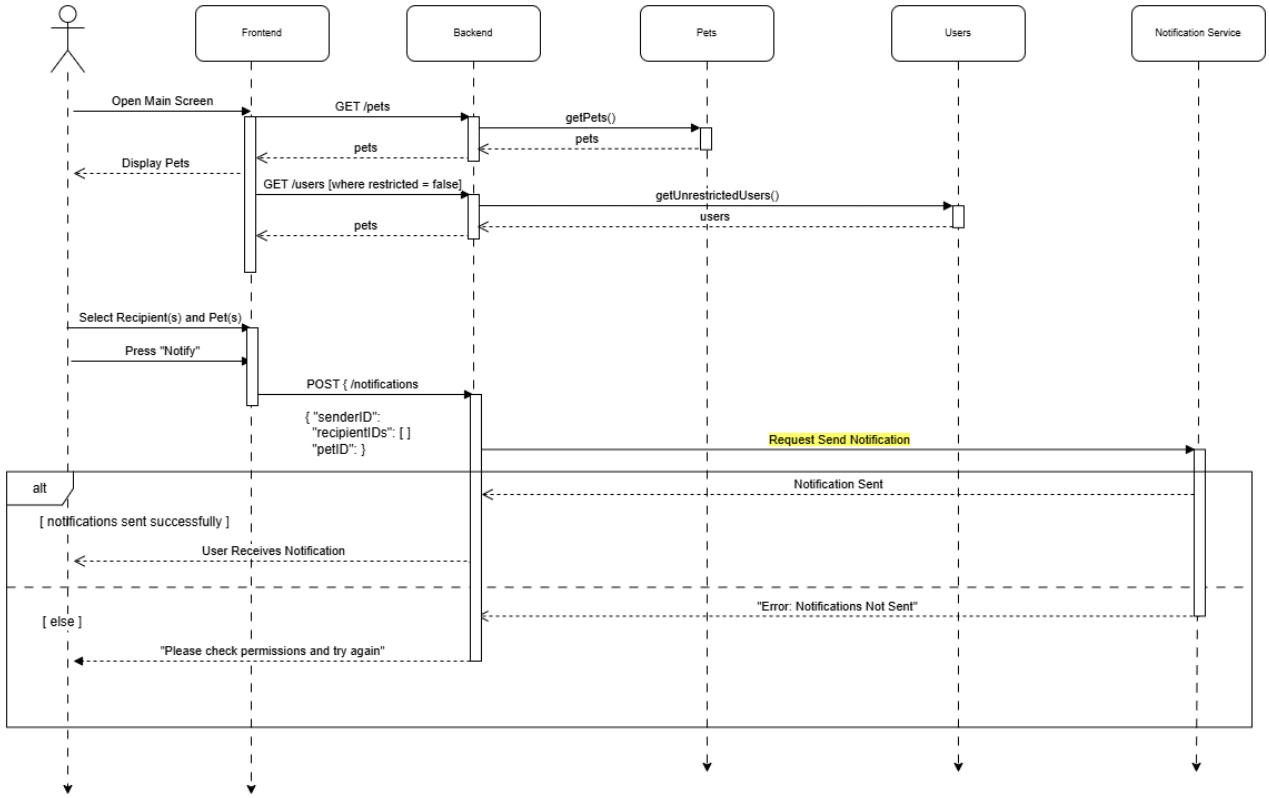


4.6. Functional Requirements Sequence Diagram

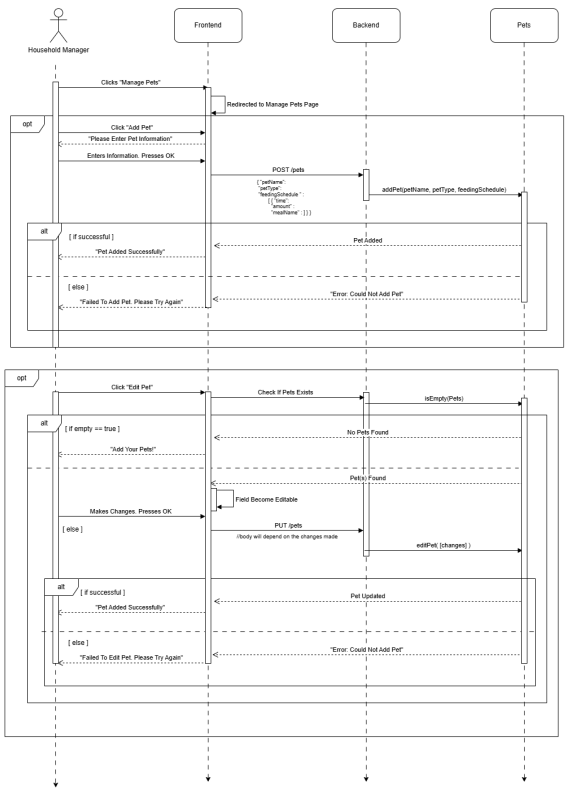
1. **Log Feeding**

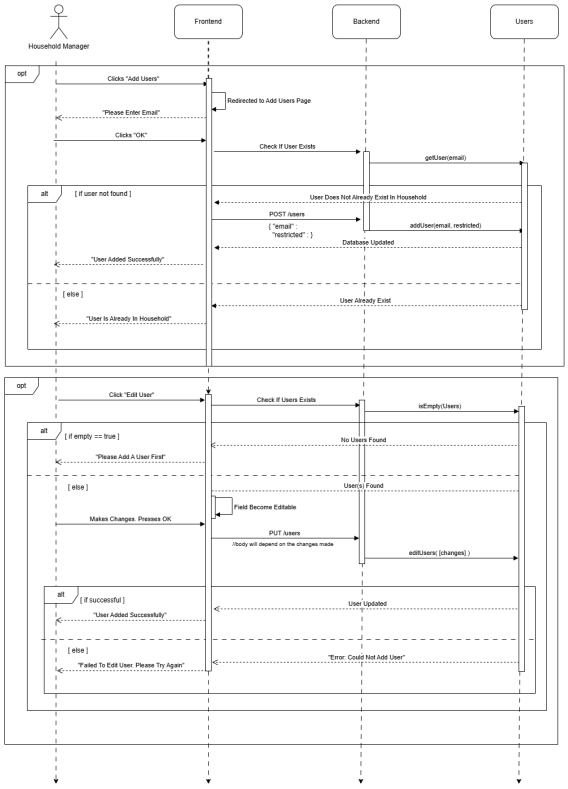


2. **Requesting Others to do Feeding**

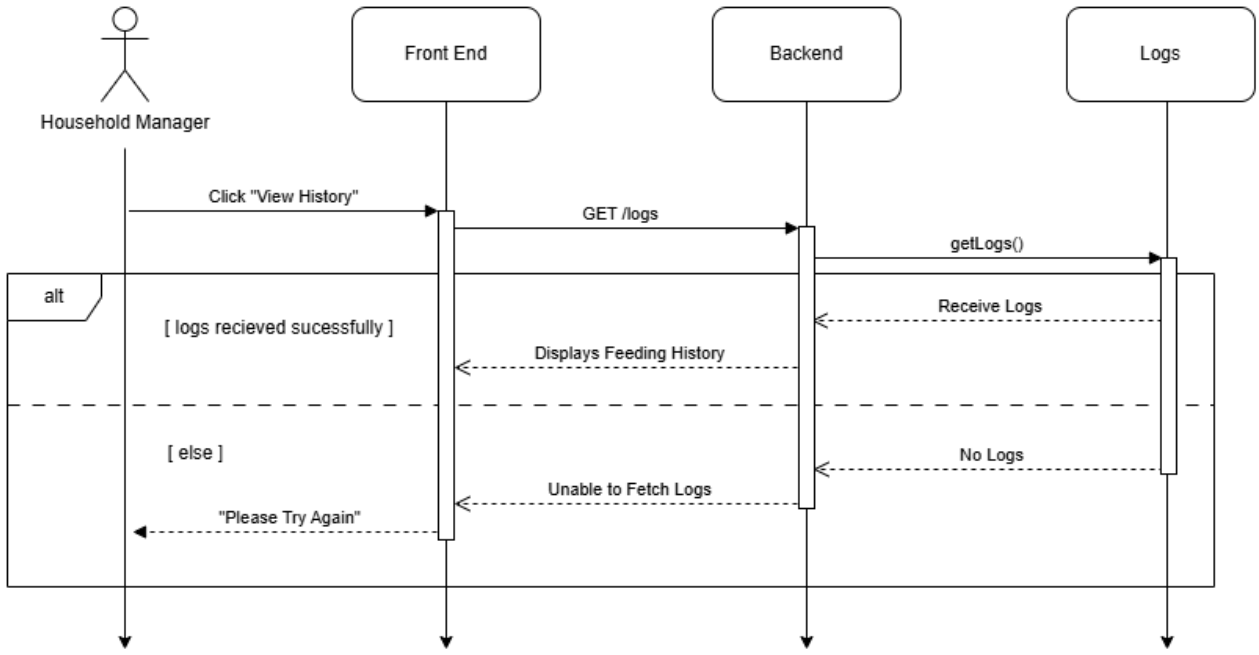


3. Manage Household

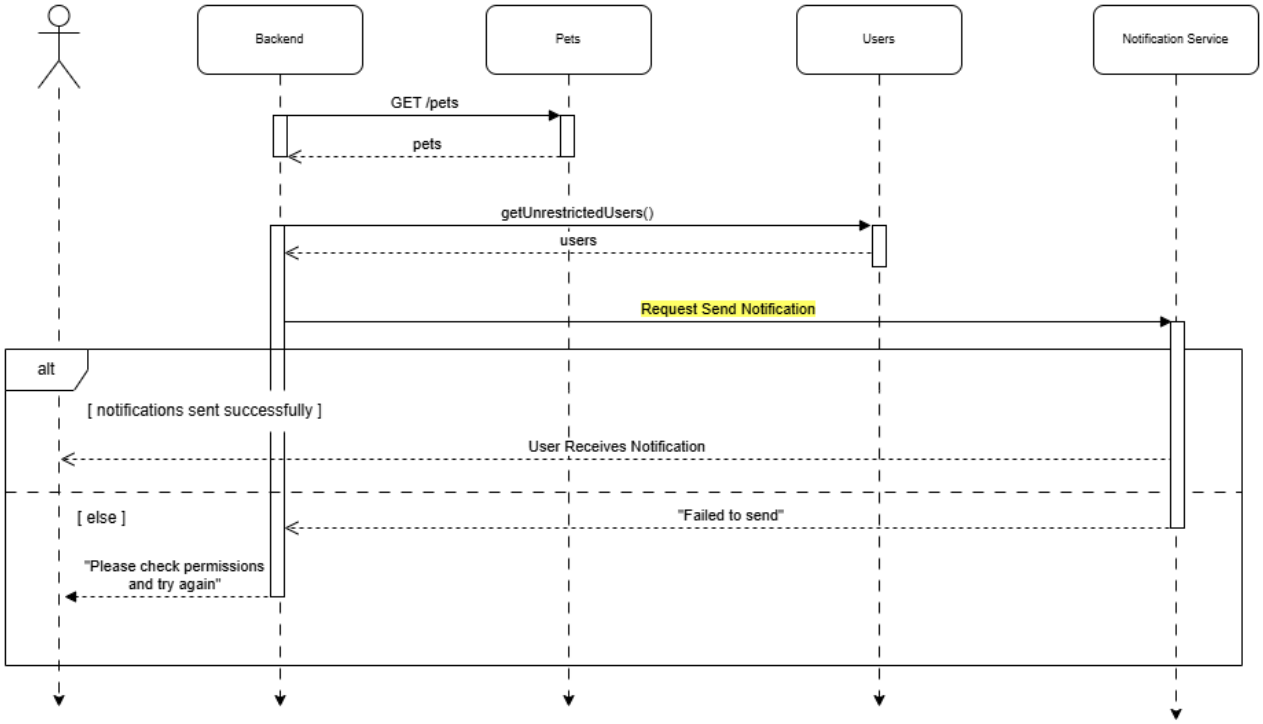




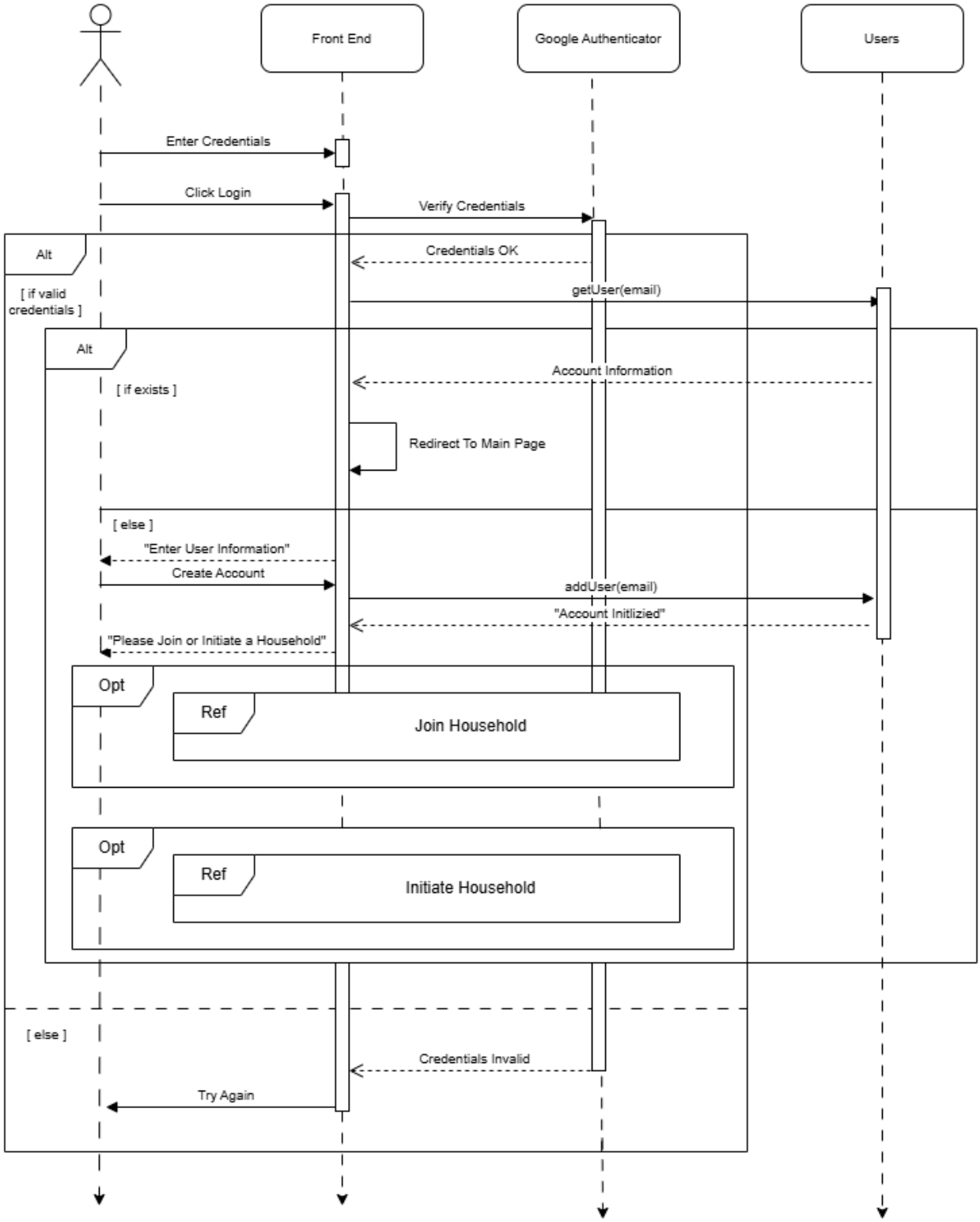
4. History Management



5. **Feeding Time Notification**



6. **Login Authentication**



4.7. Non-Functional Requirements Design

1. **Accessibility**
   - **Validation**: The app will support multiple languages, clear navigation, large readable fonts, high-contrast visuals, and intuitive icons to ensure accessibility for users with varying abilities. Key features will be easily accessible from the main screen or through shortcuts for a seamless experience.
2. **Maintainability**

- **Validation**: The app will store feeding logs in a scalable database with a retention policy that ensures data is kept for at least one year. Automated backups and secure storage will prevent data loss, while efficient indexing will allow users to quickly access and analyze historical feeding records.

4.8. Main Project Complexity Design

Feeding Data Analytics & Cost Prediction System

- **Description**:
  This system tracks pet feeding data, ranks users based on their feeding contributions, predicts monthly feeding costs, and detects anomalies in pet feeding patterns. The algorithm processes feeding logs, pet data, and food bag pricing to generate meaningful insights for users.
- **Why complex?**:
  - The algorithm involves **nested loops** to calculate **weekly rankings** of users based on feeding contributions.
  - Uses a **moving average** over multiple weeks to predict feeding costs.
  - Implements **anomaly detection** using **standard deviation**, identifying unusual feeding patterns.
  - Processes multiple data sources (feeding logs, pet profiles, food pricing) and performs multi-step computations.
- **Design**:
  - **Input**:
    - **Feeding Log Data**:
      - `Pet ID`
      - `User ID` (who fed the pet)
      - `Amount of food fed (grams)`
      - `Date and time of feeding`
    - **Pet Data**:
      - `Pet ID`
      - `Pet size (small, medium, large)`
    - **Food Bag Data**:
      - `Bag size (kg)`
      - `Price per bag`
  - **Output**:
    - **User rankings** based on weekly food contributions per pet.
    - **Total weekly food consumption** per pet.
    - **Predicted monthly feeding cost** based on past food consumption and bag prices.
    - **Anomalies** in pet feeding behavior, flagging sudden spikes or drops.
  - **Main Computational Logic**:
    1. **Compute weekly food consumption per pet** by summing feeding log data.
    2. **Rank users** based on their relative contribution to feeding each pet.
    3. **Predict feeding costs** using a **4-week moving average**, adjusting for food bag sizes and prices.

4. **Detect anomalies** by computing the **standard deviation** of feeding data and flagging outliers.

- **Pseudo-code**:

```
# Step 1: Compute total food consumption per pet per week
Initialize an empty dictionary: food_consumed_per_pet
For each feeding log:
    Extract pet_id, food_amount, and week_number from the log
    If pet_id is not in food_consumed_per_pet:
        Initialize an empty dictionary for this pet

    If week_number is not in food_consumed_per_pet[pet_id]:
        Set food_consumed_per_pet[pet_id][week_number] to 0

    Add food_amount to food_consumed_per_pet[pet_id][week_number]

# Step 2: Compute user rankings based on feeding contribution (Nested
Loop)
Initialize an empty dictionary: user_feed_ranking

For each pet_id in food_consumed_per_pet:
    For each week_number in food_consumed_per_pet[pet_id]:
        Retrieve total_food_consumed for this pet in this week
        For each feeding log:
            If log.pet_id matches pet_id and log's week_number matches:
                Extract user_id and food_amount
                If user_id is not in user_feed_ranking:
                    Initialize an empty dictionary for this user

                If pet_id is not in user_feed_ranking[user_id]:
                    Initialize an empty list for this pet

                Calculate contribution_percentage = (food_amount /
total_food_consumed) * 100
                Append (week_number, contribution_percentage) to
user_feed_ranking[user_id][pet_id]

# Step 3: Compute a moving average for cost predictions (last 4 weeks)
Initialize an empty dictionary: pet_costs

For each pet in pet_data:
    Extract pet_id and pet_size
    Retrieve bag_size and bag_price for this pet_size from
food_bag_data

    Retrieve last 4 weeks of food consumption data for this pet
    Compute avg_weekly_food as the average food consumption over the
last 4 weeks

    Compute bags_needed_per_month = (avg_weekly_food * 4) / (bag_size *
1000)
```

```
    Compute monthly_cost = bags_needed_per_month * bag_price
    Store monthly_cost in pet_costs[pet_id]

# Step 4: Anomaly Detection - Identify abnormal feeding behavior
Initialize an empty dictionary: anomalies

For each pet_id in food_consumed_per_pet:
    Compute average_food_consumption over all weeks
    Compute standard_deviation of food consumption

    For each week_number in food_consumed_per_pet[pet_id]:
        Retrieve total_food_consumed
        If total_food_consumed deviates more than 2 *
standard_deviation from the average:
            If pet_id is not in anomalies:
                Initialize an empty list for this pet
            Append week_number to anomalies[pet_id]

# Output results
Return user_feed_ranking, food_consumed_per_pet, pet_costs, anomalies
```

5. Contributions

- Tjammie Ko: Sequence Diagrams, Formatting, Other (14h)
- Dean McCarthy: Functional Requirements, Actors, Other (7h)
- Aidan Cotsakis: Presentation, Reflections, Other (8h)
- Matthew Fung: MD File, Functional Requirements, Dependency Diagram, Other (16h)