VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MULTI-DISCIPLINARY PROJECT

PROJECT REPORT

PROJECT : HAMSTER CARE

Lecturer: Trần Trương Tuấn Phát

Class: L08

Group : 1

No	Name	StudentID	Faculty
1	Chu Minh Tâm	2213009	CS
2	Đoàn Ngọc Hoàng Sơn	2212935	CS
3	Trần Quang Tác	2212962	CS
4	Thịnh Trần Khánh Linh	2211862	CS
5	Mai Hải Sơn	2212940	CE

Ho Chi Minh City, 2025

Contents

1 Introduction

The demand for pet ownership has been steadily increasing, encompassing not only traditional pets like dogs and cats but also a variety of newer, unconventional choices. Among these, hamsters have gained popularity due to their small size and adorable appearance. As a result, they are now widely available in pet stores, with some shops even specializing exclusively in hamsters.

However, raising hamsters comes with significant challenges. These small animals are highly susceptible to illness and require strict environmental conditions to thrive. Pet shop owners and breeders often invest heavily in specialized equipment to maintain optimal living conditions. Despite these efforts, much of the monitoring and care still rely on human intervention. Given the delicate nature of hamsters, even a minor oversight can lead to severe consequences, placing immense pressure on caretakers.

To address this issue, we propose the development of a smart system that automates the monitoring and regulation of key environmental factors such as temperature, humidity, and lighting. Additionally, this system will provide real-time data on habitat conditions and enable remote control of various aspects of hamster care, ensuring a healthier and more sustainable environment for these pets.

2 User Requirements

2.1 Functional Requirements

Environmental Monitoring & Adjustment

The system must utilize sensors to measure temperature, humidity, light levels, and water levels, then transmit the data to the backend.

- When data exceeds predefined thresholds, the system must automatically activate the appropriate devices:
  - If the temperature exceeds 30°C, the system must turn on a mini fan for cooling.
  - If the light level is insufficient, the system must turn on an LED light to ensure adequate illumination. If the light intensisty is too high system will automatically turn off the LED.

Water Supply System

- When the distance sensor detects a low water level, the system must automatically refill the water bottle water via the their phone.

Alerts and Remote Control

- When an issue arises (e.g., high temperature, insufficient light, low water level) or if status of devices being changed, the system will send real-time notifications to users' phone.
- Users must be able to remotely control devices, including turning on/off the fan, LED light, water pump via the application just by one click.
- The system must support two control modes:
  - **Automatic Mode**: Devices operands based on sensor data and automation rules set by users.
  - **Manual Mode**: Users can manually control devices when needed.

2.2 Non-Functional Requirements

Performance & Stability

- The system must be capable of processing data from multiple sensors simultaneously without delays or interruptions.
- The latency from receiving sensor data to displaying it on mobile applications must not exceed 5 seconds.
- Remote control actions must have a response time of no more than 5 seconds between command execution and device responses.
- The system must operate continuously and stably 24/7 with minimal downtime.
- The system must have mechanisms to detect and handle errors in case of sensor or device malfunctions, ensuring uninterrupted operation.

Usability & User Experience

- The system must have an intuitive, user-friendly interface that is easy to navigate.
- The system must provide comprehensive user guides and support for mobile applications.
- The application must be highly compaptible with Android devices.
- Page load time must be under 500ms for a smooth user experience.
- The application must support local data storage of up to 100MB and synchronize with the server.

Scalability & Future Expansion

- The system must be scalable to support additional sensor types and new devices in the future.
- The system must be able to expand to monitor and control multiple rearing areas without affecting overall performance.

3 Devices

3.1 Temperature & Humidity Sensor DHT20

- **Application**: Measures air temperature and humidity.
- **Input**: Surrounding air.
- **Output**: Temperature and humidity values.

DHT20

3.2 Distance Sensor

- **Application**: Measures distance to objects using ultrasonic waves.
- **Input**: Ultrasonic waves reflected from objects.
- **Output**: Distance value.

Distance Sensor

3.3 Light Sensor

- **Application**: Measures environmental light intensity.
- **Input**: Surrounding light.
- **Output**: Light intensity value.

Light Sensor

3.4 Mini Fan

- **Application**: Provides cooling or airflow control.
- **Input**: On/off control signal from the circuit.
- **Output**: Airflow.

Mini Fan

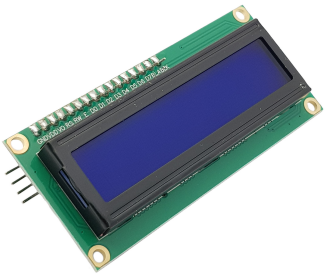
3.5 4 RGB LED Light

- **Application**: Displays colors based on control input.
- **Input**: Color control signals from the main board.
- **Output**: Colored light display.

4 RGB LED Light

3.6 LCD 16x2

- **Application**: Displays information about the environment.
- **Input**: Sensor signals.
- **Output**: Displays information about temperature, humidity, light and water level



LCD 16x2

3.7 Pump

- **Application**: Water pump.
- **Input**: Control signal from user.
- **Output**: Water pump.

Pump

3.8 IR

- **Application**: Read infrared signal from remote.
- **Input**: Infrared signal from remote.
- **Output**: Infrared value.

IR

4 Usecase Diagram

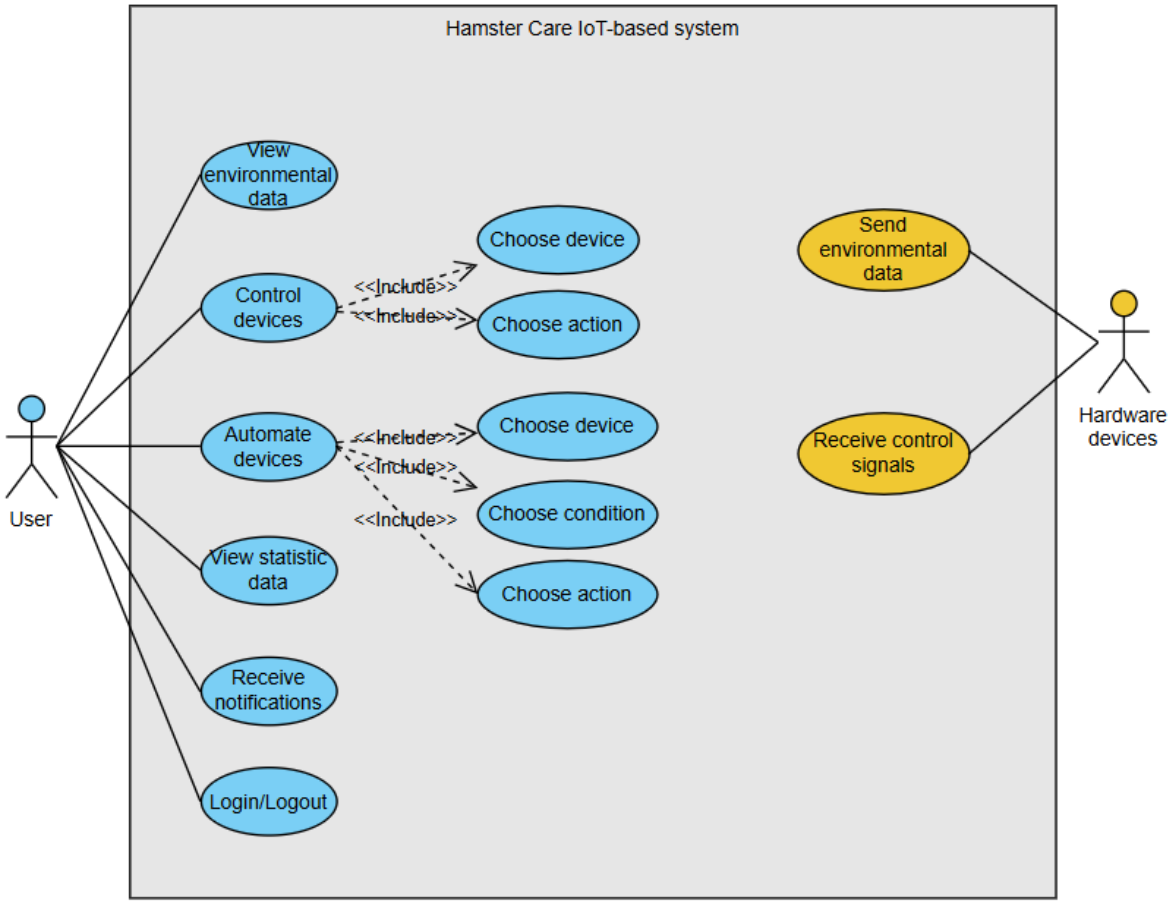


Figure 1: Use-case Diagram

5 Usecase Details

5.1 View Status Parameters

Use-case ID	1		
Use-case name	View Status Parameters		
Created by:	Trần Quang Tác	**Last updated by:**	Trần Quang Tác
Date created:	24/02/2025	**Date last updated:**	25/02/2025
Actors:	User		
Description:	User can view environmental parameters of each cage, including temperature, humidity, light level, water level, etc.		
Preconditions:	The user must have logged into the system.		
Postconditions:	1. The user can view the status parameters of the selected cage. 2. If any parameter exceeds the threshold, the system will alert the user.		
Normal flows:	1. User selects a cage. 2. The system sends a request to retrieve data from the database. 3. The system displays a list of environmental parameters. 4. If any parameter exceeds the safe limit, the system displays status alerts.		
Exceptions:	1. If the connection to the database fails or data is unavailable, the system displays an error message. 2. If any parameter exceeds the safe threshold, the system sends an alert.		
Note and issues:			

Table 1: Use Case Specification for Viewing Status Parameters

5.2 Control Devices

Use-case ID	2		
Use-case name	Control Devices		
Created by:	Trần Quang Tác	**Last updated by:**	Trần Quang Tác
Date created:	24/02/2025	**Date last updated:**	25/02/2025
Actors:	User		
Description:	The user can control devices in the cage by turning them on/off or refill.		
Preconditions:	1. The user must be logged into the system. 2. Devices must be available and connected to the system.		
Postconditions:	1. The device receives and executes the control command from the user. 2. The system updates the new device status after executing the command.		
Normal flows:	1. The user selects a cage. 2. The system displays a list of available devices in the cage. 3. The user selects the device to control. 4. The user selects the action to perform (turn on/off/refill). 5. The system sends the control signal to the device. 6. The system verifies and updates the new device status.		
Exceptions:	If the device does not respond, the system displays an error message.		
Note and issues:			

Table 2: Use Case Specification for Controlling Devices

5.3 Automate Devices

Use-case ID	3		
Use-case name	Automate Devices		
Created by:	Trần Quang Tác	**Last updated by:**	Trần Quang Tác
Date created:	24/02/2025	**Date last updated:**	25/02/2025
Actors:	User		
Description:	The user can set up automation rules for devices to operate based on predefined conditions.		
Preconditions:	1. The user must be logged into the system. 2. Devices must support automation features.		
Postconditions:	1. Automation rules are stored in the system. 2. When conditions are met, devices execute actions automatically without user intervention.		
Normal flows:	1. The user selects a cage. 2. The system displays a list of devices. 3. The user selects a device to automate. 4. The user switches the device to Auto mode. 5. The system verifies and activates automation rules (if any). 6. The system updates the device status in the system.		
Alternative flows:	4a. Instead of just switching to Auto mode, the user chooses to configure automation rules. 4a.1. The system displays a rule setup pop-up with two options: Condition or Schedule. 4a.2. Condition: - When an environmental parameter (temperature, humidity, light level, water level, food level, etc.) meets a certain condition (>, <, =, etc.), the system executes an action (turn on, turn off, refill, etc.). 4a.3. Schedule: - At a specific time (e.g., 17:00), the system executes an action (turn on, turn off, refill, etc.). - The user can select the days of the week to apply the schedule (Sunday → Saturday). 4a.4. The user clicks "Save" to store the new rule. 4a.5. The system updates the rule list and automatically activates them when conditions are met.		
Exceptions:	If multiple conflicting rules exist, the system prioritizes them or notifies the user for adjustments.		
Note and issues:			

Table 3: Use Case Specification for Automating Devices

5.4 View data analysis

Use-case ID	4		
Use-case name	View data analysis		
Created by:	Mai Hải Sơn	**Last updated by:**	Mai Hải Sơn
Date created:	25/02/2025	**Date last updated:**	26/02/2025
Actors:	User		
Description:	The user can follow the environmental data analysis		
Preconditions:	The user must be logged into the system.		
Postcondition:	Analyzed data will be displayed on charts.		
Normal flows:	1. The user press view data analysis. 2. The system retrieve data from database. 3. Get data from database, analyze and draw chart. 4. Data analysis display in mobile.		
Exceptions:	If the device does not respond, the system displays an error message.		
Note and issuses:			

Table 4: Use Case Specification for View data analysis

5.5 Send environmental data

Use-case ID	5		
Use-case name	Send envirnmental data		
Created by	Mai Hải Sơn	**Last updated by:**	Mai Hải Sơn
Date created:	25/02/2025	**Date last updated:**	26/02/2025
Actors	Hardware devices		
Description:	Hardware devices receive signal from sensors and send them to server		
Preconditions:	1.Sensors must connect successfully to the microbit. 2.The microbit must connect successfully to the server.		
Postconditions:	The data must be sent successfully to server.		
Normal flows:	1. Sensors get signal from environment and send them to the microbit. 2. The microbit receives signal from sensors and send them to the server. 3. The server receives and send them to the database and the mobile.		
Exceptions:	If the device does not respond, the system displays an error message.		
Note and issues:			

Table 5: Use Case Specification for Send environmental data

5.6 Receive control signals

Use-case ID	6		
Use-case name	SReceive control signals		
Created by	Mai Hải Sơn	**Last updated by:**	Mai Hải Sơn
Date created:	25/02/2025	**Date last updated:**	26/02/2025
Actors	Hardware devices, user		
Description:	Hardware devices receive signal from server and control devices		
Preconditions:	1.Sensors must connect successfully to the microbit. 2.The microbit must connect successfully to the server. 3.If it is manual signal, user must log in to system.		
Postconditions:	Hardware must be controlled as require		
Normal flows:	1. Sensors get signal from environment and send them to the microbit. 2. The microbit receives signal from sensors and send them to the server. 3. If scheduled signal is trigged, send control signal to microbit. 4. If the condition meets the rule, a control signal will be sent to the microbit. 5. Microbit receive control signal and control devices.		
Exceptions:	If the device does not respond, the system displays an error message.		
Note and issues:			

Table 6: Use Case Specification for Receive control signals

6 Activity Diagrams

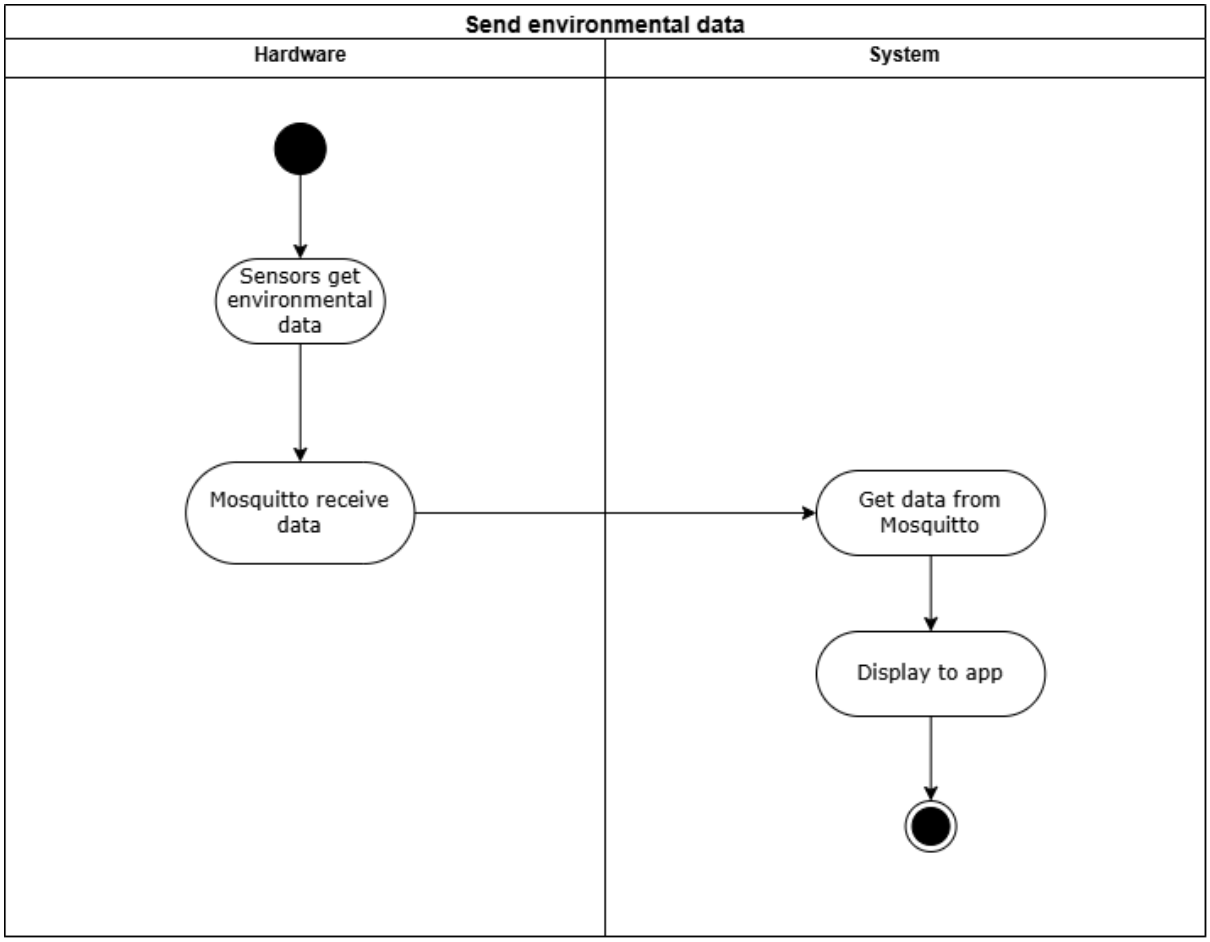


Figure 2: Activity diagram for Sending sensor data from device to app

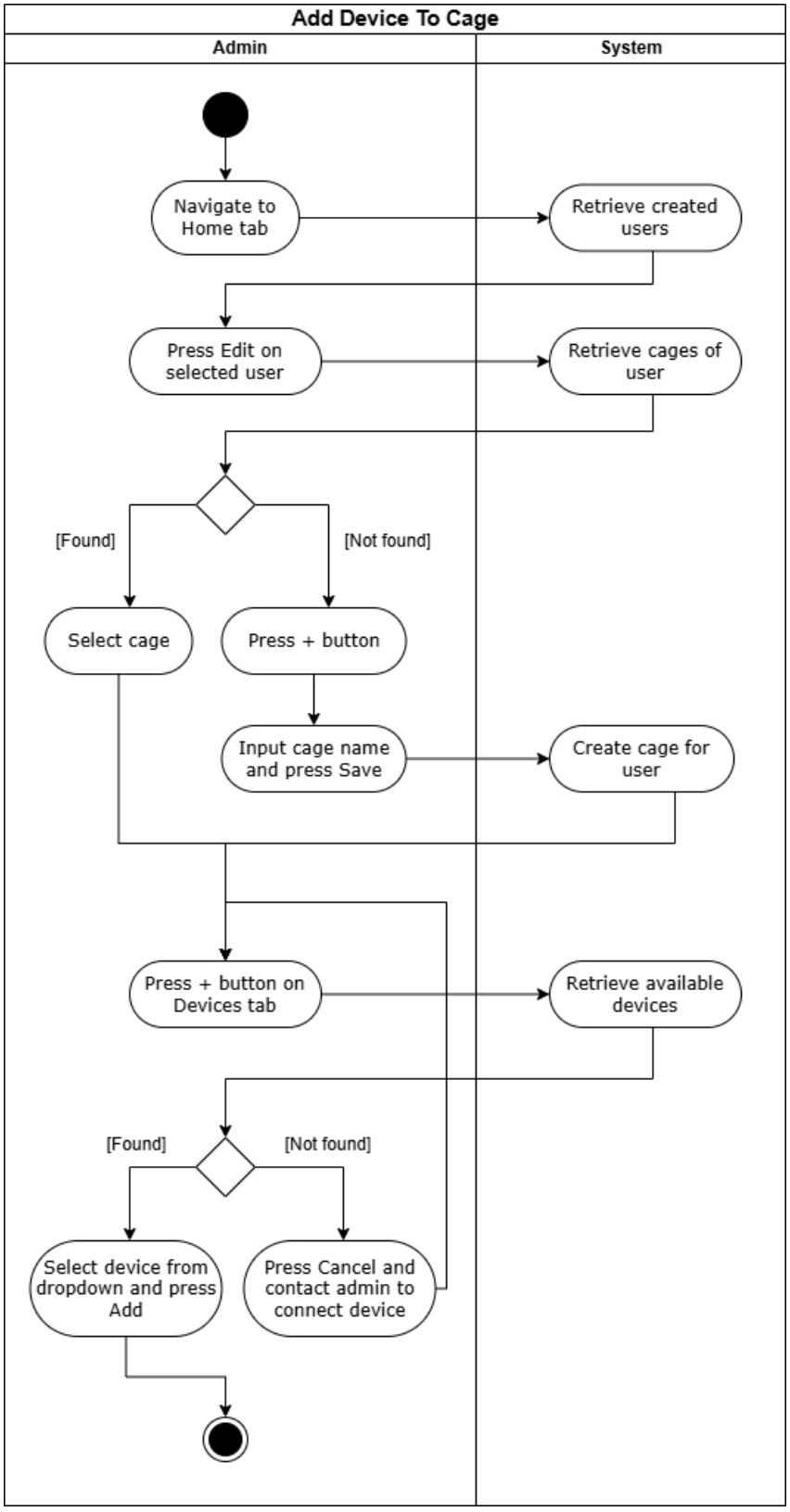


Figure 3: Activity diagram for Adding devices to a cage

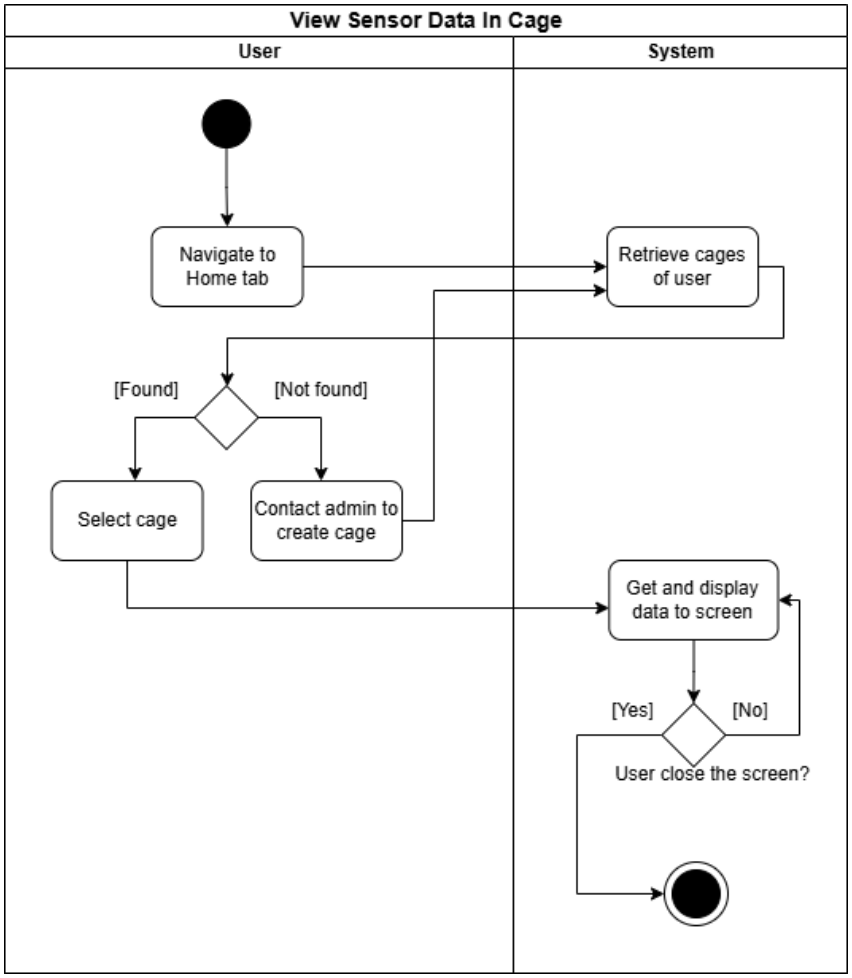


Figure 4: Activity diagram for Viewing sensor data in a cage

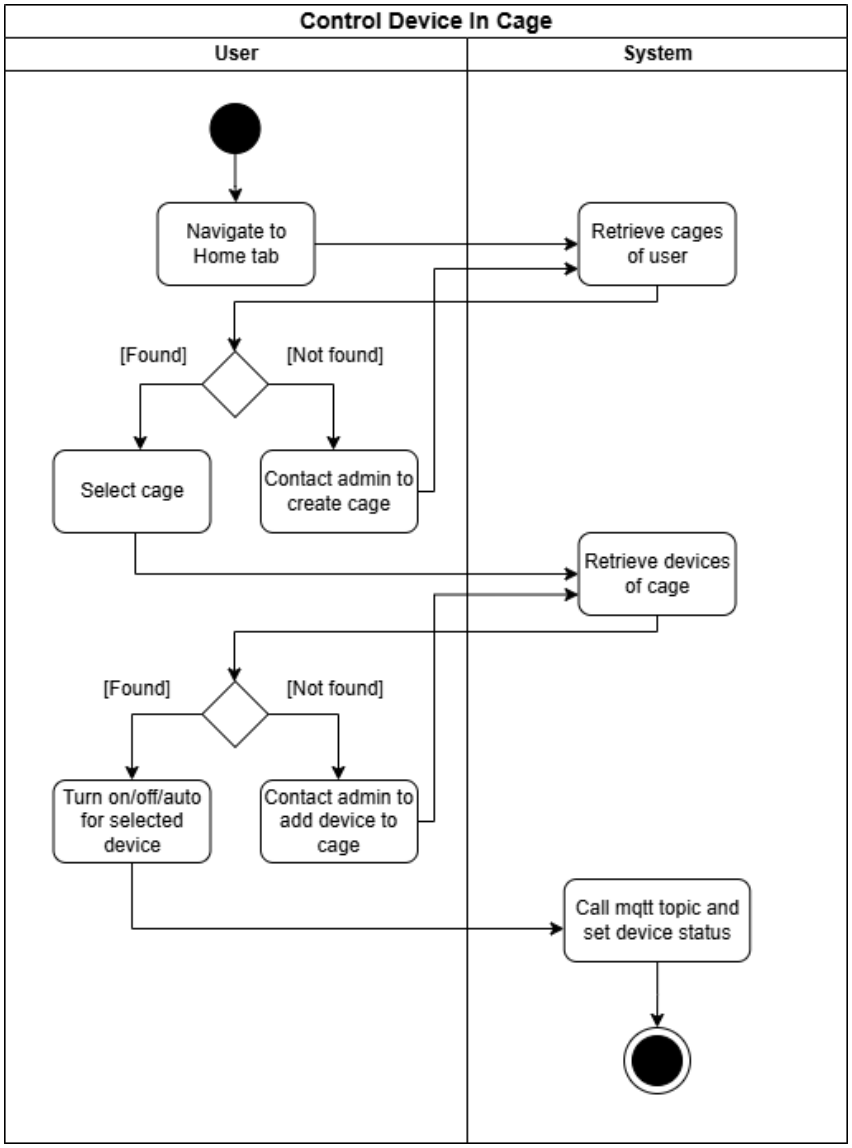


Figure 5: Activity diagram for Controlling devices in a cage

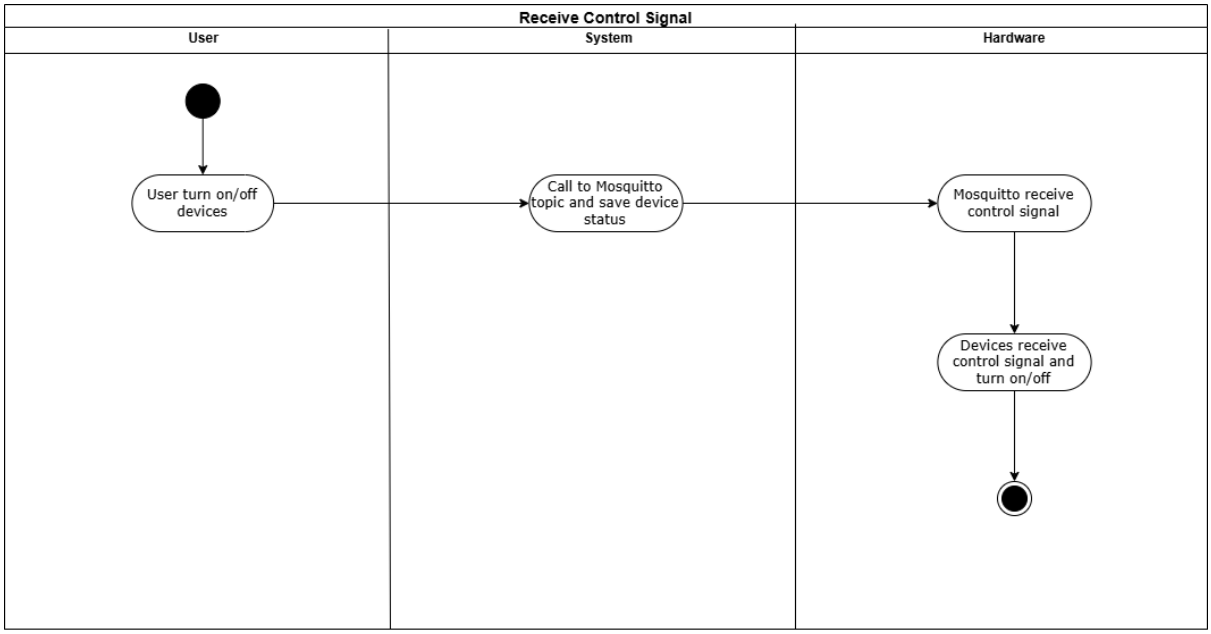


Figure 6: Activity diagram for Receiving control signal from app

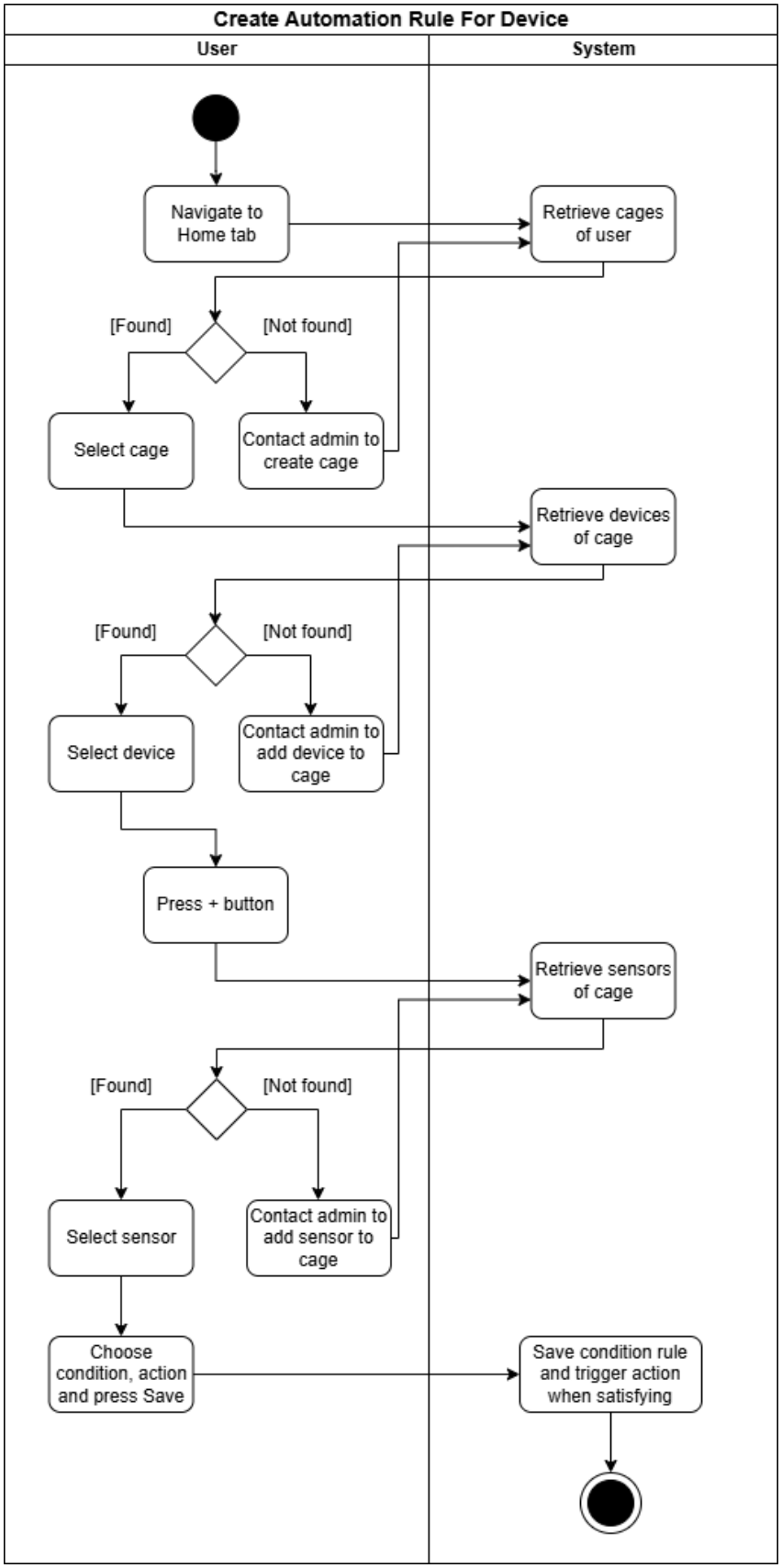


Figure 7: Activity diagram for Creating automation rules for a cage

7 Technologies

Frontend

- **Programming Language: Dart/Flutter**
  - The frontend of the HamsterCare system is developed using Flutter
- **Reasons choosing flutter:**
  - Cross-platform Development: One codebase that can run on both iOS and Android
  - Hot Reload: Enables rapid development and real-time view of changes
  - Rich Widget Library: Material Design and Cupertino widgets for native look and feel
  - Reactive Framework: Built-in state management capabilities
- **Key Libraries Used (from pubspec.yaml analysis):**
  - http: For REST API communication with the Go backend
  - flutter_secure_storage: Secure storage for sensitive data like JWT tokens
  - web_socket_channel: Real-time communication with backend

Backend

- **Programming Language : Golang**
  - The backend of the HamsterCare system is developed in Go (Golang), chosen for its efficient concurrency, fast compilation, and built-in support for parallel processing. Go is ideal for real-time communication and large-scale data handling, making it well-suited for managing IoT devices in the HamsterCare platform.
- **Reasons for Choosing Go**
  - **Performance:** Go compiles to machine code, offering performance similar to C/C++ with easier maintenance and readability.
  - **Concurrency:** Go's `goroutines` and `channels` handle parallel processing effectively, supporting scalable real-time data exchange.
  - **Simplicity:** Go's simple syntax reduces development time and errors, making it easier to maintain.
  - **Scalability:** Go's performance and single binary deployment make it an ideal choice for scalable, easy-to-maintain systems.
- **Libraries Used in the Project**

- **gin-gonic/gin:** A high-performance web framework used to build fast and efficient RESTful APIs, which form the core of communication between the backend and frontend systems.
- **eclipse/paho.mqtt.golang:** A library that supports communication with IoT devices via the MQTT protocol, allowing for real-time data exchange between devices and the HamsterCare platform.
- **database/sql, github.com/lib/pq:** These libraries provide robust support for connecting to and interacting with a PostgreSQL database, allowing HamsterCare to manage data storage efficiently and reliably.
- **github.com/golang-migrate/migrate/v4:** A powerful tool for managing database migrations, ensuring that the HamsterCare platform's database schema is always up to date and in sync with the latest changes in the application.
- **golang-jwt/jwt/v5:** A library that enables the use of JSON Web Tokens (JWT) for user authentication, helping to secure API endpoints and protect sensitive data.
- **sendgrid/sendgrid-go:** A Go client library for SendGrid, which is used to send email notifications to users, ensuring they are kept informed about important system events.
- **joho/godotenv:** A library that helps manage environment variables by loading them from a `.env` file, facilitating configuration management and securely handling sensitive information, such as database credentials and API keys.

Database

In this project, **PostgreSQL** was selected as the primary database management system due to its strong compliance with SQL standards, high performance, and excellent scalability. PostgreSQL is an open-source relational database system that supports complex data storage, querying, and management needs, offering modern features such as `JSONB` types, arrays, and full-text search.

Reasons for Selection:

- **Standards Compliance:** Full support for SQL standards ensures seamless integration with other systems and tools.
- **High Performance:** Advanced query optimization and diverse indexing techniques (e.g., B-tree, GiST, GIN) enable efficient handling of large datasets.
- **Advanced Features:** Native support for semi-structured data, full-text search, and complex data types enhances flexibility.
- **Reliability and Security:** Robust backup and recovery mechanisms, including Point-in-Time Recovery (PITR), and fine-grained access control enhance system robustness.
- **Open Source:** No licensing costs, supported by a strong community and extensive documentation.

The database schema is designed following normalization principles to minimize redundancy and ensure data consistency through primary keys, foreign keys, and `UNIQUE` constraints. Relational tables store information related to users, transactions, and system logs. Query performance is optimized by applying indexes to frequently accessed columns and analyzing queries with the `EXPLAIN` command. Tools such as `pgAdmin` are employed to monitor database performance and address potential issues proactively.

Other Technologies

MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for low-bandwidth, high-latency, or unreliable networks. It follows a publish-subscribe model where devices (clients) communicate through a central broker, making it highly efficient for IoT (Internet of Things) applications. In this project, MQTT is used to facilitate real-time communication between devices (sensors, actuators) and the server.

Postman

Postman is a popular API testing tool that allows developers to easily send requests to a web service and examine the responses. It supports multiple HTTP methods such as GET, POST, PUT, DELETE, and can also handle authentication, headers, and query parameters. In this project, Postman is used for testing the API endpoints and ensuring proper communication between the frontend, backend, and database. By simulating real-world client-server interactions, Postman helps validate the functionality and performance of the backend system before deployment.

Git Flow

Git flow is a popular branching model that structures how features, releases and hotfixes are handled.
- **main** : The stable of version of the code, which always reflects production-ready code.
- **develop** : The integration branch whre features are merged before they are released.
- **feature** : These are created from the **develop** branch for working on new feature. Once feature is complete, it is merge back in to **develop**.
- **hotfix** : These are used to quickly address issues or bugs in production. Hotfixes are typically created from **main** and merged back into both **main** and **develop** to keep both branches up-to-date.

Docker

- Consistency Across Environments: Docker ensures the app runs the same way on every developer's machine and in all deployment environments.

- Faster Setup: Easily set up the project by simply running a Docker container, speeding up onboarding and making it easy to replicate environments.

- Scalable Deployments: Docker containers can be quickly scaled across different environments, making it easier to deploy to production or test environments. This allows for faster iteration and scaling as the project grows.

- Simplified Dependency Management: Docker encapsulates all dependencies, saving time and reducing complexity.

8 Database Schema

8.1 The Entity-Relationship Diagram (ERD) of Hamster Care

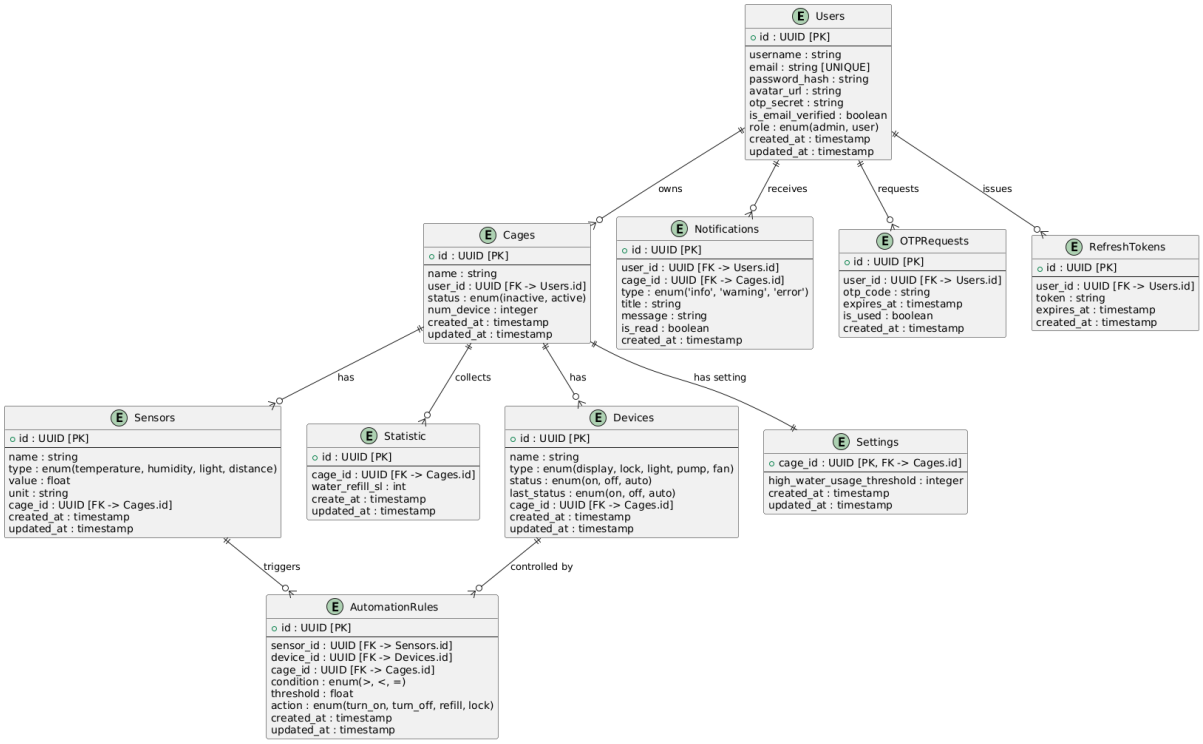


Figure 8: ERD Diagram of Hamster Care

8.2 Tables and Relationships

- Users Table
  - Stores user account information, authentication details, and user roles.
  - Fields:
    - id (PK): A unique identifier for each user.
    - username: The user's login name.
    - email: The user's email address
    - avatar_url: URL to the user's profile picture.
    - password_hash: Hashed version of the user's password for secure authentication.
    - otp_secret: Secret key for generating OTP codes (used in two-factor authentication).
    - is_email_vertified: Indicates whether the user's email has been verified.
    - role: The user's role (admin, user)
    - created_at: Timestamp when the user account was created.
    - update_at: Timestamp when the user account was last updated.

```sql
-- Create table users with UNIQUE constraint on username
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    username VARCHAR(255) NOT NULL UNIQUE, -- UNIQUE constraint applied directly
    email VARCHAR(255) UNIQUE NOT NULL,
    avatar_url TEXT DEFAULT '',
    password_hash TEXT NOT NULL,
    otp_secret TEXT, -- Store secret for OTP generation
    is_email_verified BOOLEAN DEFAULT FALSE,
    role VARCHAR(10) CHECK (role IN ('admin', 'user')) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 9: SQL CREATE TABLE statement for the Users table.

- Cages Table
  - Stores information about cages managed by users.
  - Fields:
    - id (PK): A unique identifier (UUID) for each cage.
    - name: The name of the cage.
    - user_id (FK → Users.id): The ID of the user who owns the cage.
    - status: The operational status of the cage (inactive or active).
    - num_device: The number of devices associated with the cage.
    - created_at: Timestamp when the cage record was created.
    - updated_at: Timestamp when the cage record was last updated.

```sql
DO $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'cage_status') THEN
        CREATE TYPE cage_status AS ENUM ('on', 'off');
    END IF;
END $$;

CREATE TABLE cages (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    user_id UUID NOT NULL,
    status cage_status DEFAULT 'off' NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

Figure 10: SQL CREATE TABLE statement for the Cages table.

- Sensors Table
  - Stores information about sensors.

  - Fields:
    * id (PK): A unique identifier (UUID) for each sensor.
    * name: The name of the sensor.
    * type: The type of sensor (temperature, humidity, light, or distance).
    * value: The latest recorded value from the sensor.
    * unit: The measurement unit for the sensor's value (°C,
    * cage_id (FK → Cages.id): The ID of the cage where the sensor is located.
    * created_at: Timestamp when the sensor record was created.
    * updated_at: Timestamp when the sensor record was last updated.

```sql
DO $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'sensor_type') THEN
        CREATE TYPE sensor_type AS ENUM ('temperature', 'humidity', 'light', 'distance');
    END IF;
END $$;


CREATE TABLE sensors (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    type sensor_type NOT NULL,
    value FLOAT,
    unit VARCHAR(50),
    cage_id UUID,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (cage_id) REFERENCES cages(id) ON DELETE CASCADE
);
```

Figure 11: SQL CREATE TABLE statement for the Sensors table.

- Devices Table
  - Stores information about devices
  - Fields:
    * id (PK): A unique identifier (UUID) for each device.
    * name: The name of the device.
    * type: The type of device (display, lock, light, pump, fan).
    * status: The current operating status of the device (on, off, auto).
    * last_status: The previous status before the current one.
    * cage_id (FK → Cages.id): The ID of the cage where the sensor is located.
    * created_at: Timestamp when the device record was created.
    * updated_at: Timestamp when the device record was last updated.

```sql
DO $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'device_type') THEN
        CREATE TYPE device_type AS ENUM ('display', 'lock', 'light', 'pump', 'fan');
    END IF;
    IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'device_status') THEN
        CREATE TYPE device_status AS ENUM ('on', 'off', 'auto');
    END IF;
END $$;


CREATE TABLE devices (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL UNIQUE,
    type device_type NOT NULL,
    status device_status DEFAULT 'off' NOT NULL,
    last_status device_status,
    cage_id UUID,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (cage_id) REFERENCES cages(id) ON DELETE CASCADE
);
```

Figure 12: SQL CREATE TABLE statement for the Devices table.

- AutomationRules Table
  - Stores automation rules that trigger actions on devices based on sensor values.
  - Fields:
    - id (PK): A unique identifier (UUID) for each automation rule.
    - sensor_id (FK → Sensors.id): The sensor associated with the rule.
    - device_id (FK → Devices.id): The device controlled by the rule.
    - cage_id (FK → Cages.id): The cage where the automation rule is applied.
    - condition: The condition to evaluate (>, <, or =).
    - threshold: The threshold value to trigger the action.
    - action: The action to perform (turn_on, turn_off, refill, or lock).
    - created_at: Timestamp when the rule was created.
    - updated_at: Timestamp when the rule was last updated.

```sql
DO $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'condition_enum') THEN
        CREATE TYPE condition_enum AS ENUM ('>', '<', '=');
    END IF;
    IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'action_enum') THEN
        CREATE TYPE action_enum AS ENUM ('turn_on', 'turn_off', 'refill', 'lock');
    END IF;
END $$;

CREATE TABLE automation_rules (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    sensor_id UUID NOT NULL,
    device_id UUID NOT NULL,
    cage_id UUID NOT NULL,
    condition condition_enum NOT NULL,
    threshold FLOAT NOT NULL,
    action action_enum NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (sensor_id) REFERENCES sensors(id) ON DELETE CASCADE,
    FOREIGN KEY (device_id) REFERENCES devices(id) ON DELETE CASCADE
    FOREIGN KEY (cage_id) REFERENCES cages(id) ON DELETE CASCADE
);
```

Figure 13: SQL CREATE TABLE statement for the AutomationRules table.

- Notifications Table
  - Stores notifications sent to users regarding events related to their cages.
  - Fields:
    - id (PK): A unique identifier (UUID) for each notification.
    - user_id (FK → Users.id): The user who receives the notification.
    - cage_id (FK → Cages.id): The cage related to the notification.
    - type: The type of notification (info, warning, error).
    - title: A short title summarizing the notification.
    - message: The detailed message content of the notification.
    - is_read: A boolean indicating whether the notification has been read.
    - created_at: Timestamp when the notification was created.

```sql
CREATE TABLE notifications (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL,
    cage_id UUID NOT NULL,
    type VARCHAR(50) NOT NULL,
    title VARCHAR(255) NOT NULL,
    message TEXT NOT NULL,
    is_read BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT notifications_type_check
        CHECK (type IN ('info', 'warning', 'error'))
);
```

Figure 14: SQL CREATE TABLE statement for the Notifications table.

- Statistic Table
  - Stores statistical data related to each cage, primarily tracking water refill counts.
  - Fields:
    - id (PK): A unique identifier (UUID) for each statistic record.
    - cage_id (FK → Cages.id): The cage associated with the statistic.
    - water_refill_sl: The number of times water has been refilled.
    - create_at: Timestamp when the record was created.
    - updated_at: Timestamp when the record was last updated.

```sql
CREATE TABLE statistic (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    cage_id UUID NOT NULL,
    water_refill_sl INT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (cage_id) REFERENCES cages(id) ON DELETE CASCADE
);
```

Figure 15: SQL CREATE TABLE statement for the Statistic table.

- OTPRequests Table
  - Stores One-Time Password (OTP) requests for user authentication or verification processes.
  - Fields:
    - id (PK): A unique identifier (UUID) for each OTP request.
    - user_id (FK → Users.id): The user associated with the OTP request.
    - otp_code: The generated OTP code (usually 6 characters).
    - expires_at: The expiration timestamp for the OTP.
    - is_used: A boolean indicating whether the OTP has been used.
    - created_at: Timestamp when the OTP request was created.

```sql
-- Create table otp_request
CREATE TABLE otp_request (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    otp_code VARCHAR(6) NOT NULL,
    expires_at TIMESTAMP NOT NULL,
    is_used BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 16: SQL CREATE TABLE statement for the OTPRequests table.

- RefreshTokens Table
  - Stores refresh tokens used for maintaining user sessions and enabling token renewal without re-login.
  - Fields:
    - id (PK): A unique identifier (UUID) for each refresh token.
    - user_id (FK → Users.id): The user associated with the refresh token.
    - token: The actual refresh token string.
    - expires_at: The expiration timestamp of the refresh token.
    - created_at: Timestamp when the refresh token was created.

```sql
-- Refresh token table
CREATE TABLE refresh_tokens (
    id UUID PRIMARY key DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    token TEXT NOT NULL,
    expires_at TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 17: SQL CREATE TABLE statement for the RefreshTokens table.

- Settings Table
  - Stores configuration settings for each cage, such as water usage thresholds.
  - Fields:
    - cage_id (PK, FK → Cages.id): The unique identifier of the cage that this setting is associated with. It is also a foreign key referring to the Cages table.
    - high_water_usage_threshold: The threshold value for high water usage, beyond which notifications or actions may be triggered.
    - created_at: Timestamp when the setting was created.
    - updated_at: Timestamp when the setting was last updated.

```sql
CREATE TABLE settings (
    cage_id UUID PRIMARY KEY,
    high_water_usage_threshold INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 18: SQL CREATE TABLE statement for the Settings table.

- Relationships between Tables
  - **Users** (U) **owns Cages** (C): one user can own multiple cages.
  - **Cages** (C) **has Sensors** (S): one cage can have multiple sensors.
  - **Cages** (C) **has Devices** (D): one cage can have multiple devices.
  - **Cages** (C) **collects Statistics** (STT): one cage can have multiple statistics records.
  - **Sensors** (S) **triggers AutomationRules** (AR): a sensor can trigger multiple automation rules.
  - **Devices** (D) **controlled by AutomationRules** (AR): a device can be controlled by multiple automation rules.
  - **Users** (U) **receives Notifications** (N): one user can receive multiple notifications.
  - **Users** (U) **requests OTPRequests** (O): one user can have multiple OTP requests.
  - **Users** (U) **issues RefreshTokens** (RT): one user can have multiple refresh tokens.
  - **Cages** (C) **has setting Settings** (SET): each cage has one settings record.

9 System Architecture

9.1 Architecture diagram

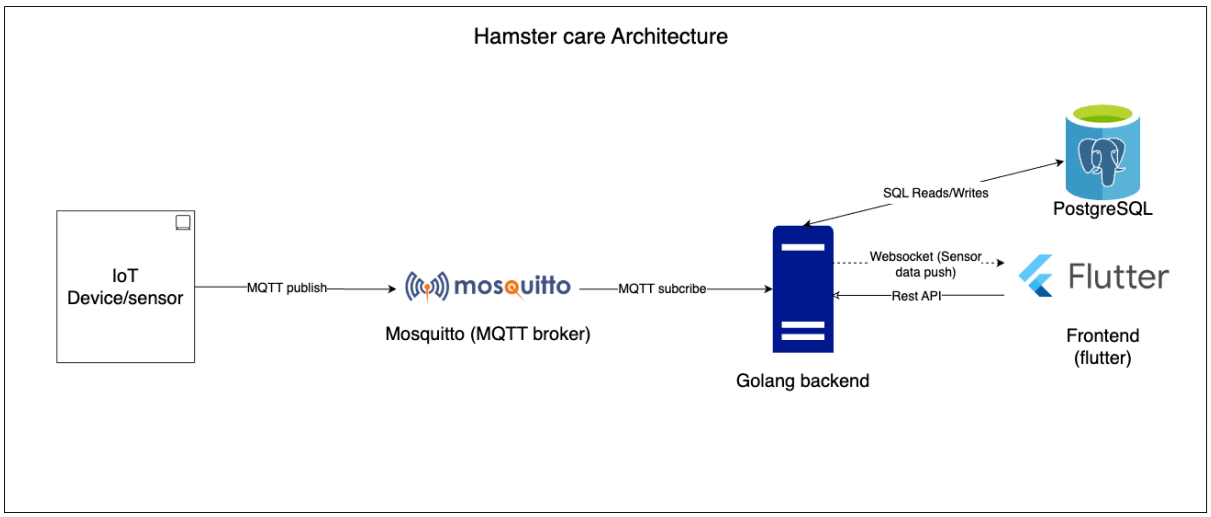


Figure 19: Architecture diagram

This architecture is designed to monitor and manage hamster environmental conditions using IoT sensors, a lightweight backend, and a mobile-friendly frontend. The system has four major components:

- IoT Device/Sensor
  - Devices (such as temperature, humidity, or motion sensors) are deployed near hamster habitats.
  - These devices publish sensor data using the MQTT protocol to a central message broker.
- Mosquitto (MQTT Broker)
  - It forwards (via MQTT subscribe) the messages to the backend service for further processing.
- Golang Backend
  - Data Persistence: Reading and writing data to PostgreSQL database.
  - Real-time Communication: Using WebSocket to push live sensor data to the Flutter frontend.
  - API Services: Exposing REST APIs to the frontend for user interaction and historical data retrieval.
- Frontend (Flutter Application)
  - WebSocket for receiving live sensor updates.
  - REST API for regular data queries, user management, and historical monitoring.
- PostgreSQL Database
  - The backend performs read/write operations to this database as needed.

Frontend Architecture

```
lib/
|-- services/
|   |-- apis.dart
|   |-- permission.dart
|   `-- session.dart
|
|-- models/
|   |-- cage.dart
|   |-- cagedetail.dart
|   |-- cageinit.dart
|   |-- chartdata.dart
|   |-- device.dart
|   |-- noti.dart
|   |-- rule.dart
|   |-- sensor.dart
|   `-- user.dart
|
|-- views/
|   |-- admin/
|   |   |-- admin_cage.dart
|   |   |-- admin_home.dart
|   |   |-- admin_new_user.dart
|   |   |-- admin_profile.dart
|   |   `-- admin_view_user.dart
|   |
```

```
|   `-- user/
|       |-- user_cage.dart
|       |-- user_device.dart
|       |-- user_home.dart
|       |-- user_notification.dart
|       |-- user_profile.dart
|       `-- user_statistic.dart
|
|-- admin_app.dart
|-- constants.dart
|-- forgot_password.dart
|-- login.dart
|-- user_app.dart
|-- utils.dart
`-- main.dart
```

The frontend adopts a clean, modular structure to ensure scalability, maintainability, and clear separation of concerns. The organization of the source code is as follows:

9.2 Services

This directory contains the service layer classes responsible for managing interactions between the Flutter frontend and the Golang backend APIs, as well as handling device-level operations:

- *apis.dart*: defines methods for communicating with the backend server through RESTful API or WebSocket endpoints.
- *permission.dart*: manages camera and image storage permissions required for the application's operation.
- *session.dart*: handles user authentication tokens (JWT), session persistence, and secure storage.

9.3 Models

This folder holds all the data models representing the entities used throughout the app:

- *cage.dart*, *cagedetail.dart*, *cageinit.dart*: models for cage-related data structures.
- *chartdata.dart*: keeps data needed for displaying water consumption chart and statistic information.
- *device.dart*, *noti.dart*, *rule.dart*, *sensor.data*: keeps data needed for displaying devices, notifications, automation rules and sensors information.
- *user.dart*: keeps user profile information.

Models typically map JSON data from the backend into Dart objects and from objects to JSON payloads sent to API endpoints.

9.4 Views

The views folder separates user interface (UI) code based on user roles:

- Admin views (*views/admin/*): screens related to administration panels, such as managing user account and their cage, adding devices and sensor to a cage.
- User views (*view/user/*: screens for regular users to manage their cages, viewing sensor data and control devices in each cage, view statistic information, receive notifications and edit profile.

Each screen is isolated in its file, following the single-responsibility principle for better readability and maintenance.

9.5 Other files

Files directly under *lib/* define the overall application setup:

- *admin_app.dart*, *user_app.dart*: the display entry points, separating user and admin flows.
- *constants.dart*: centralized file for static values such as theme colors, font sizes, height, base URL, ensuring a consistent UI.
- *forgot_password.dart*, *login.dart*: authentication-related screens.
- *utils.dart*: utility/helper functions used across the app.
- *main.dart*: the main entry point of the application, initializing the app, setting up routes, and loading token from storage.

9.6 Session Manager Design Pattern

The *SessionManager* class is implemented using the **Singleton design pattern**. This pattern ensures that only one instance of the SessionManager exists throughout the application's lifecycle, providing a single source of truth for authentication and session-related data.

- Declaring a private static instance:

```
static final SessionManager _instance = SessionManager._internal();
```

- Using a factory constructor to always return the same instance:

```
factory SessionManager() => _instance;
```

- Making the actual constructor private (`SessionManager._internal()`), preventing external instantiation.

Backend Architecture

The backend follows a **Layered Architecture** inspired by **Clean Architecture**, utilizing the **Repository Pattern** to ensure a clear separation of concerns. The system is structured into the following layers:

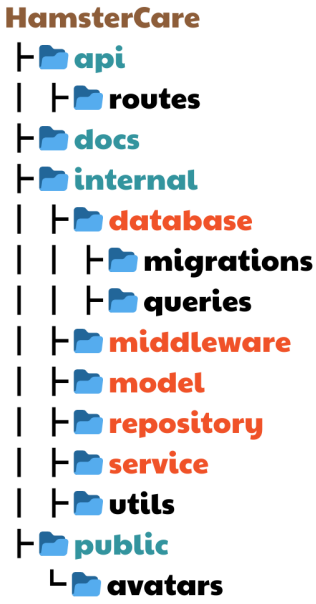


Figure 20: Backend Architecture

- **API Layer**: Responsible for handling client requests and routing them to the Service layer.
- **Service Layer**: Encapsulates business logic, serving as the intermediary between the API and other layers.
- **Repository Layer**: Provides an abstraction for database access, enabling communication with PostgreSQL while decoupling data access logic from the business logic.
- **Database Layer**: Manages data persistence through PostgreSQL, ensuring reliable data storage and retrieval.
- **Middleware Layer**: Handles cross-cutting concerns such as authentication, authorization, and logging before the request reaches the Service layer.
- **MQTT Layer**: Facilitates real-time messaging, specifically for IoT applications, ensuring low-latency communication.
- **Model Layer**: Defines the data structures (entities) used throughout the system, ensuring consistency and type safety.
- **Utils Layer**: Provides shared utility functions and helpers to support common tasks across the application.

Repository Pattern: The **Repository Pattern** is used to decouple data access logic from the business logic, offering several advantages:

- Enhances maintainability by isolating data-related concerns.
- Improves testability by allowing easy mocking of data access during testing.
- Increases flexibility by abstracting the database technology, making future changes (e.g., switching databases) easier without affecting other layers.

Data Flow:

- Client requests are received as the API layers and routed to the Service layer.
- The Service layer processes the business logic, interacting with the Repository layer to fetch or modify data.
- The Repository communicates with PostgreSQL to store and retrieve data.
- Finally, the results are returned to the client via the Service and API layers.

Benefits of the Architecture:

- **Modularity**: Clear separation of layers, making it easy to replace or upgrade components.
- **Maintainability**: The architecture allows for easier changes to business logic and data access without affecting other layers.
- **Scalability**: It supports the addition of new features and services while ensuring that the system can handle increased load.
- **Performance**: Caching mechanisms reduce the load on the database, improving response times.
- **Testability**: The use of the Repository Pattern facilitates unit testing and easy mock data handling.

10 User Interfaces

10.1 Common

10.1.1 Sign in

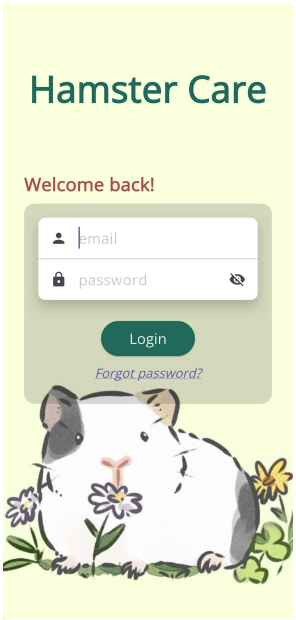


Figure 21: Sign in screen

The sign-in feature applies to users with security options, the user must enter the right email and password. If a user enters the wrong password or email, the system will alert to make the user enter these fields again. In the screen, there is also a button that if users forget their password, this will

lead the user to a reset password screen. The reset password screen will be described in the next section.

10.1.2 Forget password

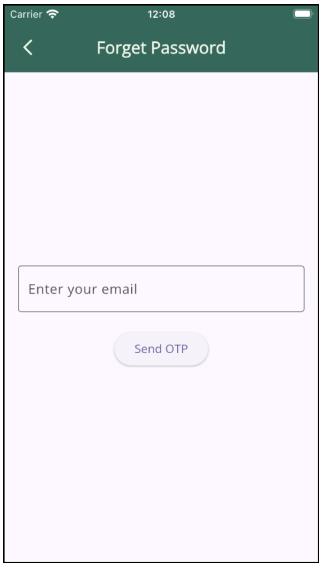


Figure 22: Forget password screen

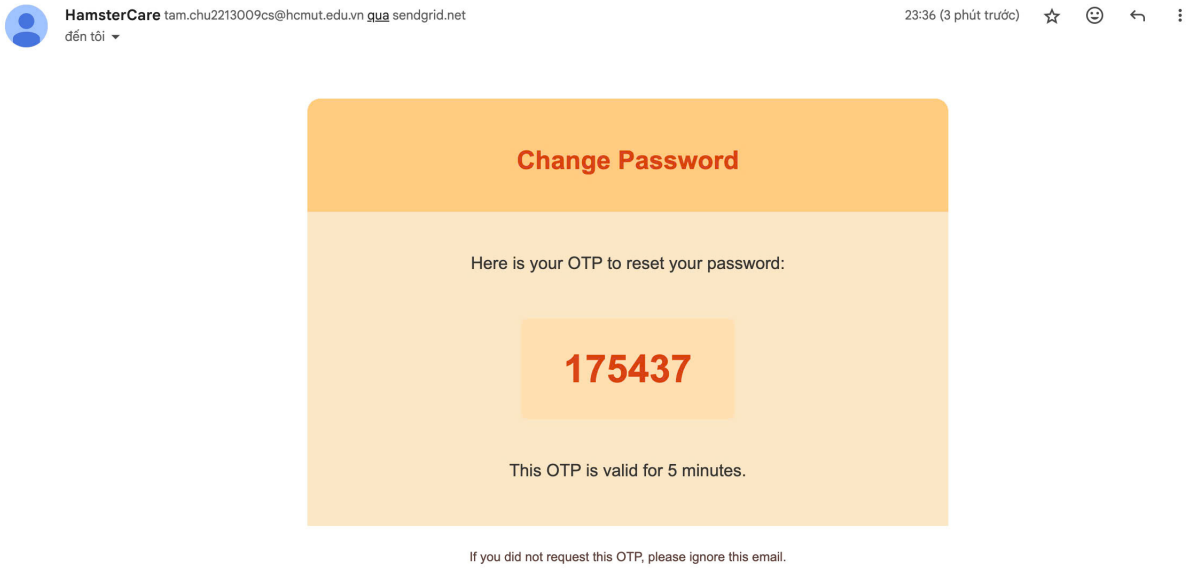


Figure 23: OTP sent

The password reset feature for users who forget password allows users who forget their password to recover their account. Users must first enter their email address, after which the system will send an OTP to their email inbox. This OTP must be entered into the designated OTP field. If the OTP is valid, the system will allow the user to set a new password and log in again.

10.1.3 Profile management

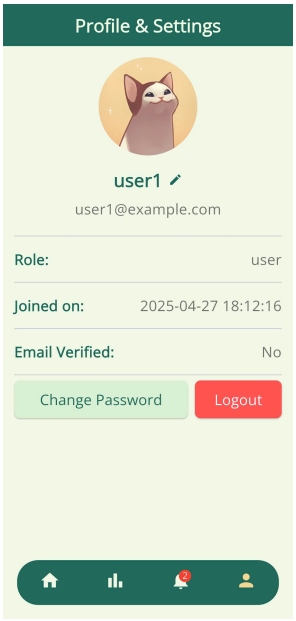


Figure 24: Profile screen

The profile screen features a simple design that displays basic user information, including the username, avatar, email, role, account creation date, and additional options such as changing the password, username, or avatar, and logging out. If the user presses the logout button, the app will return to the login screen.

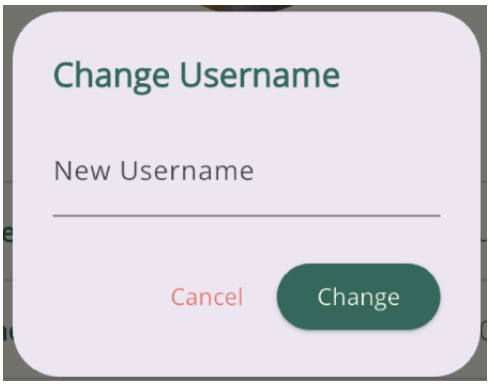


Figure 25: Change username

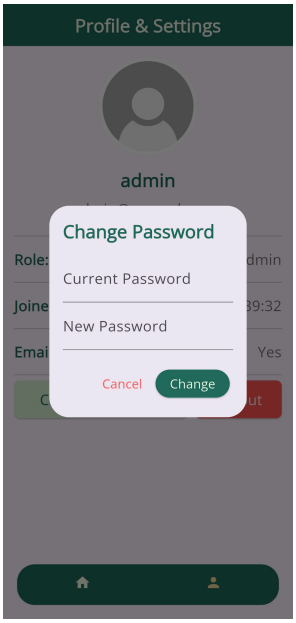


Figure 26: Change password

If users choose to change their username, a pop-up will appear prompting them to enter a new username, they need to press change to complete the workflow if the field is filled. If they press "Change" without entering anything, the app will display an alert. They can press "Cancel" to discard the process.

For the change password feature, users are required to enter their current password. If they do not remember it, they must reset their password from the login screen.

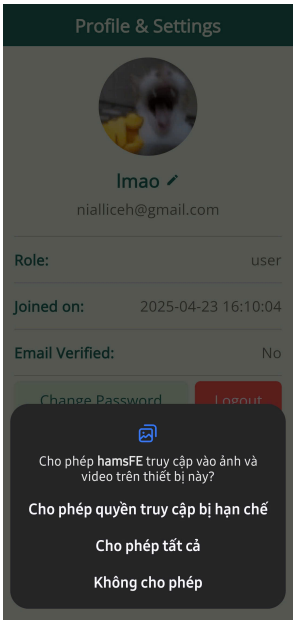


Figure 27: Change avatar

As for the avatar change feature, the system allows users to upload a photo from the device running the app. Users need to adjust their privacy settings to grant the app access to the device's image storage.

10.2 Admin

10.2.1 Home

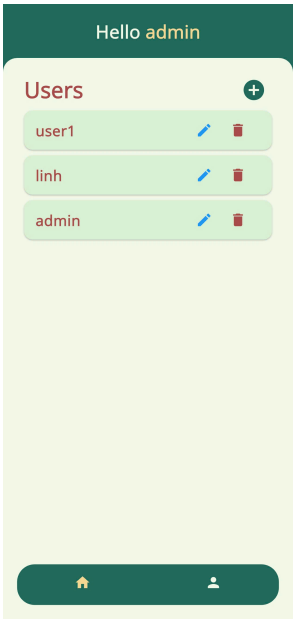


Figure 28: Admin home screen

The admin home screen displays a list of users, with options to delete or modify user accounts. The plus button allows the admin to create a new user, which redirects to the Add User screen (described in the next section). A confirmation popup will appear if the admin attempts to delete a user.

10.2.2 Add user

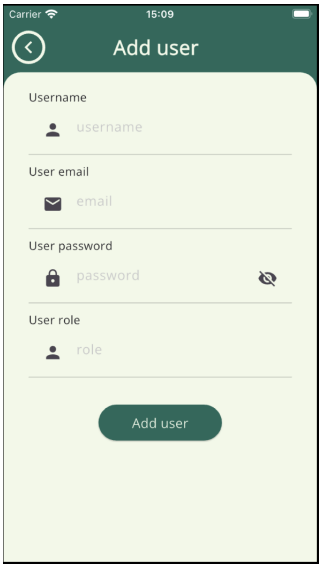


Figure 29: Admin add user

This screen contains fields that must be filled out to create a new user: username, email, password, and user role. If any field is left empty when attempting to create a new user, the app will display

an alert prompting the admin to try again.

10.2.3 User managing

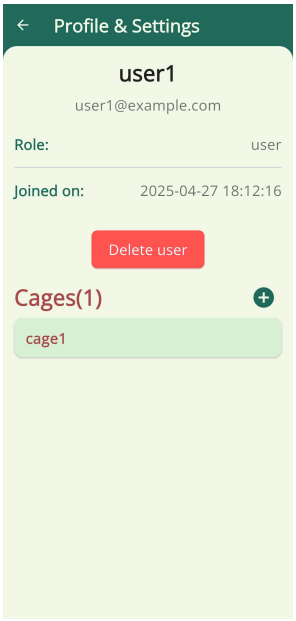


Figure 30: Admin view user

To begin with, the top section of this screen is similar to the profile screen, but with a few differences. Firstly, the admin cannot see the user's avatar. Secondly, below the profile information, there is a Delete button that allows the admin to remove the user—functioning the same way as the delete option on the home screen. Lastly, there is a list of cages, along with an Add Cage button that navigates the app to the Admin View Cage screen (described in the next section).

10.2.4 Cage management

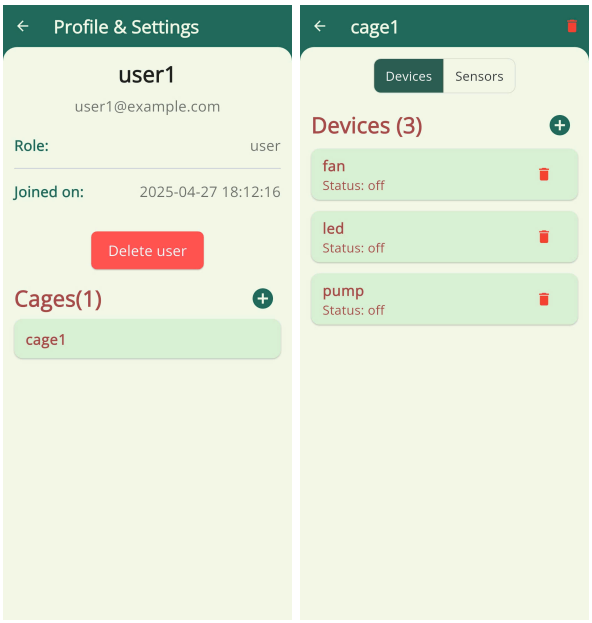


Figure 31: Admin view cage and add cage

The cage management feature for administrators begins with a list of cages displayed beneath the user profile information. Similar to how the user list is presented on the admin homepage, this section also includes a plus (+) button to activate the "Add Cage" functionality.

Pressing the button navigates the admin to a new screen where a pop-up appears, prompting them to enter the name of the new cage.

This screen is also used to display detailed information about a specific cage, and pressing the cage existed in the list will also lead to this screen; however, the detailed description of this screen will be provided in the next part of this section.

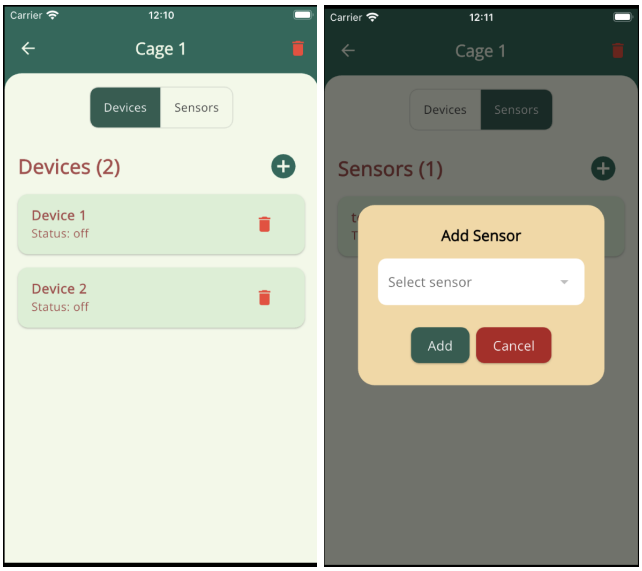


Figure 32: Admin view devices/sensors and add devices/sensors

The next part of the cage management feature for administrators involves managing a list of devices and sensors within each cage. A small navigation bar allows users to toggle between viewing the list of devices and the list of sensors. Each list is structured similarly to the other lists previously described.

When the add button is pressed for either devices or sensors, a pop-up window appears. This pop-up contains a dropdown menu listing all available devices or sensors within the system. Administrators can select an item from this list to add it to the cage.

Additionally, there are options to delete a cage or remove individual devices or sensors from a cage.

10.3 User

10.3.1 Home

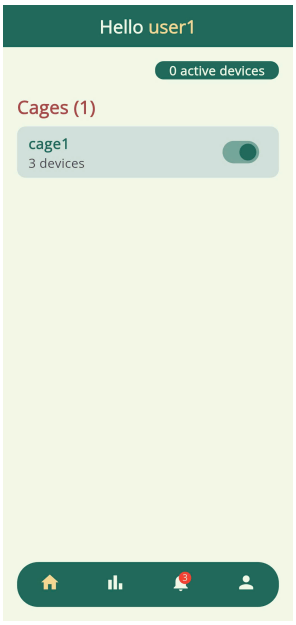


Figure 33: User home screen

The user home screen displays a list of cages, along side with the number of cages that are activated. Each cage has an option for user to activate or disable it. Pressing the cage will lead to the cage screen

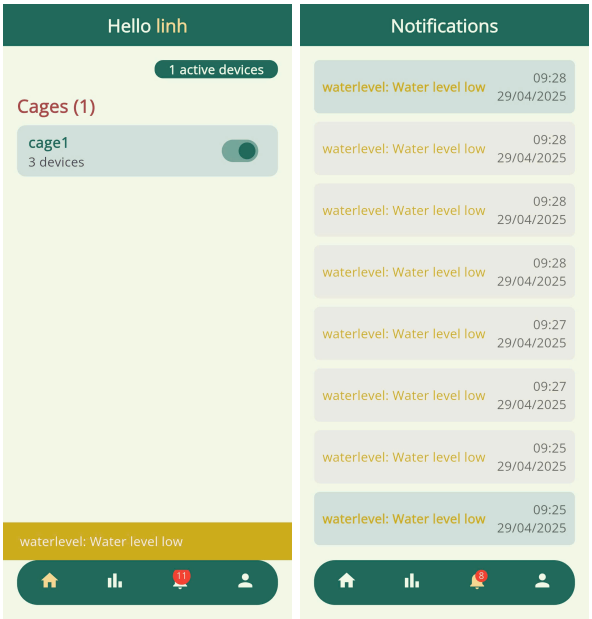


Figure 34: User notification view

The user notification view displays a list of notifications. This screen indicates which notifications have been read and which remain unread. There are four types of notifications:

- Warning: Alerts the user about issues related to hamster care conditions.

– Error: Notifies the user of any device errors (if applicable).
– Activation: Informs the user when a device has been activated.
– Deactivation: Informs the user when a device has been disabled.

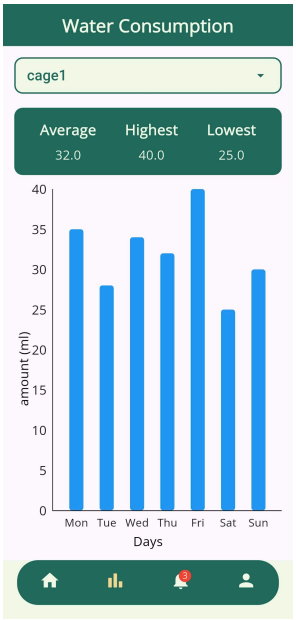


Figure 35: User statistic view

The statistic view will show the amount of water consumption for each day of the week. Users are allowed to choose which cage they want to view statistics for. A chart will display the amount of water consumption, with milliliters (ml) as the unit. Additionally, there will be a summary field showing the average daily water consumption, as well as the highest and lowest consumption recorded in a day.

10.3.2 Cage management

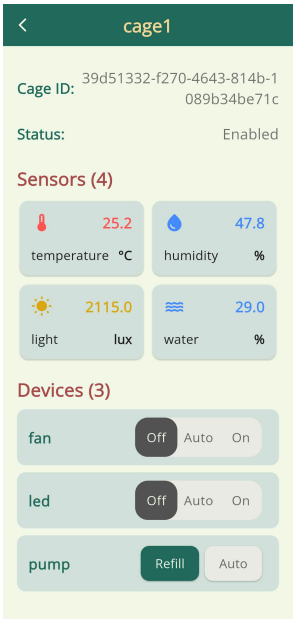


Figure 36: User cage view

Different from the admin view, the cage management feature for user begins with a list of cages displayed beneath the cage information, . Apart from that, there are also the sensors and its data.

To display the data as close as for the condition of the enviroment in real time, this app uses websocket api to get the data from the backend of the app

Each device has the option to turn off, turn on or set the device to auto mode. The detail about the auto mode will be describe in the next part of this section.

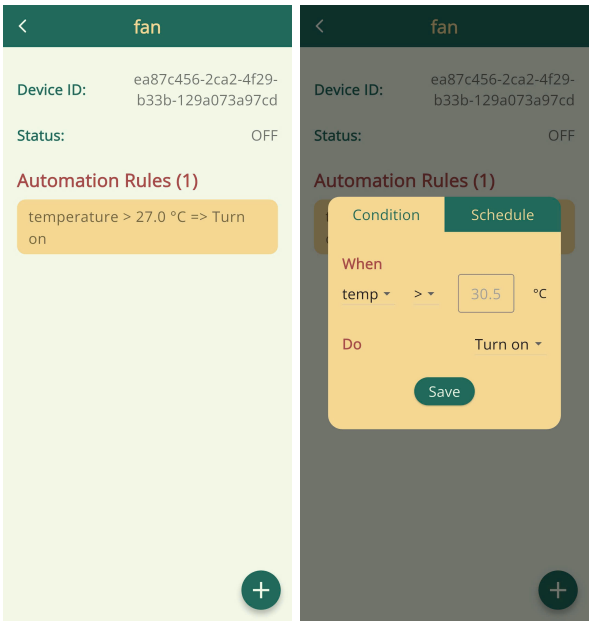


Figure 37: User view/modify device rules

The next part of the cage management feature for users involves managing device statuses and setting up rules for devices. A device rule defines a condition under which a device will automatically activate when the environmental conditions meet the specified criteria. This feature supports the auto mode of devices, as previously mentioned.

When a user wants to add a new rule, pressing the add button triggers a pop-up window. The pop-up includes:

- A dropdown menu to select the type of environmental condition to monitor
- A field to input the threshold value for the condition
- An option to define the action the device should perform when the condition is met

Additionally, users can delete existing rules by swiping the rule card to the left on the screen.

11 Evaluation

11.1 Strengths

- **Modular Device and Sensor Management**: The application allows users to manage devices and sensors by assigning them to specific cages. This approach significantly enhances the app's scalability and aligns with user needs for flexible and organized device management.

- **Automation Rule Support**: Users can create automation rules to control devices within cages based on specific conditions, supporting the growing demand for automation in IoT environments.
- **Data Visualization for Hamster Care**: The app provides statistical charts displaying the water consumption of hamsters over time. This enables users to make informed predictions and decisions to optimize hamster care routines.
- **Real-time Environmental Alerts**: Users receive warning notifications triggered by backend servers when environmental changes potentially harmful to hamster health are detected, allowing timely interventions.
- **Password Recovery via OTP**: In cases of forgotten passwords, users can reset their credentials securely via a one-time password (OTP) sent to their registered email addresses, improving user account security.
- **JWT-based Access Control**: The use of JSON Web Tokens (JWT) for managing user sessions ensures secure and appropriate access to the application's data and resources according to user roles.

11.2 Limitations

- **Missing Sensor/Device Setup Interface**: The process for configuring and setting up sensor data streams on the server side is not yet fully available in the frontend interface for administrators, resulting in potential inconveniences.
- **Manual Account and Cage Setup via Admin**: Users must request the administrator to create accounts, cages, and associate devices/sensors, which reduces system privacy and user autonomy.
- **Limited Notification Capabilities in Background Mode**: Notifications are not reliably delivered when the mobile application is fully closed, potentially causing delays in user response to critical alerts.
- **Limited Automation Rule Types**: The system currently only supports conditional rules, where automatic device actions are triggered based on real-time sensor values.

11.3 Future Improvements

- **Frontend Sensor Stream Management**: Develop an administrative interface that allows direct management and configuration of sensor data streams from the mobile frontend, minimizing reliance on server-side setup.
- **Self-service Account and Cage Management**: Implement user self-registration, cage creation, and device/sensor association functionalities within the app to improve user independence and data privacy.
- **Background Notification**: Integrate push notification services such as Firebase Cloud Messaging (FCM) to ensure critical alerts are received even when the app is closed or running in the background.
- **Support for Scheduled Automation Rules**: Extend the automation system to allow users to define scheduled rules, enabling devices to be automatically activated or deactivated at specific times or on selected days of the week.

12 Conclusion

This project has been an invaluable learning experience, allowing us to gain hands-on knowledge in software engineering. Not only have us learned how to design and implement robust systems, but also how to effectively collaborate using modern tools like GitHub for version control and Docker for environment consistency. These tools have taught me the importance of organization and automation in the development process.

In addition, we have enhanced my problem-solving skills, debugging issues, and optimizing code for performance. We also learned how to adapt to new challenges, such as integrating new technologies and managing the project through its various stages, from development to deployment.

We would also like to take this opportunity to thank my teacher for his continuous support and guidance throughout the project. His feedbacks and insights were instrumental in helping us improve and refine our work.

Overall, this project has helped us develop both my technical and collaborative skills, preparing us for future challenges in the software engineering field.

13 Installation Guide

To begin, clone the repository from the following GitHub link:

`https://github.com/jncmtam/DADN-SE`

This repository contains all the necessary files for setting up the HamsterCare project. Please follow the installation steps outlined in the subsequent sections to configure and run the application on your local machine.

References

[1] Flutter. Flutter documentation. `https://docs.flutter.dev/`.

[2] The Go Programming Language. Documentation. `https://golang.org/doc/`.

[3] Refactoring Guru. Design Patterns Explained. `https://refactoring.guru/design-patterns`.

[4] Eclipse Mosquitto. MQTT Documentation. `https://mosquitto.org/documentation/`.

[5] PostgreSQL Global Development Group. PostgreSQL Documentation. `https://www.postgresql.org/docs/`.