

Final Report

A Lab Equipment Reservation & Borrowing System

Group Members:

6522770419	Phonchana Matta
6522771896	Nipun Kharuehapaisarn
6522780863	Nawasawan Yenrompho
6522781259	Kaweewat Noisuwan
6522781333	Nattawin Yamprasert
6522790136	Haseeb Ali

Instructor:

Dr. Apichon Witayangkurn

Course:

DES424 Cloud-based Application Development (Section 1)

SCHOOL OF INFORMATION, COMPUTER AND COMMUNICATION TECHNOLOGY

**SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY
THAMMASAT UNIVERSITY
ACADEMIC YEAR 2025**

TABLE OF CONTENTS

1. Overview and Background	3
2. Problem Statement	3
3. Key Users & Stakeholders	3
4. Functional requirements	4
5. Non-functional requirements	5
6. Technology Stack	6
7. Software Architecture	7
8. System Workflows	21
9. Prototype (User Interface)	25
10. Testing Report	37
11. Conclusion	40
Links and Repository	41

1. Overview and Background

Universities and student clubs manage valuable, shareable assets—such as robot kits, engineering tools, and cameras. Currently, these items are often tracked through disjointed spreadsheets, paper forms, or informal chat messages. This leads to inconsistent procedures, poor visibility of asset availability, and inefficient resource allocation.

This inventory management app is a cloud-based, campus-wide reservation and borrowing platform designed to standardize how assets are listed, booked, checked out, and returned. The platform replaces trust-based, informal borrowing with a structured system featuring:

- Real-time availability calendars.
- On-site checkout and return using QR code scanning.
- Condition logging to record asset status before and after use.
- Automated notifications to reduce late returns and equipment loss.

2. Problem Statement

The current asset borrowing landscape relies heavily on manual procedures, resulting in several operational inefficiencies:

- **Double Bookings:** Lack of a "single source of truth" leads to scheduling conflicts, particularly during high-demand periods like exams.
- **Low Accountability:** Without strict tracking, it is difficult to assign responsibility for damaged or missing items.
- **Poor Visibility:** Staff cannot easily track utilization trends to justify new purchases.
- **Inefficient Communication:** Approvals often happen via disparate chat channels, causing missed reminders and delays.
- **Compliance Gaps:** PDPA-related consent and asset history logs are scattered across multiple platforms.

3. Key Users & Stakeholders

The platform supports three primary user roles with distinct responsibilities:

3.1. Borrowers (Students, Researchers, Club Members)

- View real-time availability of lab equipment.
- Reserve items and manage borrowing schedules.
- Scan QR codes to initiate pickup and return.
- Receive automated reminders for due dates.

3.2. Moderators (Lab Technicians, Club Officers)

- Manage item inventories (add/remove items, update status).
- Approve or reject borrowing requests (for high-risk items).
- Verify item condition upon return (Good, Damaged, Lost).
- Manage member permissions within their specific club.

3.3. Administrators (Club Advisors, SIIT Staff)

- Oversee the creation and deletion of club organizations.
- Manage high-level permissions and moderators.
- Access system-wide audit logs and compliance data.

4. Functional requirements

The system is organized around four core functional areas:

4.1 User Management System

- **Authentication:** Secure login via Google OAuth (restricted to users with a valid SIIT account).
- **Session Management:** Maintain the user's session and accurately identify their assigned role (Borrower, Moderator, Admin).
- **Profile Access:** Users can fetch their basic info, clubs they belong to, and personal borrowing history.

4.2 Club & Membership Management

- **Club Creation:** Admins can create and delete club organizations.
- **Role Assignment:** Admins/Moderators can assign, revoke, or change roles within their club hierarchy (Admin > Moderator > Member).

- **Member Management:** Members can be added, removed, and managed by moderators. Additionally, moderators and members can be added, removed, and managed by the admin.

4.3. Item & Transaction Management

- **Inventory Control:** Admins can add, update, or remove equipment from the club inventory.
- **QR Code System:** QR codes are used to scan items, and each item has a unique QR code.
- **Risk Classification:** Items are classified as "Low Risk" (auto-approve) or "High Risk" (requires moderator approval).
- **Transaction Flow:** Supports borrowing via QR scan and returning (attended or unattended).
- **Item Approval:** Moderator can approve the item with high risk when a member borrow the item.
- **Condition Logging:** Moderators log item condition (e.g., "out of service," "damaged") at checkout and return.

4.4 Notification & Reminders

- **Automated Reminders:** Send email reminders to borrowers before an item's due date.
- **Overdue Notices:** Automatically send overdue notifications if an item is not returned by the due date.

5. Non-functional requirements

5.1. Performance

- **API Response:** API response time should be under **2 seconds** for standard requests.
- **Real-Time Updates:** Real-time availability updates must reflect within **1 second** of a transaction.

5.2. Reliability & Availability

- **Uptime Target:** Target **99.5%** uptime during working hours.
- **Fault Tolerance:** Multi-AZ database deployment (RDS) to ensure failover capabilities.

5.3. Scalability

- **Concurrency:** Support at least **100 concurrent users**.
- **Architecture:** Leverage Serverless components (Lambda) and Container orchestration (ECS) to handle load spikes.

5.4. Security

- **Data Protection:** Data encryption in transit (HTTPS/TLS) and at rest (RDS encryption).
- **Access Policy:** Strict IAM role policies for backend services to enforce least privilege access.

6. Technology Stack

Component	Technology	Justification
Frontend	Vite.js (React)	Provides faster page loading, better SEO, and a smoother user experience, efficient, and easy to scale.
Backend	FastAPI (Python)	High performance; automatic OpenAPI documentation; native async support.
Database	PostgreSQL (AWS RDS)	Robust relational data integrity for transactions.
Storage	AWS S3	Scalable storage for static assets (images).
Container	Docker / AWS Fargate	Serverless container management removes infrastructure overhead.
Auth	Google OAuth 2.0	Secure integration with university email system.
Email	AWS SES	Cost-effective, reliable email delivery.

7. Software Architecture

7.1. Architecture Overview

This project adopts a **cloud-native microservices-inspired architecture** hosted on AWS. It features a containerized backend (FastAPI) managed by AWS Fargate, a decoupled frontend (React/Next.js) hosted on AWS Amplify, and a serverless notification subsystem.

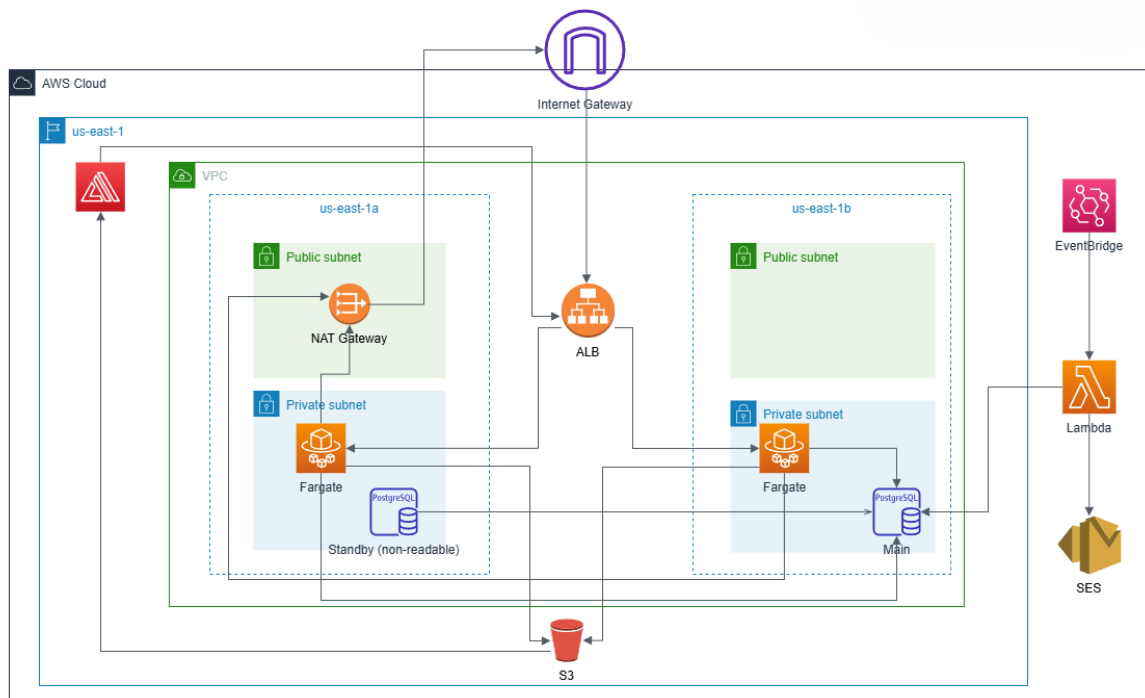


Figure 1: System Architecture Diagram on AWS Cloud

7.2. Subsystem Decomposition

The application is architected into distinct logical subsystems to ensure modularity and separation of concerns. Each subsystem is responsible for a specific domain of the application logic.

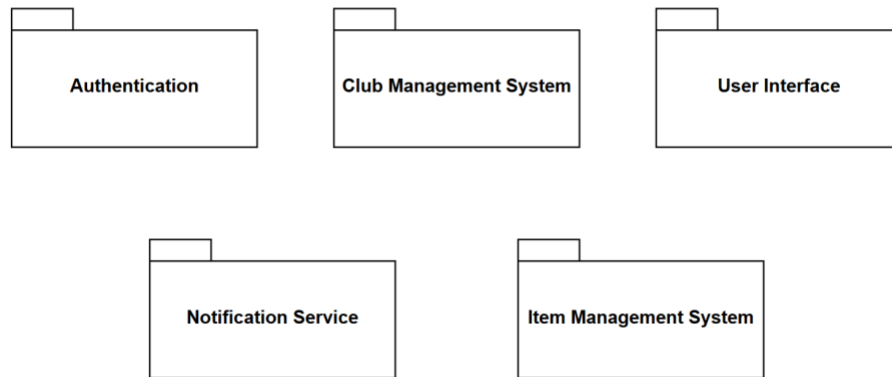


Figure 2: Subsystem Decomposition Diagram

1. Authentication Subsystem

- **Purpose:** Manages user identity verification, secure login sessions, and access control via OAuth2.
- **Key Operations:**
 - `GET /auth/` : Initiates the Google login redirection process.
 - `GET /auth/google/callback` : Handles the OAuth callback, validates the token, and establishes the user session.

2. Club Management Subsystem

- **Purpose:** Handles the lifecycle and administration of club organizations, including creation, deletion, and membership management.
- **Key Operations:**
 - `GET /clubs/` : Retrieves a list of all available clubs.
 - `GET /clubs/search` : Searches for clubs by name.
 - `POST /clubs/` : Creates a new club entity.
 - `DELETE /clubs/{club_id}` : Removes a club (Admin only).
 - `GET /clubs/{club_id}/members` : Lists all members of a specific club.
 - `PUT /clubs/{club_id}/roles/{user_id}` : Assigns roles (Admin, Moderator, Member) to users.
 - `DELETE /clubs/{club_id}/roles/{user_id}` : Removes a member from a club.
 - `POST /clubs/{club_id}/upload-image` : Uploads a club logo/image.
 - `DELETE /clubs/{club_id}/delete-image` : Deletes a club logo/image.

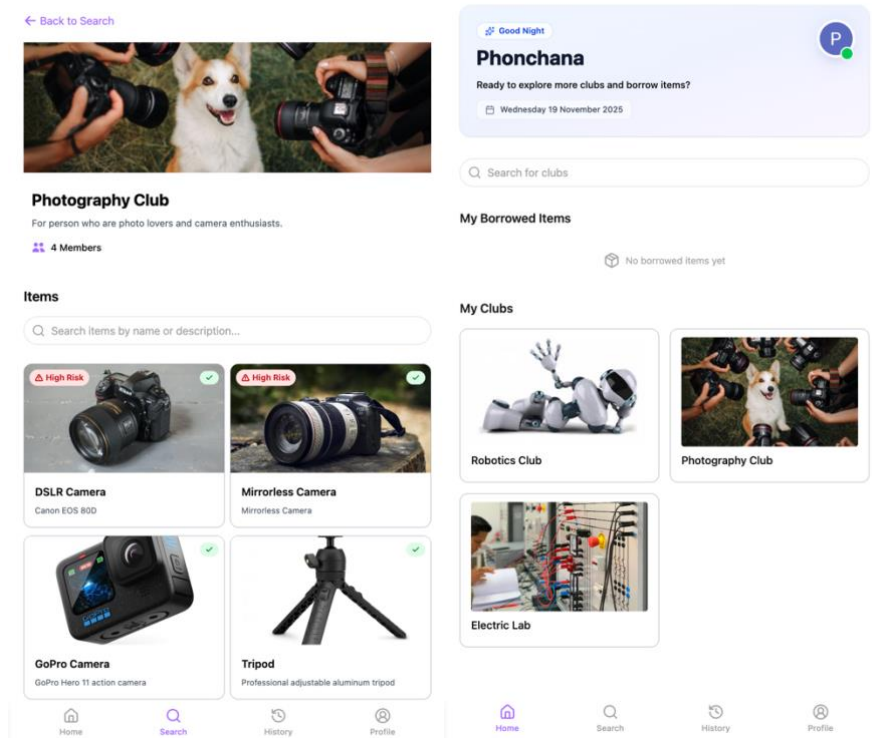
- `GET /clubs/{club_id}/details` : Fetches detailed information about a specific club.

3. Item Management Subsystem

- **Purpose:** Manages the inventory of lab equipment, including adding items, updating details, and handling transaction logic (borrowing/returning).
- **Key Operations:**
 - `POST /clubs/{club_id}/items` : Adds a new item to the club's inventory.
 - `PUT /clubs/{club_id}/items/{item_id}` : Updates item details (status, description).
 - `POST /clubs/{club_id}/items/{item_id}/upload-images` : Uploads item photos to S3.
 - `DELETE /clubs/{club_id}/items/{item_id}/delete-images` : Removes item images.
 - `POST /clubs/{club_id}/borrow` : Initiates a borrow request via QR code scan.
 - `POST /clubs/{club_id}/return` : Processes an item return via QR code scan.

4. User Interface (UI) Subsystem

- **Purpose:** The frontend application (hosted on AWS Amplify) serves as the primary interaction point for users, communicating with the backend API to present data and capture user actions.



5. User Management Subsystem

- **Purpose:** Handles user-specific data retrieval and role-based queries.
- **Key Operations:**
 - **GET /users/profile** : Fetches the authenticated user's profile information.
 - **GET /users/clubs** : Lists the clubs the user is a member of.
 - **GET /users/history** : Retrieves the user's personal borrowing history.
 - **GET /users/admin/club/{club_id}** : Lists administrators for a specific club.
 - **GET /users/moderator/club/{club_id}** : Lists moderators for a specific club.
 - **GET /users/search/** : Searches for a specific user

6. Notification Service

Purpose: To handle time-sensitive, scheduled communication (reminders, late notices) to users via email. This uses a decoupled, serverless architecture.

- **AWS EventBridge Scheduler:** Triggers the Lambda function on a set interval of 15 minutes (moderate level to be promptly and real-time while also budget-friendly)
- **AWS Lambda Function (Python):** Connects securely to the RDS database, queries for pending notifications (e.g., items due today), formats the email content, and calls SES to send email to target members who do not return the items.
- **AWS Simple Email Service (SES):** Handles the sending of emails to late item returning members.
- **RDS PostgreSQL (Private Subnet):** Stores all persistent data, including user emails, item due dates, and transactions (statuses) of items.

7.3. Data Management

The system utilizes a hybrid data management strategy, leveraging object storage for static assets and a relational database for transactional integrity.

7.3.1. Filesystem / Object Storage

- **Use case:** Data that is accessed by multiple readers but updated by a single writer.
- **Implementation:** Cloud-based storage (AWS S3) is ideal:
 - Store **club images** and **item images**
 - Provides high durability, scalability, and multiple region support

- **Characteristics:**

- Supports multiple readers
- Single writer at a time per object (API – Fargate instances)
- Images served publicly (GET only)

7.3.2 Database

- **Use case:** Data accessed by **concurrent readers and writers** (API, Lambda Service, EventBridge)
- **Implementation:** Amazon RDS PostgreSQL (Multi-AZ with 1 non-readable)
- **Data:** All data except for images, including logging data is stored.
- **Features leveraged for concurrency:**
 - ACID transactions → ensure consistency during concurrent writes

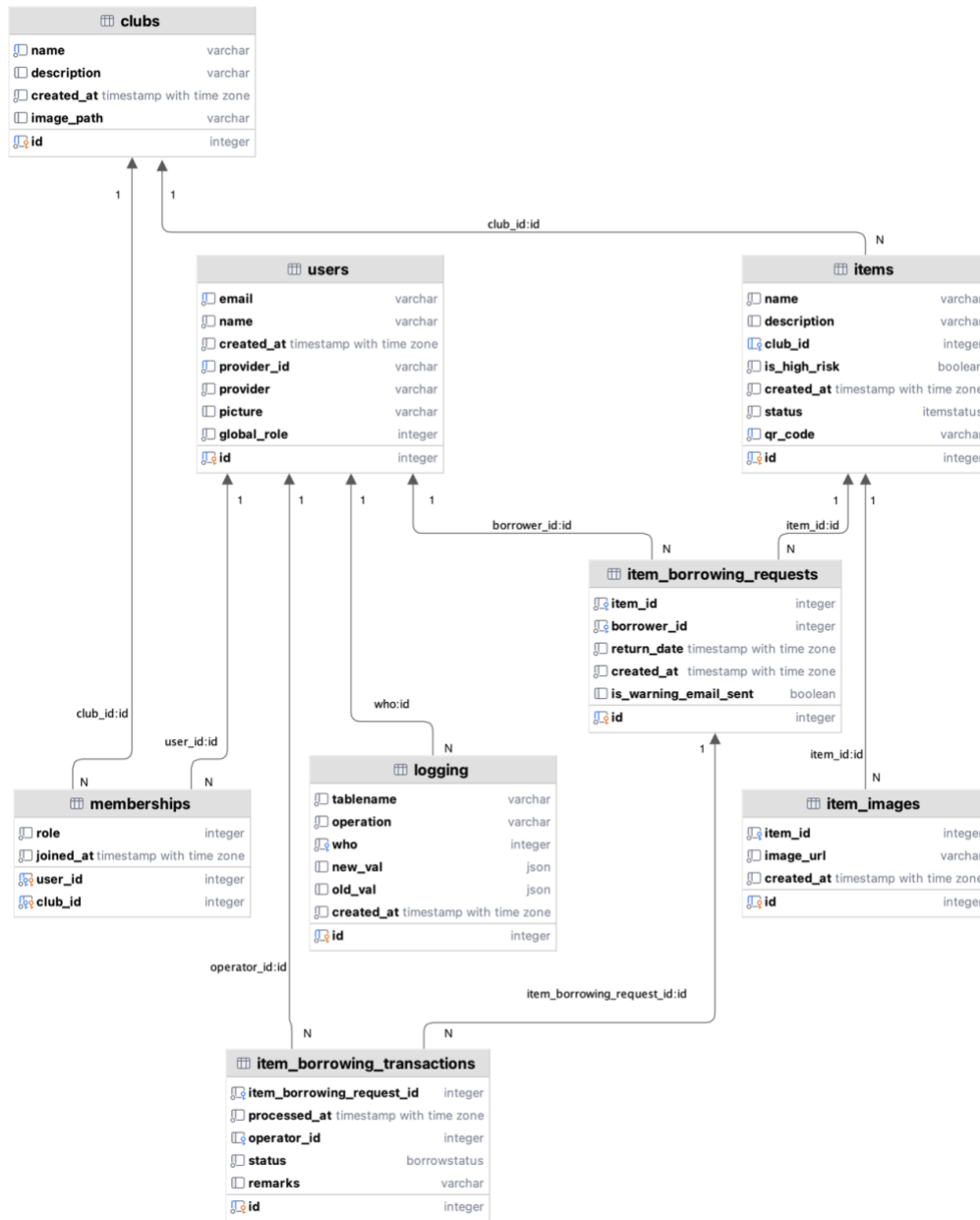


Figure 3: Entity Relationship (ER) Diagram for Inventory Management App

7.4. System Lifecycle & Boundary Conditions

7.4.1 Initialization

1. Infrastructure Initialization

Before the application can accept and process traffic, a sequence of infrastructure and service initialization steps must be completed:

- **VPC** must exist with the route tables.
- **Public subnets** must be ready for the ALB and NAT gateway.
- **Private subnets** must be ready for Fargate tasks and the RDS Postgres instance.
- **NAT Gateway** must be initialized so that Fargate tasks in private subnets have outbound internet access.
- **IAM roles** must be initialized:
 - Execution role for Fargate
 - Task role for Fargate
- **Application Load Balancer (ALB)** must be initialized with Redirect to URL and Listener rule at 443 and redirection to 443 at port 80 to ensure secure redirection to the API. Health Check URL must be consistent with Fargate (“/”).
- **RDS Postgres** must be initialized with a database schema.
- **S3 bucket** must be created for data storage.
- **AWS Lambda Service** must be configured to run **inside the VPC** in the **Private Subnets** to gain access to the RDS database and outbound internet services
- **EventBridge Schedule** must be created to trigger the Lambda function on a set time.

2.1 Data accessed

- **Environment variables from task definition as well as by Lambda service**, including:
 - Database connection string (and Migration flag)
 - Secrets (from AWS Secrets Manager or SSM)
 - OAuth2 credentials (Google client ID / client secret)
 - App-specific configuration (JWT secrets)
 - AWS Access Credentials
 - CORS
 - Run DB migrations if required (**RUN_MIGRATIONS** = 1)

- Days before which a reminder notification must be sent to user for borrowed items.
- **Docker image pulled from ECR**
 - Requires NAT gateway for internet access.
- **Application configuration files**
 - Router and middleware definitions

2.2 Backend services that must be registered

When FastAPI starts, it must register:

- **Routers**
 - /login, /clubs, /users, /items, /borrow, /returns
- **Middleware**
 - Session middleware
 - CORS middleware
- **Database connection**
 - Initialize connection pool to RDS Postgres
- **OAuth2 login provider**
 - Register Google OAuth2 client
 - Register allowed redirect URLs
- **Load Balancer Target Registration**
 - Fargate task registers itself to the ALB target group
 - Health checks must succeed

2.3 Networking requirements at backend startup

- Fargate must have:
 - Access to **NAT Gateway** to reach public services (OAuth2, API calls, ECR image pulls).
 - Access to **ALB** for inbound requests.
 - Access to **RDS** database in private subnet.

3. Frontend (Amplify Hosting) Initialization

1. **Go to AWS Amplify Console:** Open Amplify in our group AWS account.
2. **Choose “Host your web app”:** Select the option to connect a Git repository.
3. **Connect GitHub:** Log in to GitHub and give Amplify permission to access your repos.
4. **Select Your Repository + Choose the main Branch:**
Choose the repo `troposphere-frontend` (URL: <https://github.com/KaweewatN/troposphere-frontend.git>) and select the **main** branch for deployment.
5. **Enable SSR:** On the setup screen, **select** SSR hosting (Server-Side Rendering).
6. **Set Amplify Environment Name:** Enter the environment name used in your app (e.g., `prod`, `dev`, or `main`). Amplify will use this during build.
7. **Review & Click Deploy:** Amplify will build and deploy your project automatically.
8. **Automatic Deployments:** After setup, every git push to the main branch of `troposphere-frontend` will automatically trigger a new build and update your live site.

7.4.2 Termination

1. Fargate Resource Cleanup

AWS ECS/Fargate handles cleanup automatically:

- The container is stopped.
- CPU and memory resources are deallocated.
- Temporary container storage is removed.
- Network interfaces (ENIs) attached to the task are detached and deleted.

2. ALB Target Group Deregistration

Before Termination

- ECS begins deregistering the container from the ALB target group.
- The ALB stops routing new incoming requests to the task.
- Existing active connections are allowed to complete during the connection draining period.
- Other healthy tasks remain active and continue serving traffic without interruption.

On Termination

- AWS automatically handles the shutdown and cleanup of the container and its resources.
- If the backend task terminates unexpectedly, the front end may receive errors until ECS replaces the task and the ALB routes traffic to healthy instances again.
- If the frontend hosting (Amplify) terminates or becomes unavailable, the website will simply be inaccessible to users, while the backend API continues functioning normally and remains reachable through the load balancer or any other direct integrations.
- The API is built monolithic therefore, each subsystem is not allowed to terminate independently, however runtime instances of the backend are allowed to terminate independently.
- The Lambda Service and EventBridge may terminate silently, without any notification service available for users, however it will not affect the API or the frontend at all.

Database Updates

- All changes to the database occur through ACID-compliant transactions.
- Updates are made persistent by calling commit().
- On termination or failure, uncommitted transactions are rolled back automatically.
- The database remains consistent regardless of backend container shutdowns.

Failure Handling:

2.1 Backend (Fargate Task) Failure

If a single Fargate task fails:

- ALB health checks detect it as **unhealthy**.
- It is removed from the target group and no longer receives traffic.
- Other healthy backend tasks continue serving traffic.
- ECS attempts to start a replacement task automatically to maintain the desired count.

Users may experience:

- Temporary errors if no healthy backend instances are available.
- Brief increased latency during task replacement

2.2 Database Communication Failure

If FastAPI loses communication with RDS:

- Requests that require DB access fail with server errors (500 / 503).
- The backend continues running but cannot fulfill DB-dependent operations.

However since we are using Multi-AZ RDS:

- AWS automatically performs a failover to the standby instance.

2.3 Frontend (Amplify) Failure

If Amplify hosting becomes unavailable:

- The website is inaccessible to users.
- However, the backend API still remains operational.
- If there are any other clients using the API directly, will be able to do so (although, not in our project for now)

2.4 Network / NAT Connectivity Failure

If the Fargate task cannot reach the internet (e.g., NAT outage):

- Google OAuth login will fail.
- External API calls will fail.
- ECR image pulls will fail during deployment, causing new tasks not to start.

2.5 FastAPI API Bugs:

If the API faces any bugs:

- In most cases, the particular operation will not come to fruition as database commit() is performed last
- In rare cases, it might allow users to bypass role-based permissions since it is implemented in the API
- There may be unexplored cases which may cause unexpected failure causing no appropriate response from the API or unexpected behaviors

2.6 Frontend Bugs:

2.7 Lambda and EventBridge failure:

- **Lambda Timeout/Failure:** If the Lambda fails (e.g., DB connection issue, unhandled code error), **EventBridge** can be configured to retry the invocation automatically.
- **Database Down:** If RDS is unavailable (during failover), the Lambda invocation will fail with a connection error and utilize the retry mechanism. Because RDS is Multi-AZ, recovery should be quick, allowing subsequent Lambda runs to succeed.

- **SES Throttling/Failure:** If SES limits are hit (unlikely for this volume but possible), the ses:SendEmail call will fail.

3. Recovery From Failure

3.1 Automatic Recovery (AWS-Managed)

AWS provides built-in mechanisms for fault recovery:

- **ECS self-healing:** automatically starts new Fargate tasks if one fails.
- **ALB failover:** routes traffic only to healthy tasks.
- **RDS Multi-AZ failover:** promotes a standby database instance automatically.
- **Amplify recovery:** redeploy or rollback to the last working build.
- Skipped email delivery from Lambda failure is negligible since the next round of delivery is within 15 minutes

3.2 Application-Level Recovery

- Transactions are rolled back automatically on failure (commit).
- Database connection reconnects when RDS becomes available.
- FastAPI restarts on new tasks using a clean environment and fresh configuration.
- Failed requests can be retried by frontend or client applications.

3.3 Manual Recovery

- Developers can redeploy the backend task definition.
- Restart RDS or fix broken migrations/configuration.
- Roll back frontend deployments on Amplify.
- Fix environment variables or secrets.
- Manually replace corrupted images in ECR.

7.5. Access Control & Security

Security & Access Control in AWS

Configuration:

1. **Amplify (User Entry Point):**
 - This is the public-facing web application that users access.

- It sends API requests to Application Load Balancer (ALB).
2. **Application Load Balancer (ALB):**
- **Inbound:** Its security group is configured to **only** accept traffic from the internet (originating from users interacting with our Amplify app) at port 443.
 - **Outbound:** It forwards allowing traffic to the Fargate instance.
3. **Fargate Instance (API Backend):**
- **Inbound:** Its security group is configured to **only** accept traffic from the ALB. It is not publicly accessible.
 - **Outbound (Internet):** It can access the internet (e.g., for third-party APIs) by routing traffic through a **NAT Gateway** located in the public subnet.
 - **Outbound (Database):** It can send traffic to the RDS database.
4. **RDS Database:**
- **Inbound:** Its security group is configured to **only** accept traffic from specific AWS resources:
 - The Fargate Instance.
 - The Lambda function.
 - It has been made publicly available for testing/development purposes

Asynchronous Operations

We also have a separate, internal process for handling email notifications:

- **Amazon EventBridge:** This service likely triggers events on a schedule or based on database changes.
- **AWS Lambda:** EventBridge invokes a Lambda function. This function runs within the AWS environment, and its security rules allow it to connect to the RDS database to query information. After processing, it can use other services (like Amazon SES) to send emails.

Within the App we have defined several kinds of roles, which can be broadly categorized into two main roles:

Club-level Roles: are roles a user has within a club, this role is divided into three roles:

- **Admin:** the highest role within a club. They manage the club's items and (by hierarchy) its members and transactions.
 - **Item Management (Specific to their club):**
 - **Add** a new item to their club.
 - **Update** the details of an item belonging to their club.
 - **Upload** images for an item in their club.
 - **Delete** images from an item in their club.
 - **Inherited Permissions:**
 - Inherits all permissions from **Club Moderator** and **Club Member**.
 - **Role Management:** Can add, remove, or change the roles of other users in their club, as long as the target user has a role *lower* than Admin (i.e., they can manage Moderators and Members).
- **Moderator**
 - This role is responsible for managing the borrowing and returning of items.
 - **Transaction Management (Specific to their club):**
 - **Approve or reject** pending borrow requests for items in their club.
 - **Approve or reject** pending returns (condition checks) for items in their club.
 - **View the list** of all pending transactions for their club.
 - **Inherited Permissions:**
 - Inherits all permissions from **Club Member**.
 - **Role Management:** Can add, remove, or change the roles of other users in their club, as long as the target user has a role *lower* than Moderator (i.e., they can manage Members).
- **Member**
 - The baseline role for being part of a club.
 - **Borrowing & Viewing (Specific to their club):**
 - Borrow an item from their club (using a QR code).
 - Get a list of all items (or search items) within their club.
 - Get the details of their club (description, member count, etc.).
 - Get a list of all members in their club.
 - Get the details of a single member from their club.
 - Get a list of all Admins in their club.
 - Get a list of all Moderators in their club.
 - **Inherited Permissions:**
 - Inherits all permissions from Global User.

Global Access Roles

- **Superuser:** has unrestricted access to all resources of the API and can perform all available actions, across clubs and within clubs

- **User:** This is the base-level authenticated user. They have no club-specific privileges by default but can view public information and manage their own profile.
 - Search for clubs by name.
 - Search for other users by name or student ID.
 - Get their own user profile information.
 - Get their own personal borrowing history.
 - Get the list of clubs they are a member of.

8. System Workflows

This section details the core user journeys within the system.

8.1. Borrowing Workflow

1. **Login:** User logs in via Google OAuth.
2. **Search:** User searches for specific clubs or equipment items.
3. **Reservation:** User selects an item and specifies the borrowing duration.
4. **Checkout:** User scans the Item QR Code at the lab/club.
5. **Validation:** The system verifies that the QR code matches the item's QR code.
6. **Return Date Selection:** Select the date for returning back the item
7. **Confirmation:** The transaction is recorded, and the item status is updated to 'Approved' for low-risk items. High-risk items should be verified by a moderator and remain in 'Pending Approval' status.

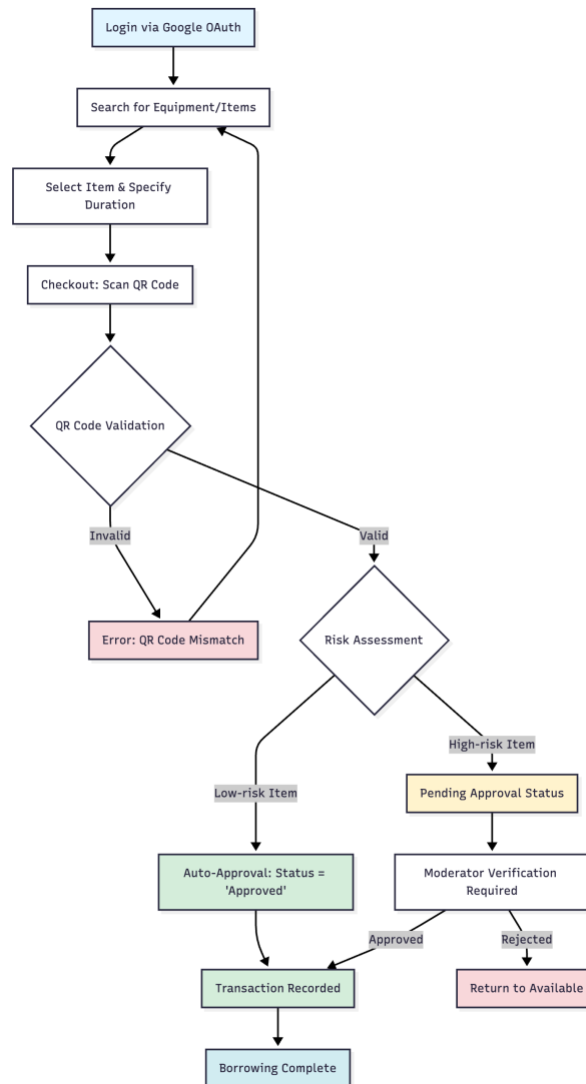


Figure 4: Borrowing Workflow

8.2. Return Workflow

1. **Initiation:** User brings the item back to the lab/club.
2. **Scan:** User scans the Item QR Code.
3. **Condition Check:** (Optional) For High-risk item, Moderator verifies the condition of the item.
4. **Completion:** Item status updates to "Available," and the user's quota is restored.

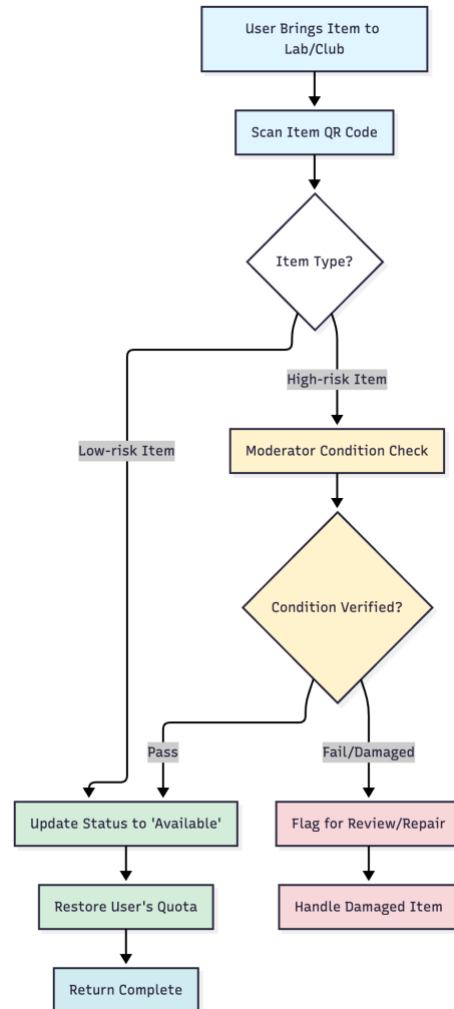


Figure 5: Return Workflow

8.3. Adding Member/Moderator Workflow

1. **Admin Management:** Admins can add, remove both members and moderators.
2. **Moderator Management:** Moderators can add, remove members, However moderator can add other moderators either.
3. **Member Management:** Members do not have permissions to add or remove users.

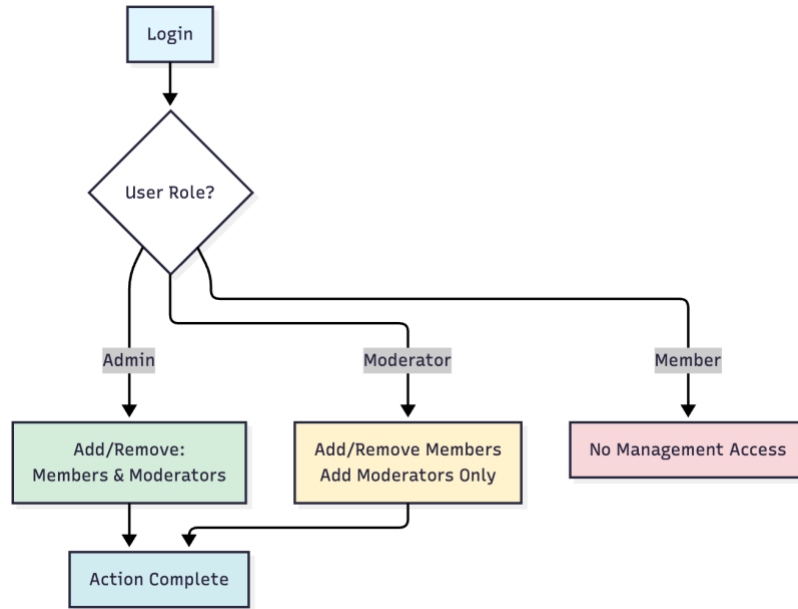


Figure 6: Adding Member/Moderator Workflow

8.3. Club Management Workflow

1. **Admin Dashboard:** Club Admin navigates to the management console.
2. **Add Item:** Admin inputs item details (Name, Description, Risk Level QR code number) and uploads photos.
3. **Publish:** Item becomes visible in the club inventory.

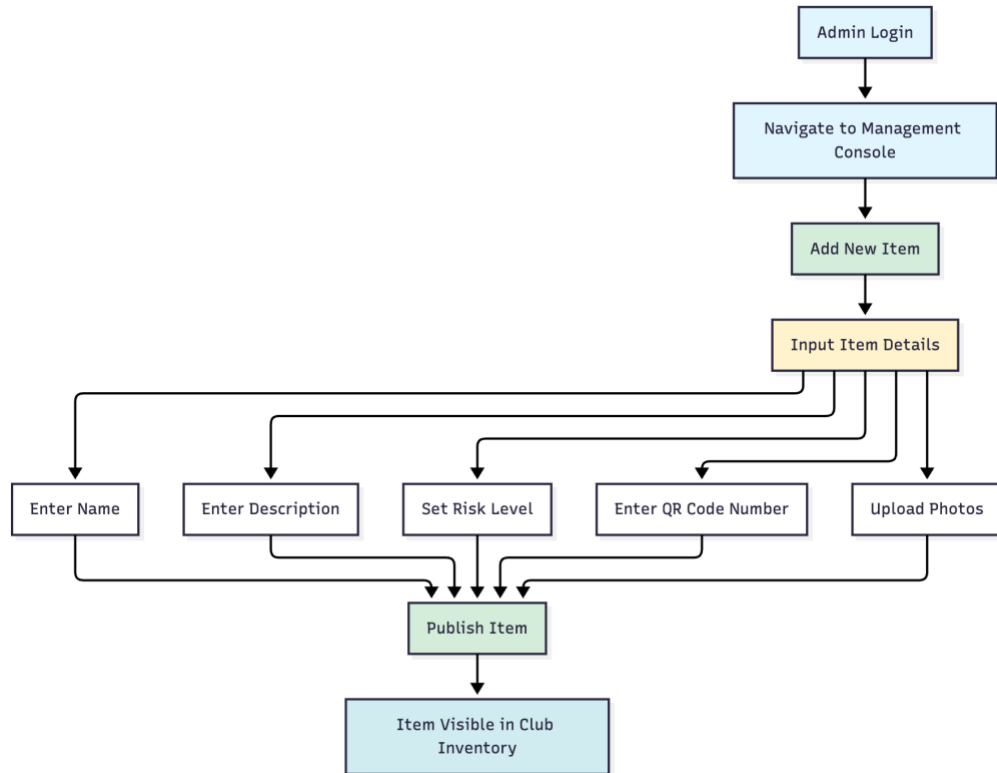


Figure 7: Club Management Workflow

9. Prototype (User Interface)

This section presents the final graphical user interface (GUI) design, demonstrating the platform's usability.

9.1. User View



SIIT Equipment Borrowing System

Borrow equipments easily and conveniently, with quick access anytime, anywhere.

Sign In to Borrow

Powered by Troposphere

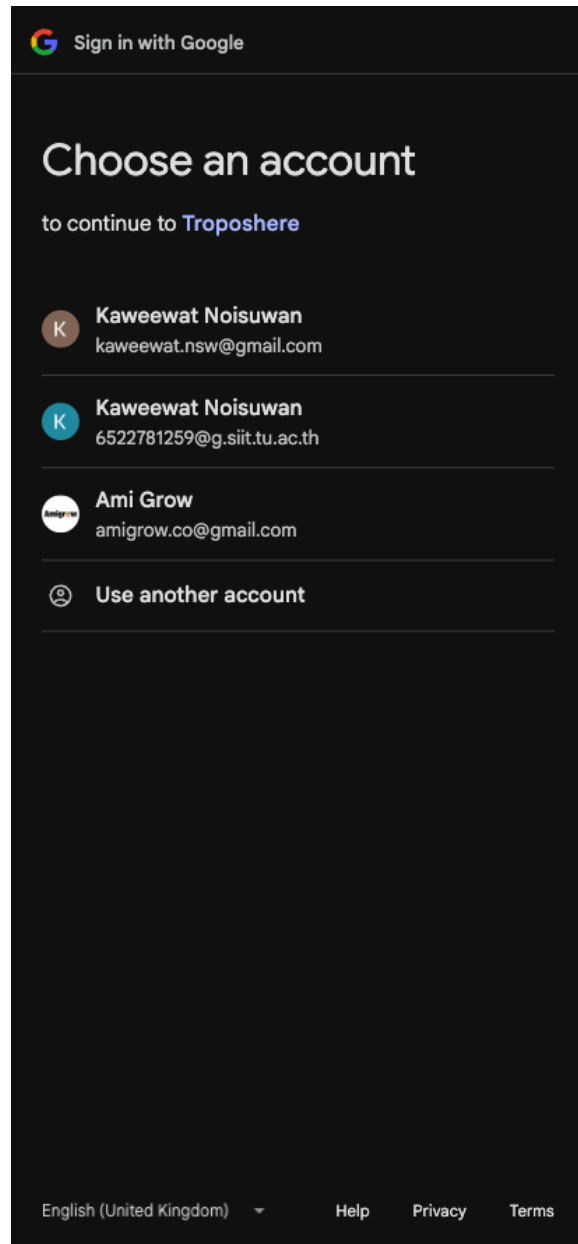


Figure 5: Sign in Page – Sign in to the system

Figure 8: OAuth page: To sign in using google account

9.2. User View

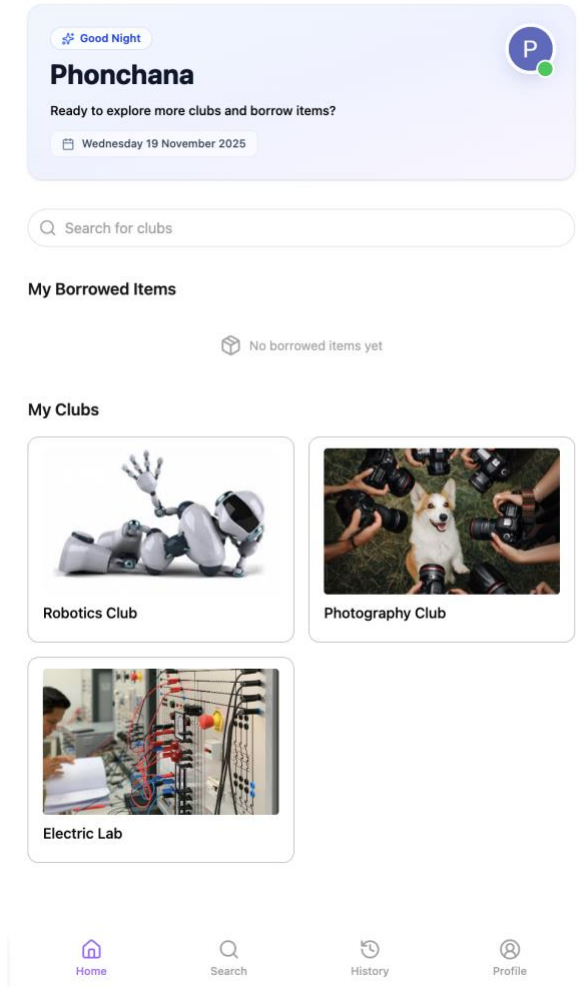


Figure 9: User Dashboard - View of available clubs and search functionality

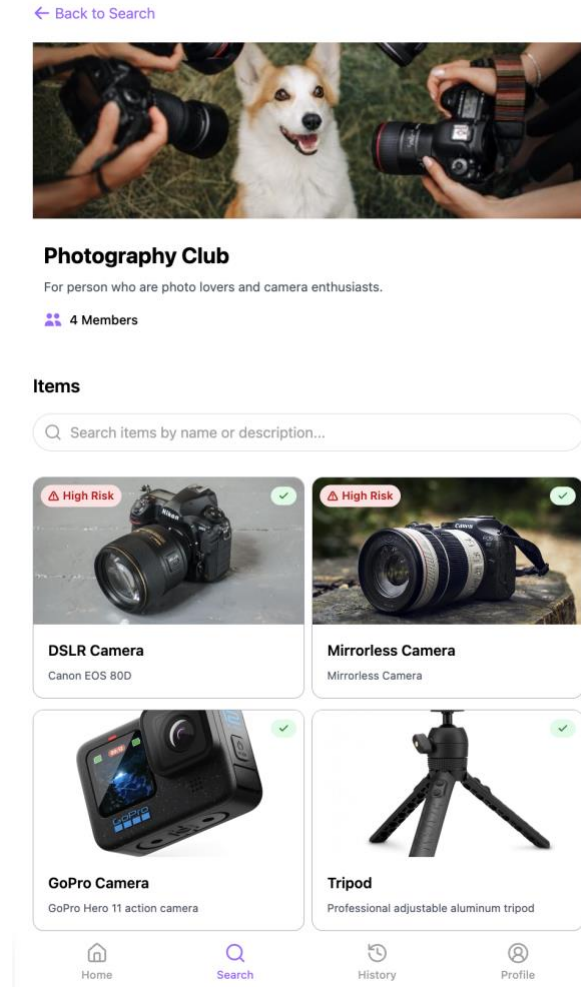


Figure 10: Club Details Screen - Viewing available club items and status

← Back



Equipment

Tripod

Type	Standard
Category	Equipment
Item Status	Available
Club	Photography Club
Created at	13 November 2025

Description

Professional



Borrow this item

Home

Search

History

Profile

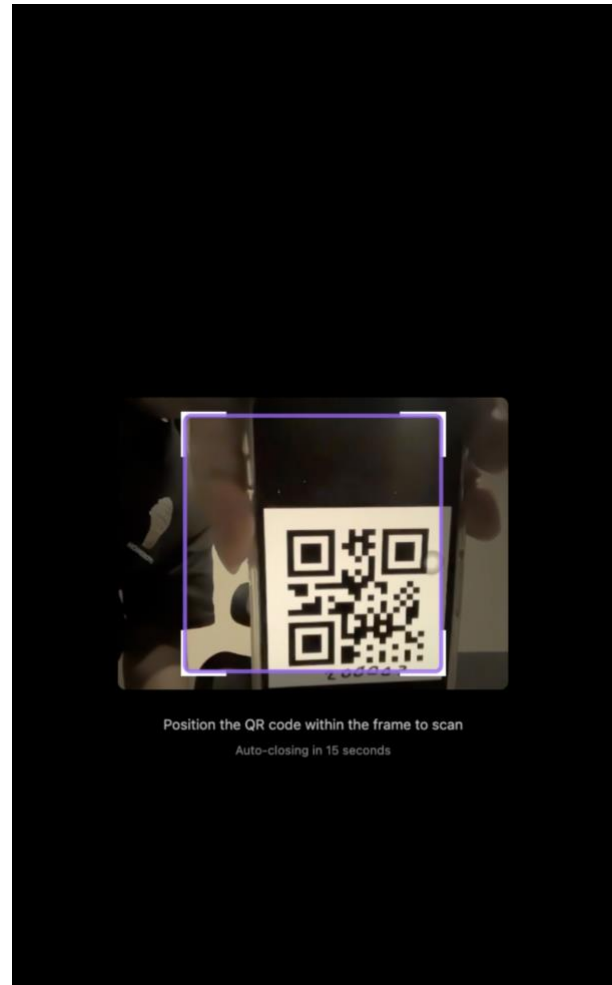


Figure 11: Item Details - Viewing equipment specifications and borrowing status

Figure 12: QR Code Scanner Interface - Active camera view for borrowing items

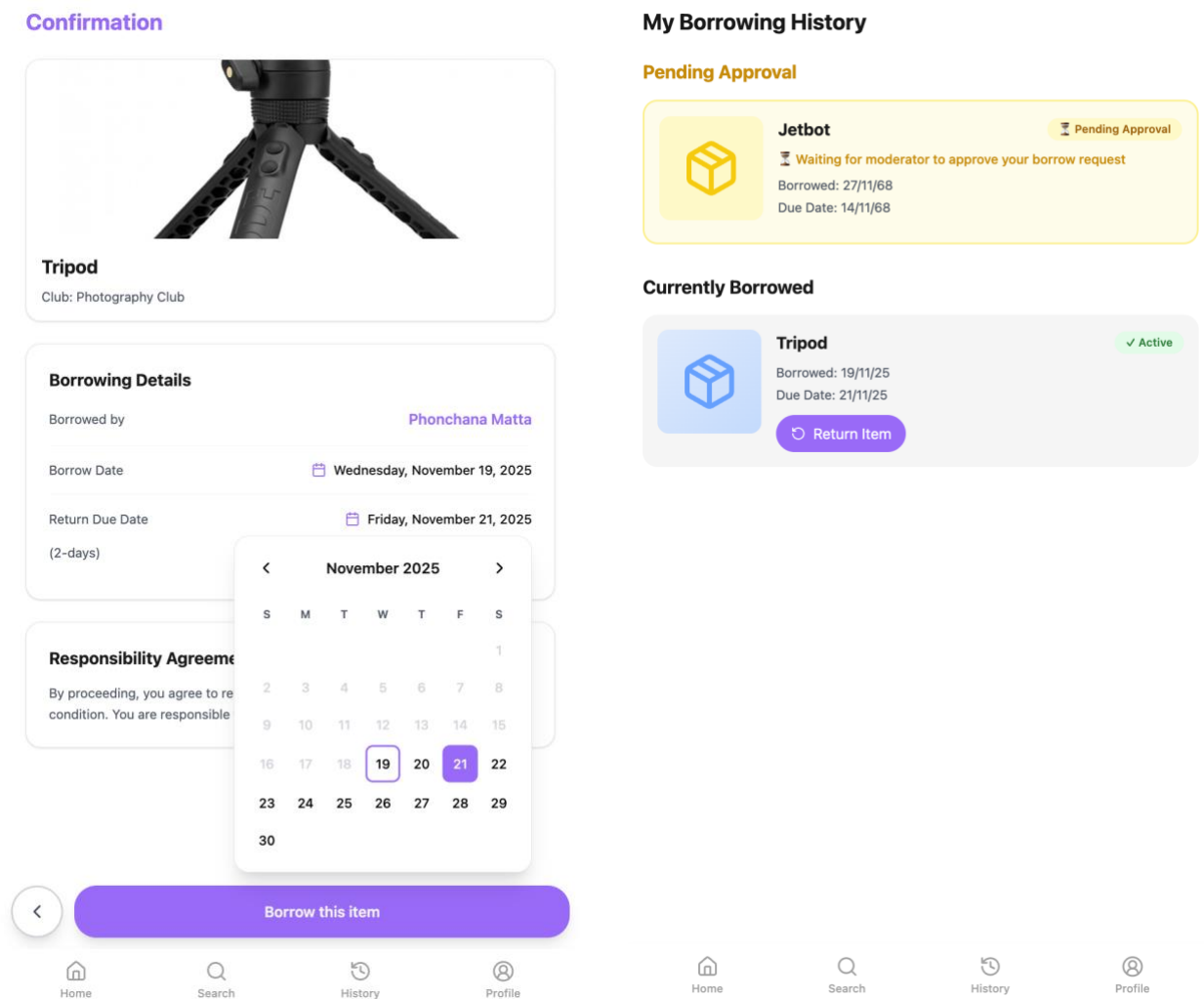


Figure 13: Borrowing Confirmation Screen - Reviewing dates and responsibility agreement

Figure 14: User History - List of pending approvals and currently borrowed items

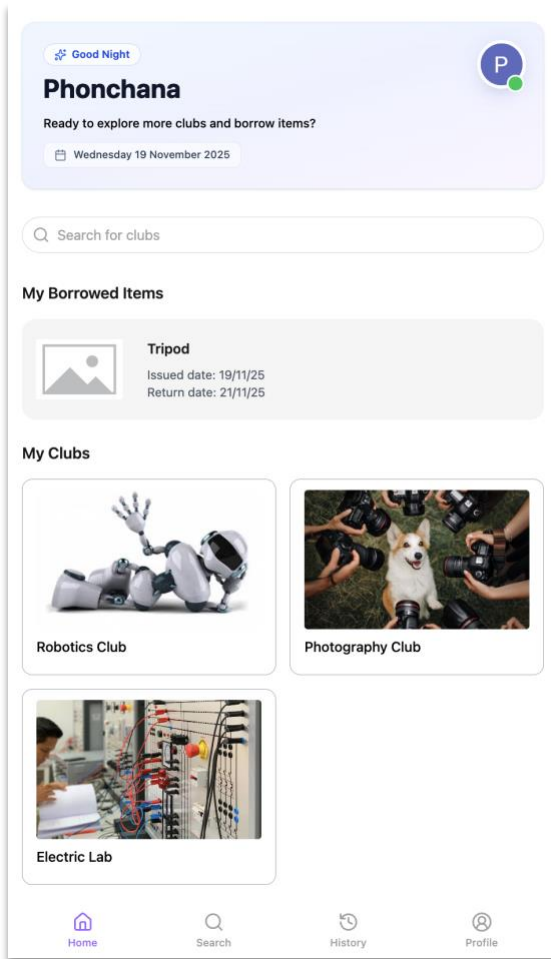


Figure 15: User Dashboard displaying borrowed items

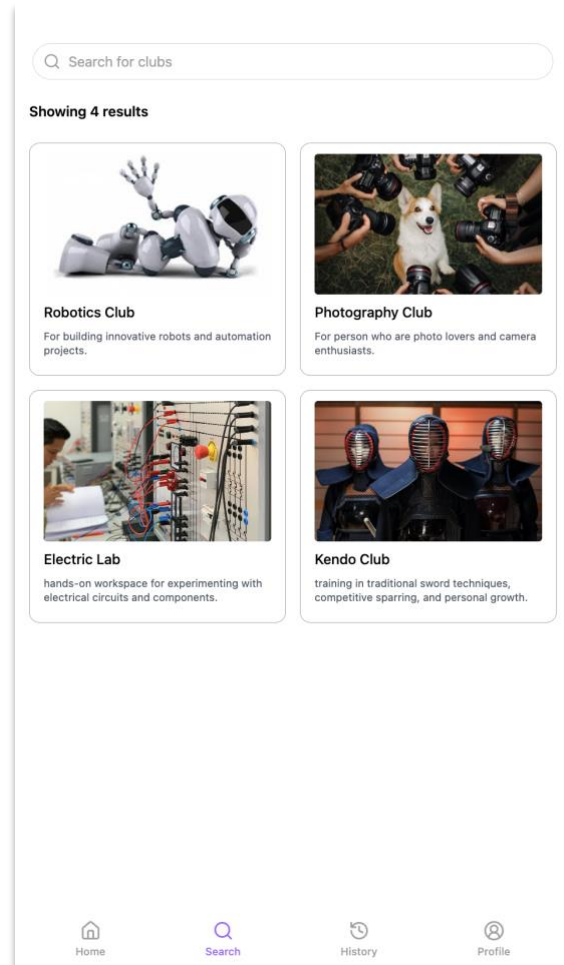


Figure 16: Search Results - Displaying search results for clubs

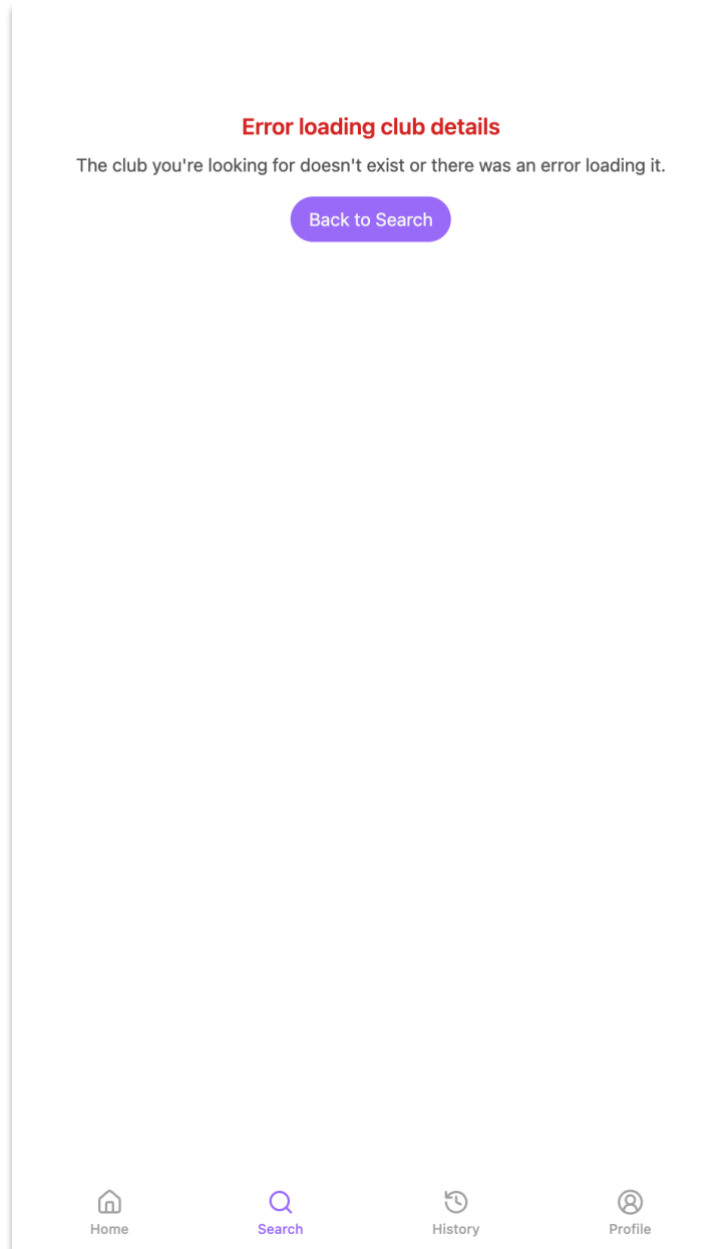


Figure 17: Access Denied - User attempting to access club details they don't belong to

9.3. Moderator View

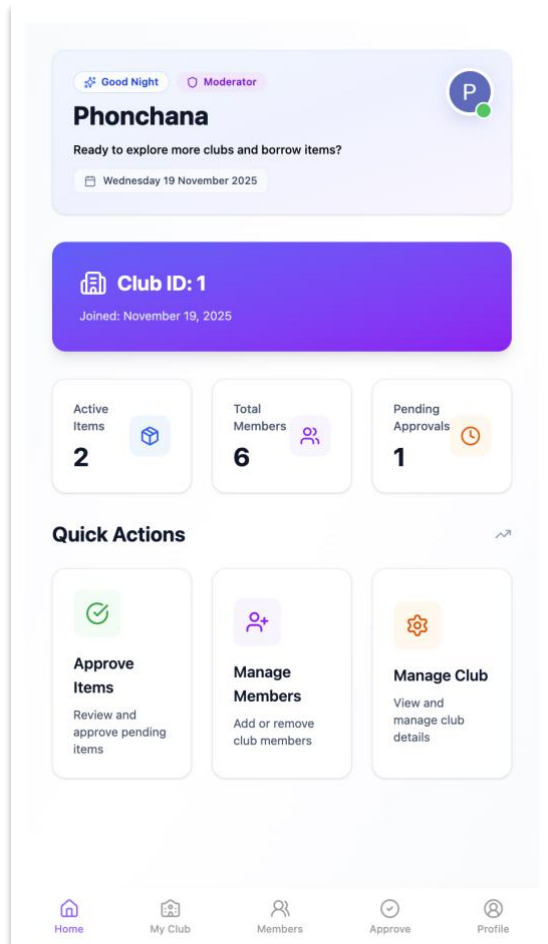


Figure 18: Moderator Dashboard - Managing pending approvals and inventory

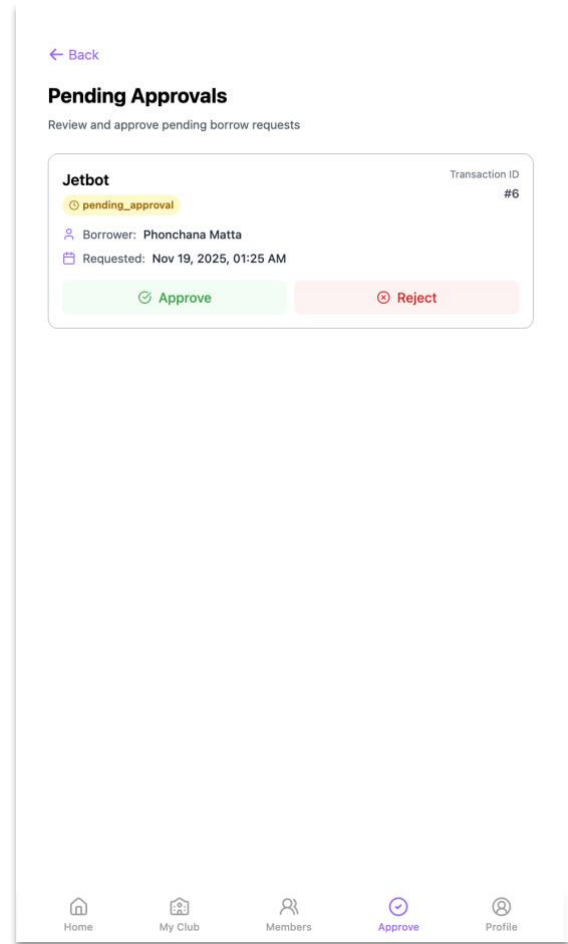


Figure 19: Pending Approval Details - Reviewing a specific borrowing request

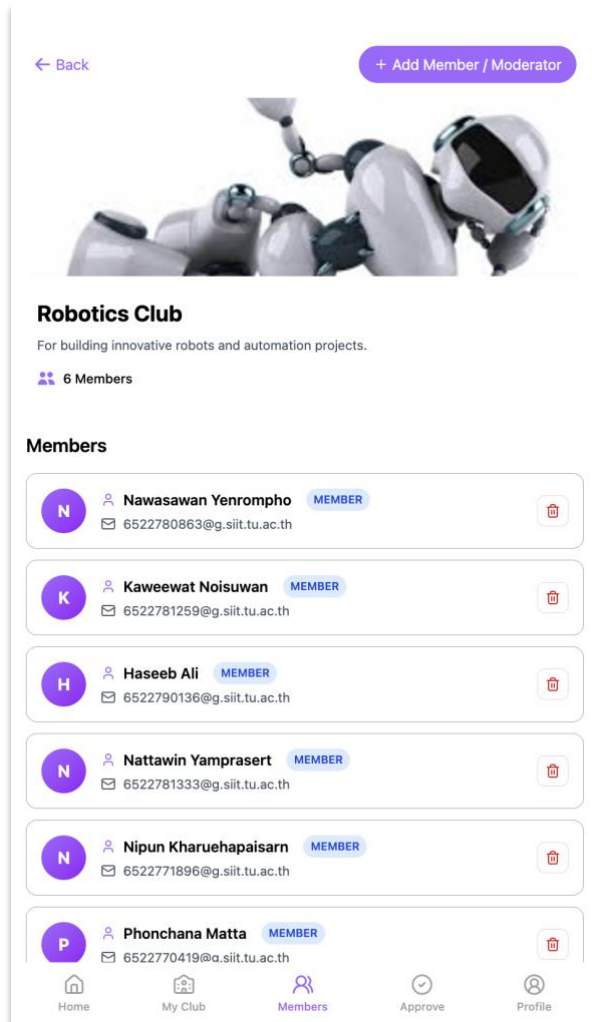


Figure 20: Club Details - Overview of club members

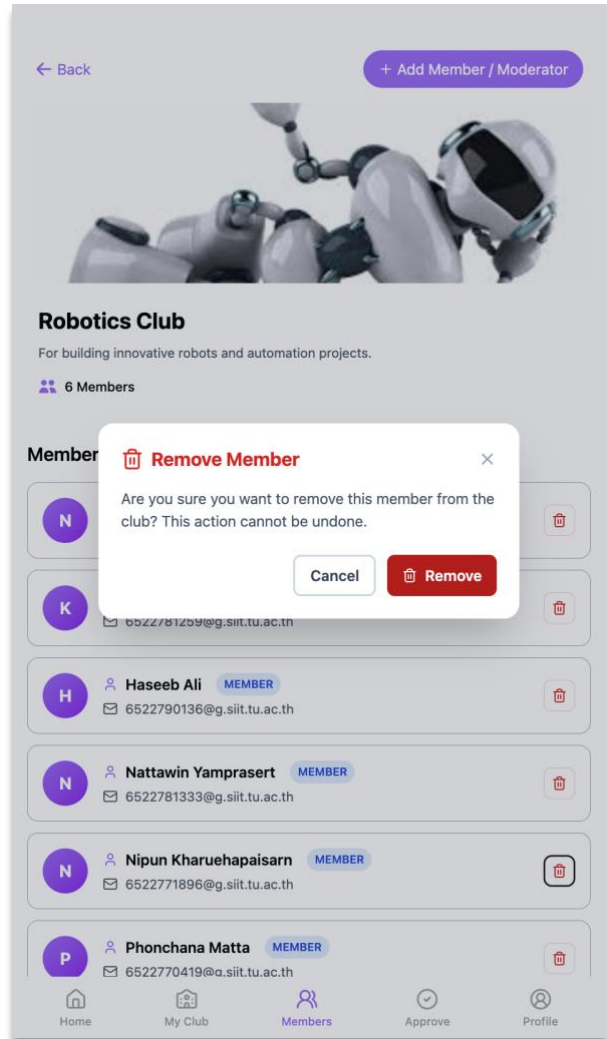


Figure 21: Remove Member Confirmation

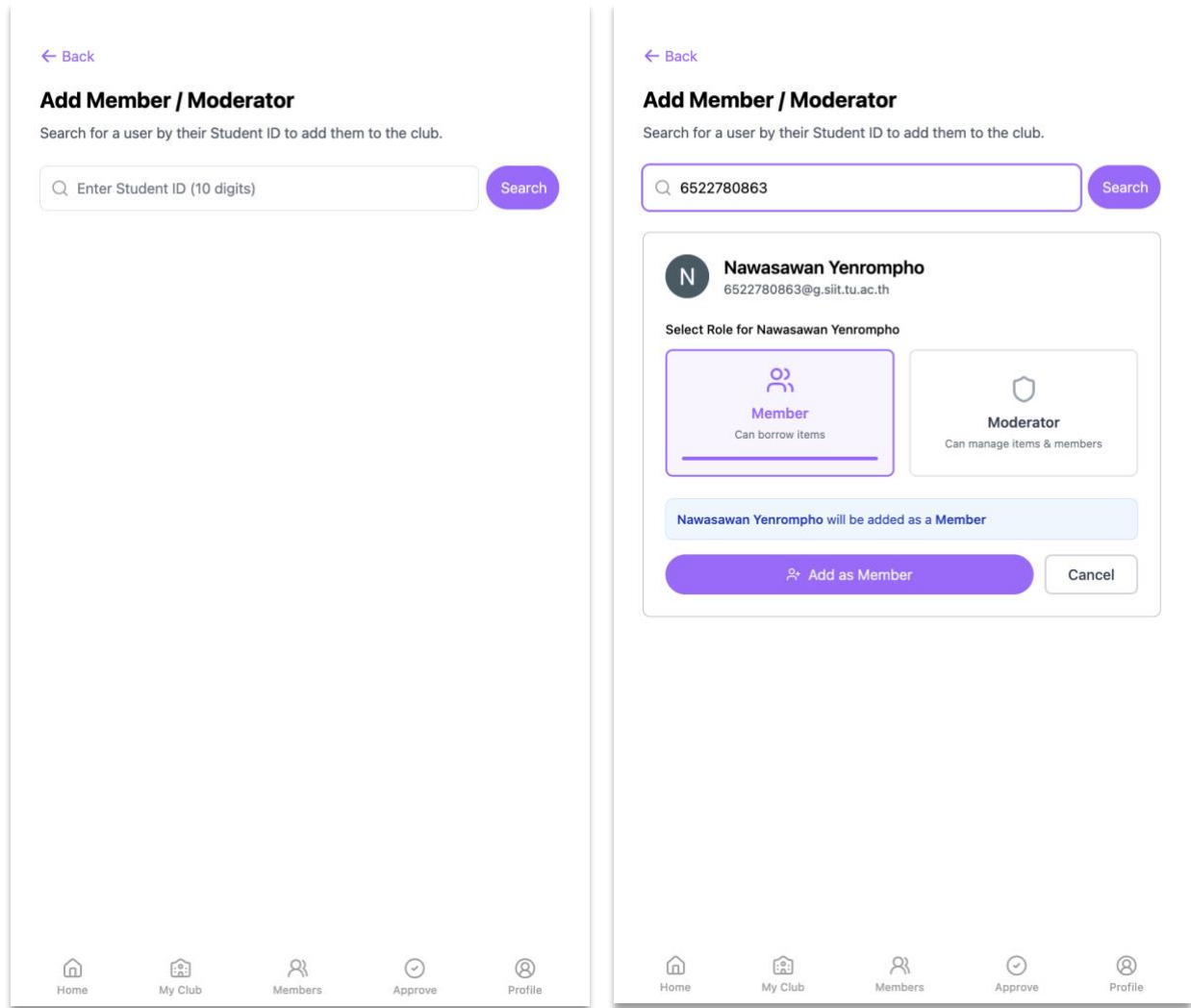


Figure 22 (left): Add Member Screen - Searching for users to add to the club

Figure 23 (right): Role Assignment - Selecting a role (Member/Moderator) for a new user

9.4. Admin View

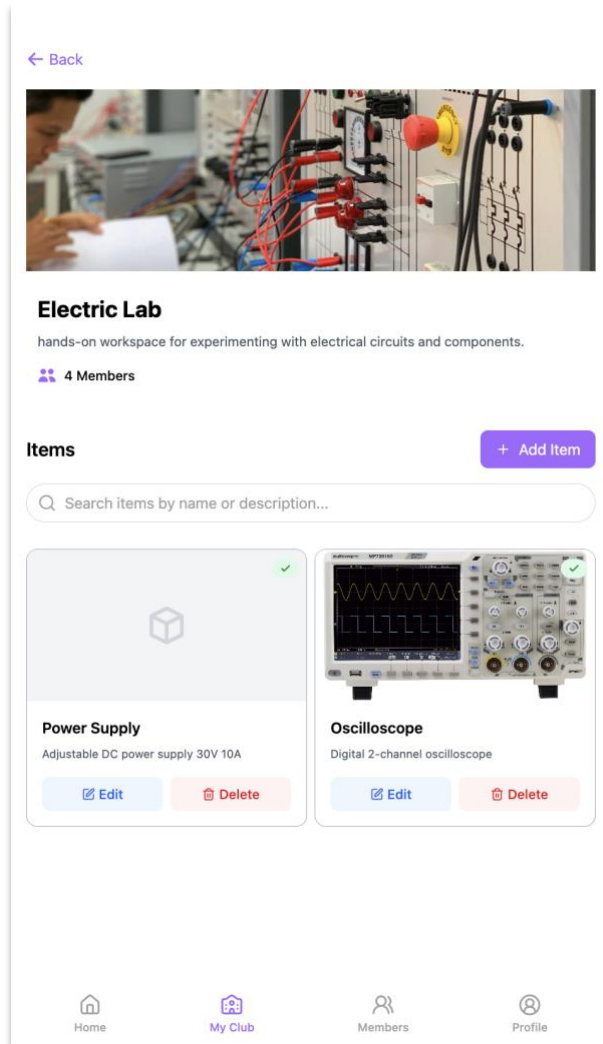


Figure 24: Admin Dashboard - Club configuration and member management

The screenshot shows the 'Add New Item' form. It starts with a 'Back' link and the title 'Add New Item'. Below the title is a instruction: 'Fill in the details to add a new item to your club'. The form contains several fields: 'Item Name' (required, with a red asterisk), 'Description' (optional), 'QR Code' (required, with a red asterisk), and 'Status' (a dropdown menu currently set to 'Available'). There is also a checkbox for 'High Risk Item' with a note: 'Mark this item as high risk if it requires special handling or permissions'. At the bottom, there are 'Cancel' and 'Add Item' buttons.

Figure 25: Admin Dashboard – Add new Item; they must fill in a form to register new item in the inventory

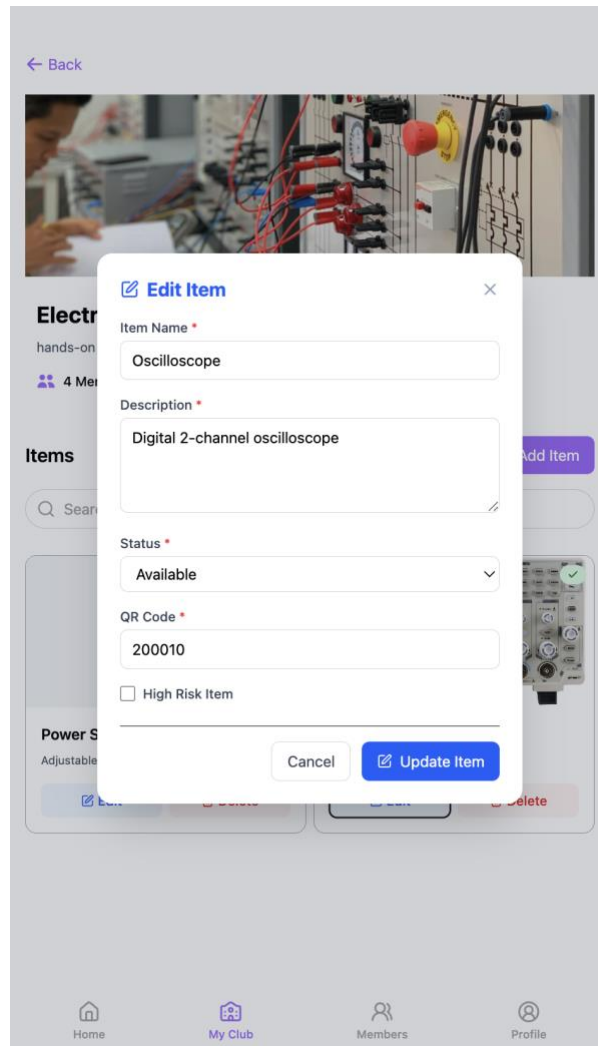


Figure 26: Edit Item Modal - Interface for updating equipment details

10. Testing Report

10.1. User Acceptance Test Summary

A series of User Acceptance Tests (UAT) confirmed that core functionality, including login, club browsing, and transaction flow, operates as expected according to the functional requirements.

No	Acceptance Requirements	Test Results	Test Date
	User Management System	Accept/Reject	
1	Login with valid Google OAuth credentials (SIIT account)	TRUE	16/11/25
2	Login with invalid/non-SIIT Google account (Access Denied)	TRUE	16/11/25
3	User session persists after refresh	TRUE	16/11/25
4	Correct user role (Borrower/Moderator/Admin) assigned upon login	TRUE	16/11/25
5	Logout function terminates session correctly	TRUE	16/11/25
	Club & Membership System		
6	View list of all available clubs	TRUE	16/11/25
7	Search for a club by name returns correct results	TRUE	16/11/25
8	Club details page displays correct info (description, members)	TRUE	16/11/25
9	Moderator can add a new member to a club	TRUE	16/11/25
10	Moderator can remove a member from a club	TRUE	16/11/25
	Item & Transaction Management		

11	Item inventory list loads correctly for a selected club	TRUE	16/11/25
12	Item status reflects real-time availability (Available/Borrowed)	TRUE	16/11/25
13	Borrow request initiated successfully via QR code scan	TRUE	16/11/25
14	"High Risk" item requires moderator approval before checkout	TRUE	16/11/25
15	Return transaction completes successfully via QR code scan	TRUE	16/11/25
16	Condition logging (Good/Damaged) saved correctly upon return	TRUE	16/11/25
	Notifications		
17	Borrower receives email reminder before due date	TRUE	16/11/25
18	Overdue notification sent if item not returned on time	TRUE	16/11/25

10.2. Automated Testing (Robot Framework)

Automated testing of the Troposphere Frontend application was conducted utilizing Selenium with Robot Framework to verify end-to-end functionality. The focus was on critical user journeys, including authentication, navigation, and interaction with dynamic elements.

Test Scope:

- **Authentication:** Gmail OAuth login process.
- **Navigation:** Basic site navigation and URL routing.
- **UI Elements:** Discovery and verification of dynamic page content.
- **Interaction:** Form input and submission handling.

Key Finding: All core tests related to element interaction and navigation passed successfully, confirming the application's stability and responsiveness under test conditions.

Observations & Risks: The testing process required manual intervention for 2FA and security bypasses, highlighting areas for environment configuration improvements to support fully automated CI/CD pipelines.

10.2.1. Execution Summary

The automated test script executed the following end-to-end flow:

1. Opened the target application URL.
2. Initiated the login process via the "Sign in to Borrow" button.
3. Handled browser security warnings (specific to the test environment).
4. Completed Gmail OAuth login (with manual 2FA step).
5. Executed post-login verification tests (Navigation, Element Discovery, Page Interaction, Page Analysis).

Overall Result: PASSED (All tests completed successfully).

10.2.2. Execution Details

Test Step	Action / Verification	Result
OAuth Login	<ul style="list-style-type: none">• Opened: https://main.d5mmkwfmc2b85.amplifyapp.com• Clicked: "Sign In to Borrow"• Handled Security Warning: Advanced → Proceed to unsafe host• Redirected to Google login, entered credentials• Manual Step: Paused for 2FA, then resumed• Redirected back to main site	SUCCESS
Test 1: Navigation	<ul style="list-style-type: none">• Verified back, forward, and refresh actions• Verified Page Title: Troposphere-frontend	PASSED
Test 2: Elements	<ul style="list-style-type: none">• Verified presence of dynamic content:• Links: 7 → 5 (dynamic change)• Buttons: 0 → 1• Input fields: 1• Div elements: 43	PASSED

Test 3: Interaction	<ul style="list-style-type: none"> • Entered text: "Selenium Test" • Submitted form • Verified page scroll functionality 	PASSED
Test 4: Analysis	<ul style="list-style-type: none"> • Verified Final URL: /search-clubs?query=Robotics+Club • Verified Page Size: 2560x1237 px • Scrolled to top, middle, bottom 	PASSED

10.2.3. Key Observations & Risks

- **Browser Security:** The test environment uses a self-signed certificate, requiring a manual bypass step in the script. This should be resolved for production environments.
- **Manual 2FA:** The requirement for manual 2FA prevents fully automated execution. This is a significant bottleneck for CI/CD integration.
- **Dynamic Content:** Element counts vary based on real-time data, requiring flexible test assertions.
- **Timing Strategy:** The use of time.sleep alongside explicit waits was effective but may be brittle on slower networks.

Identified Risks:

- **Security:** Hard-coded credentials in test scripts pose a security risk.
- **Automation:** Manual 2FA interrupts the automation chain.
- **Reliability:** Reliance on visible text for security warnings may break if browser locale changes.

11. Conclusion

The inventory management app successfully digitizes the asset management process for university clubs. By leveraging a modern cloud-native architecture on AWS, the system ensures high availability, scalability, and security. The integration of QR code technology and role-based access control addresses the critical issues of equipment loss and scheduling conflicts, providing a robust solution for campus resource sharing.

Links and Repository

Github:

Frontend: <https://github.com/KaweewatN/troposphere-frontend>

Backend: <https://github.com/alihaseeb1/Troposhere>

Notification System: <https://github.com/MystericalTH/troposphere-email-notification.git>

Robot Framework: <https://github.com/KaweewatN/troposphere-robot-framework.git>

Jira: <https://des424-g8.atlassian.net/>