Property Management Platform Design Document

Project Name: Property Management Platform
Client: CBRE Group, Inc.
Location: Boston, MA
Duration: July 2022 – Present
Developed by: Andy, Senior Software Engineer

Table of Contents

---

1. Introduction

System Purpose and Overview

The Property Management Platform is designed to improve work efficiency for community staff and enhance customer service quality. The platform offers comprehensive management services, including community asset management, visitor management, online repair reporting, and complaint management. The primary objective is to build a high-quality service platform that better serves community residents.

2. Features and Functionalities

Community Asset Management

- **Description:** Manage and monitor community assets, including facilities and equipment.
- **Functionalities:**
    - Asset registration and tracking using **SpringBoot** on the backend and **AngularJS** on the frontend.

  - Maintenance scheduling and history managed through **MySQL** database integration.
  - Asset depreciation and valuation calculations handled within the backend services.

Community Management

- **Description:** Oversee and manage community operations and interactions.
- **Functionalities:**
  - Resident information management powered by **SpringBoot** and **AngularJS**.
  - Community event planning and tracking, with data stored in **MySQL**.
  - Internal communication channels for staff facilitated by the backend and frontend setup.

Visitor Management

- **Description:** Track and manage visitor information and access.
- **Functionalities:**
  - Visitor registration and check-in/check-out using **SpringBoot** for backend processing and **AngularJS** for frontend interface.
  - Access control and monitoring, with visitor data stored in **MySQL**.
  - Visitor history and reporting capabilities integrated into the system.

Online Repair Reporting

- **Description:** Allow residents to report repair issues and track their status.
- **Functionalities:**
  - Issue reporting through a user-friendly interface built with **AngularJS**.
  - Real-time status updates and notifications managed by **SpringBoot**.
  - Repair history and analytics, with data stored and processed in **MySQL**.

Complaint Management

- **Description:** Enable residents to file complaints and monitor resolutions.
- **Functionalities:**
  - Complaint submission and tracking facilitated by **SpringBoot** and **AngularJS**.
  - Resolution status updates and notifications implemented in the backend.
  - Historical data and trends analysis stored in **MySQL**.

Role and User Management

- **Description:** Manage user roles, permissions, and authentication.

- **Functionalities:**
  - User registration and profile management using **SpringBoot** and **AngularJS**.
  - Role-based access control (RBAC) implemented with **SpringSecurity**, **JWT**, and **Redis**.
  - Secure login/logout mechanisms, including multi-factor authentication for enhanced security.

Reporting

- **Description:** Import and export reports, generate custom templates for data processing.
- **Functionalities:**
  - EasyPOI integration for report import/export, managed by **SpringBoot**.
  - Customizable report templates for Excel using **AngularJS** for frontend customization.
  - Automated data processing and analytics stored in **MySQL**.

API Standardization

- **Description:** Standardize API responses to facilitate error handling and data consistency.
- **Functionalities:**
  - Uniform API response structure designed in **SpringBoot**.
  - Enhanced error handling mechanisms integrated into the backend.
  - Improved data consistency and reliability achieved through standardized APIs.

Logging Configuration

- **Description:** Utilize Logback for precise log levels and formats.
- **Functionalities:**
  - Configurable log levels for different environments set up using **Logback**.
  - Integration with monitoring tools like **AWS CloudWatch** for real-time tracking.
  - Enhanced troubleshooting capabilities facilitated by detailed logging.

Data Integrity and Validation

- **Description:** Ensure data integrity and enforce validation rules.
- **Functionalities:**
  - Comprehensive data validation mechanisms built into **SpringBoot** services.
  - Improved data accuracy and consistency maintained through backend validation.

  - Regular audits and data quality checks stored and managed in **MySQL**.

3. Database Schema

Tables

Community Asset Module

- **Community Table: community**

Field Name	Data Type	Nullable	Description
community_id	bigint(20)	NO	Community ID
community_name	varchar(128)	YES	Community Name
community_code	varchar(128)	YES	Community Code
community_province_code	varchar(32)	YES	Province Code
community_city_code	varchar(32)	YES	City Code
community_town_code	varchar(32)	YES	Town Code
community_detailed_address	varchar(128)	YES	Detailed Address
community_longitude	varchar(32)	YES	Longitude
community_latitude	varchar(32)	YES	Latitude
dept_id	bigint(32)	YES	Property ID
community_sort	int(11)	YES	Sort Order
create_by	varchar(32)	YES	Creator
create_time	datetime	YES	Creation Time
update_by	varchar(32)	YES	Updater
update_time	datetime	YES	Update Time
remark	varchar(1024)	YES	Remark

- **Building Table: building**

Field Name	Data Type	Nullable	Description
building_id	bigint(20)	YES	Building ID
building_name	varchar(32)	YES	Building Name
building_code	varchar(32)	YES	Building Code
building_acreage	decimal(32,10)	YES	Building Area
community_id	bigint(20)	YES	Community ID
create_by	varchar(32)	YES	Creator
create_time	datetime	YES	Creation Time
update_by	varchar(32)	YES	Updater
update_time	datetime	YES	Update Time
remark	varchar(1024)	YES	Remark

- **Unit Table: unit**

Field Name	Data Type	Nullable	Description
unit_id	bigint(20)	YES	Unit ID
community_id	bigint(20)	YES	Community ID
building_id	bigint(20)	YES	Building ID
unit_name	varchar(32)	YES	Unit Name
unit_code	varchar(128)	YES	Unit Code
unit_level	int(11)	YES	Number of Floors
unit_acreage	decimal(32,10)	YES	Building Area
unit_have_elevator	varchar(32)	YES	Has Elevator
create_by	varchar(32)	YES	Creator
create_time	datetime	YES	Creation Time
update_by	varchar(32)	YES	Updater
update_time	datetime	YES	Update Time
remark	varchar(1024)	YES	Remark

- **Room Table: room**

Field Name	Data Type	Nullable	Description
room_id	bigint(20)	YES	Room ID
community_id	bigint(20)	YES	Community ID
building_id	bigint(20)	YES	Building ID
unit_id	bigint(20)	YES	Unit ID
room_level	int(11)	YES	Floor Number
room_code	varchar(32)	YES	Room Code
room_name	varchar(32)	YES	Room Name
room_acreage	decimal(32,10)	YES	Building Area
room_cost	decimal(32,10)	YES	Cost Factor
room_status	varchar(32)	YES	Room Status
room_is_shop	varchar(32)	YES	Is Shop
room_is_commercial_house	varchar(32)	YES	Is Commercial House
room_house_type	varchar(32)	YES	House Type
create_by	varchar(32)	YES	Creator
create_time	datetime	YES	Creation Time
update_by	varchar(32)	YES	Updater
update_time	datetime	YES	Update Time
remark	varchar(1024)	YES	Remark

- **Community Interaction Table: community_interaction**

Field Name	Data Type	Nullable	Description
interaction_id	bigint(20)	NO	Interaction ID
community_id	bigint(20)	YES	Community ID
create_by	varchar(32)	YES	Creator ID
update_by	varchar(32)	YES	Updater ID
create_time	datetime	YES	Creation Time
update_time	datetime	YES	Update Time
content	varchar(1024)	YES	Content
del_flag	int(2)	YES	Deletion Status (0 default, 1 deleted)
remark	varchar(255)	YES	Remark
user_id	bigint(20)	YES	User ID

Community Management Module

- **Owner Table: zy_owner**

Field Name	Data Type	Nullable	Description
owner_id	bigint(20)	NO	Owner ID
owner_nickname	varchar(30)	YES	Nickname
owner_real_name	varchar(30)	YES	Real Name
owner_gender	varchar(10)	YES	Gender (unknown/male/female)
owner_age	int(3)	YES	Age
owner_id_card	varchar(20)	YES	ID Card Number
owner_phone_number	varchar(11)	YES	Phone Number
owner_open_id	varchar(64)	YES	OpenID
owner_wechat_id	varchar(64)	YES	WeChat ID
owner_qq_number	varchar(20)	YES	QQ Number
owner_birthday	date	YES	Birthday
owner_portrait	varchar(256)	YES	Portrait
owner_signature	varchar(64)	YES	Signature
owner_status	varchar(10)	YES	Status (enable/disable)
owner_logon_mode	varchar(10)	YES	Registration Mode (WeChat/app/web)
owner_type	varchar(32)	YES	Owner Type
owner_password	varchar(128)	YES	Password
create_by	varchar(32)	YES	Creator
create_time	datetime	YES	Creation Time
update_by	varchar(32)	YES	Updater
update_time	datetime	YES	Update Time
remark	varchar(1024)	YES	Remark

- **House Binding Table**

Field Name	Data Type	Nullable	Description
owner_room_id	bigint(20)	NO	House Binding ID
community_id	bigint(20)	YES	Community ID
building_id	bigint(20)	YES	Building ID
unit_id	bigint(20)	YES	Unit ID
room_id	bigint(20)	YES	Room ID
owner_id	bigint(20)	YES	Owner ID
owner_type	varchar(32)	YES	

4. High-Level Design

Microservice Architecture

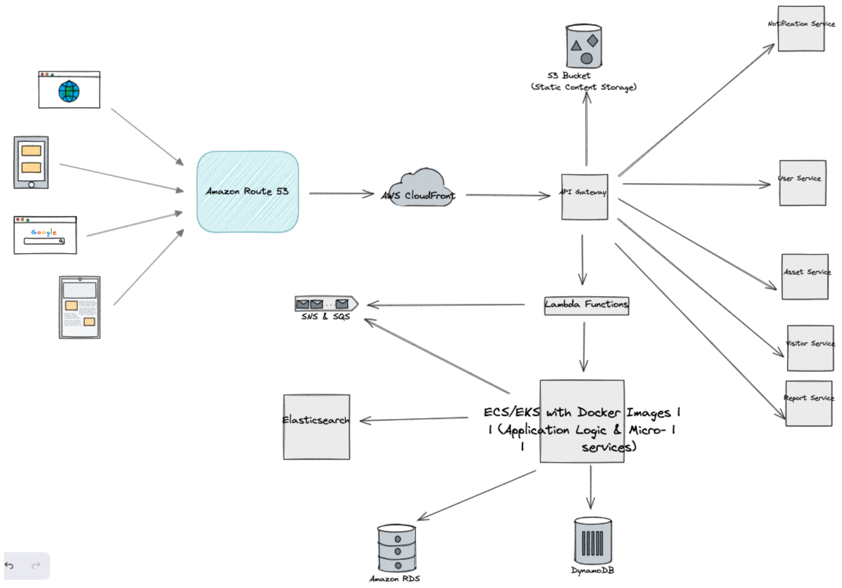

Figure 1: Microservice Architecture

- **User Service:** Manages user authentication, roles, and permissions.
- **Asset Service:** Handles community asset management.
- **Visitor Service:** Manages visitor information and tracking.
- **Report Service:** Facilitates report generation and management.
- **Notification Service:** Sends notifications and alerts to users.

Module Pictures

```
+--------------------+          +--------------------+
|    User Service    |<----->| Authentication and |
| - Manages user     |       | Authorization      |
|   authentication,  |       | - Handles login,   |
|   roles, and       |       |   logout, and      |
|   permissions      |       |   permissions      |
+--------------------+       +--------------------+
            ^
            |
            v
+--------------------+       +--------------------+
|    Asset Service   |<----->|  Community Asset   |
| - Manages and      |       |  Management        |
|   monitors         |       | - Handles asset    |
|   community assets |       |   registration,    |
|                    |       |   maintenance, and |
+--------------------+       |   valuation        |
            ^
            |
            v
+--------------------+       +--------------------+
|   Visitor Service  |<----->|  Visitor Management|
| - Manages visitor  |       | - Handles visitor  |
|   information and  |       |   registration,    |
|   tracking         |       |   check-in/out, and|
|                    |       |   access control   |
+--------------------+       +--------------------+
            ^
            |
            v
+--------------------+       +--------------------+
|   Report Service   |<----->|  Report Generation |
| - Facilitates      |       |  and Management    |
|   report generation|       | - Handles report   |
|   and management   |       |   import/export and|
|                    |       |   analytics        |
+--------------------+       +--------------------+
            ^
            |
            v
+--------------------+       +--------------------+
| Notification Service|<---->|  Notifications and |
| - Sends notifications      |  Alerts            |
|   and alerts to users      | - Manages real-time|
|                            |   notifications and|
|                            |   alerting         |
+--------------------+       +--------------------+
```

Detailed Steps for Each Data Flow

1. **User Authentication and Role Management**
    - **User Service** handles user authentication using **SpringSecurity**, **JWT**, and **Redis**.
    - Users log in through the frontend (AngularJS), which communicates with the **User Service** to authenticate credentials and manage sessions.
    - Once authenticated, the **User Service** assigns roles and permissions to users.
2. **Community Asset Management**
    - **Asset Service** manages the community assets through **SpringBoot**.
    - The frontend (AngularJS) allows administrators to register and monitor assets.
    - Data is stored and retrieved from the **MySQL** database, with maintenance schedules and asset valuations handled within the service.
3. **Visitor Management**
    - **Visitor Service** tracks visitor information and access using **SpringBoot**.
    - Visitors register their information and check in/out via the frontend interface.
    - Access control is monitored and logged, with visitor data stored in **MySQL**.
4. **Report Generation and Management**
    - **Report Service** facilitates the import and export of reports using **EasyPOI**.
    - Customizable templates allow users to generate and analyze data reports.
    - Reports are processed and managed through **SpringBoot**, with data storage handled by **MySQL**.
5. **Notifications and Alerts**
    - **Notification Service** sends real-time notifications and alerts to users.
    - Integrated with **AWS CloudWatch** for monitoring, it ensures timely and relevant notifications.
    - Users receive alerts through various channels, including emails and in-app notifications.

- **User Service** interacts with the authentication and authorization module to manage user credentials and roles.
- **Asset Service** oversees community assets, facilitating asset management and maintenance scheduling.
- **Visitor Service** ensures proper tracking of visitor information, allowing for secure access control.
- **Report Service** handles the generation and management of reports, integrating with EasyPOI for import/export functionality.

- **Notification Service** is responsible for sending alerts and notifications to users, enhancing real-time communication within the community.

5. REST API Design

Overview The REST API provides a set of endpoints for interacting with the Property Management Platform. These endpoints support operations for managing users, assets, visitors, reports, and notifications.

Endpoints **User Service - POST /api/v1/users** - Description: Create a new user - Request Body: `json { "username": "string", "password": "string", "email": "string", "role": "string" }` - Response: 201 Created

- **GET /api/v1/users/{id}**
  - Description: Retrieve user details by ID
  - Response: 200 OK
  - Response Body:
    ```
    {
      "id": "integer",
      "username": "string",
      "email": "string",
      "role": "string"
    }
    ```

Asset Service - POST /api/v1/assets - Description: Create a new asset - Request Body: `json { "name": "string", "type": "string", "value": "decimal" }` - Response: 201 Created

- **GET /api/v1/assets/{id}**
  - Description: Retrieve asset details by ID
  - Response: 200 OK
  - Response Body:
    ```
    {
      "id": "integer",
      "name": "string",
      "type": "string",
      "value": "decimal"
    }
    ```

Visitor Service - POST /api/v1/visitors - Description: Register a new visitor - Request Body: `json { "name": "string", "visit_date": "datetime", "purpose": "string" }` - Response: 201 Created

- **GET /api/v1/visitors/{id}**
  - Description: Retrieve visitor details by ID
  - Response: 200 OK

  - Response Body:
    ```
    {
      "id": "integer",
      "name": "string",
      "visit_date": "datetime",
      "purpose": "string"
    }
    ```

Report Service - POST /api/v1/reports - Description: Generate a new report - Request Body: `json { "type": "string", "date_range": { "start": "datetime", "end": "datetime" } }` - Response: 201 Created

- **GET /api/v1/reports/{id}**
  - Description: Retrieve report details by ID
  - Response: 200 OK
  - Response Body:
    ```
    {
      "id": "integer",
      "type": "string",
      "date_range": {
        "start": "datetime",
        "end": "datetime"
      },
      "status": "string"
    }
    ```

Notification Service - POST /api/v1/notifications - Description: Send a new notification - Request Body: `json { "recipient_id": "integer", "message": "string", "type": "string" }` - Response: 201 Created

- **GET /api/v1/notifications/{id}**
  - Description: Retrieve notification details by ID
  - Response: 200 OK
  - Response Body:
    ```
    {
      "id": "integer",
      "recipient_id": "integer",
      "message": "string",
      "type": "string",
      "sent_at": "datetime"
    }
    ```

---

6. Data Flow Diagrams

Level 0 DFD: Context Diagram

```
+--------------------+      +----------------------+      +--------------------+
|                    |      |                      |      |                    |
|     Residents      +------>+ Property Management  +------>+    Admins/Staff    |
|                    |      |      Platform        |      |                    |
+---------+----------+      +-----------+----------+      +---------+----------+
          |                             |                           |
          |                             |                           |
          |                             |                           |
          |                             |                           |
          v                             v                           v
+---------+----------+      +-----------+----------+      +---------+----------+
|                    |      |                      |      |                    |
|  External Systems  +------>+     Databases        +<------+  Security Staff   |
|                    |      |                      |      |                    |
+--------------------+      +----------------------+      +--------------------+
```

Level 1 DFD: Detailed Processes

User Management   Detailed Steps:

1. **User Registers:**
   - Residents submit registration forms with personal details.
   - The registration form data is validated.
2. **Validate Data:**
   - Check for completeness and correctness of user data.
   - Ensure no duplicate entries exist.
3. **Store User Info:**
   - Validated data is stored in the User Database.
4. **Send Confirmation Email:**
   - A confirmation email is sent to the user's registered email address.

```
+------------------+    +------------------+    +------------------+    +------------------
|                  |    |                  |    |                  |    |
|  User Registers  +--> + Validate Data    +--> +  Store User Info +--> + Send Confirmation
|                  |    |                  |    |  in User DB      |    |    Email
+--------+---------+    +--------+---------+    +--------+---------+    +--------+---------
         ^                       |                       |                       |
         |                       v                       v                       v
+--------+---------+    +-----------+----------+    +--------+---------+    +--------+-----
|                  |    |                      |    |                  |    |
|    User Submits  +--> +  Process Registration +--> +    User Info     +--> +    Notificat
| Registration Form|    |                      |    |                  |    |
+------------------+    +----------------------+    +------------------+    +-------------
```

Asset Management Detailed Steps:

1. **Add Asset Info:**
   - Community staff add details about new community assets.
   - The asset data is validated.
2. **Validate Asset Data:**
   - Check for completeness and correctness of asset data.
3. **Store Asset Info:**
   - Validated data is stored in the Asset Database.
4. **Update Asset Log:**
   - Any changes or updates to asset information are logged.

```
+------------------+    +------------------+    +------------------+    +------------------
|                  |    |                  |    |                  |    |
|  Add Asset Info  +--> + Validate Asset   +--> +  Store Asset Info+--> + Update Asset Log
|                  |    |     Data         |    |  in Asset DB     |    |
+--------+---------+    +--------+---------+    +--------+---------+    +--------+---------
         ^                           |                           |                   |
         |                           v                           v                   v
+--------+---------+    +-----------+----------+    +--------+---------+    +--------+----
|                  |    |                      |    |                  |    |
|  Community Staff +--> +  Process Asset Data  +--> +  Asset Database  +--> +    Notificati
|                  |    |                      |    |                  |    |
+------------------+    +----------------------+    +------------------+    +-------------
```

Visitor Management Detailed Steps:

1. **Register Visitor:**
   - Visitors provide their details at the entrance.
   - Visitor data is validated.
2. **Validate Visitor Data:**
   - Check for completeness and correctness of visitor data.
3. **Store Visitor Info:**
   - Validated data is stored in the Visitor Database.
4. **Update Visitor Log:**
   - Entry and exit times are logged for each visitor.

```
+------------------+    +------------------+    +------------------+    +------------------
|                  |    |                  |    |                  |    |
| Register Visitor +--> + Validate Visitor +--> + Store Visitor    +--> +  Update Visitor
|                  |    |     Data         |    |  Info in DB      |    |   Log
+--------+---------+    +--------+---------+    +--------+---------+    +--------+---------
         ^                           |                           |                   |
         |                           v                           v                   v
+--------+---------+    +-----------+----------+    +--------+---------+    +--------+----
|                  |    |                      |    |                  |    |
|    Visitor       +--> +  Process Visitor Data+--> +  Visitor Database+--> +    Notificati
|                  |    |                      |    |                  |    |
+------------------+    +----------------------+    +------------------+    +--------------
```

Online Repair Reporting   Detailed Steps:

1. **Submit Repair Request:**
   - Residents submit repair requests with issue details.
   - The request data is validated.
2. **Validate Repair Request:**
   - Check for completeness and correctness of repair request data.
3. **Store Repair Info:**
   - Validated data is stored in the Repair Database.
4. **Update Repair Status:**
   - Maintenance staff update the status of repair requests.
5. **Notify Resident:**
   - Residents are notified of the status updates of their repair requests.

```
+------------------+    +------------------+    +------------------+    +------------------
|                  |    |                  |    |                  |    |
|  Submit Repair   +--> + Validate Repair  +--> + Store Repair     +--> + Notify Resident
|      Request     |    |     Request      |    |  Info in DB      |    |
+--------+---------+    +--------+---------+    +--------+---------+    +--------+---------
         ^                       |                       |                       |
         |                       v                       v                       v
+--------+---------+    +-----------+----------+    +--------+---------+    +--------+----
|                  |    |                      |    |                  |    |
|  Resident        +--> +  Process Repair Data +--> +  Repair Database +--> +   Maintenance
|                  |    |                      |    |                  |    |     Staff
+------------------+    +----------------------+    +------------------+    +--------------
```

Complaint Management   Detailed Steps:

1. **Submit Complaint:**
   - Residents file complaints with details of the issues.
   - Complaint data is validated.
2. **Validate Complaint Data:**
   - Check for completeness and correctness of complaint data.
3. **Store Complaint Info:**
   - Validated data is stored in the Complaint Database.
4. **Review Complaint:**
   - Customer service reviews and processes the complaint.
5. **Resolve Complaint:**
   - Actions are taken to resolve the complaint.
   - The resolution is logged and the resident is notified.

```
+------------------+    +------------------+    +------------------+    +------------------
|                  |    |                  |    |                  |    |
|  Submit Complaint+--> + Validate Complaint+--> + Store Complaint +--> + Notify Resident
|                  |    |     Data          |    |  Info in DB     |    |
+--------+---------+    +--------+---------+    +--------+---------+    +--------+--------
         ^                          |                          |                  |
         |                          v                          v                  v
+--------+---------+    +-----------+----------+    +--------+---------+    +--------+----
|                  |    |                      |    |                  |    |
|  Resident        +--> +  Process Complaint   +--> +  Complaint DB    +--> + Customer Servi
|                  |    |                      |    |                  |    |
+------------------+    +----------------------+    +------------------+    +-------------
```

Reporting   Detailed Steps:

1. **Generate Report:**
   - Admins initiate the report generation process.
   - Report generation request data is validated.
2. **Validate Request Data:**
   - Check for completeness and correctness of report request data.
3. **Store Report Info:**
   - Generated report data is stored in the Report Database.
4. **Notify Admin:**
   - Admins are notified when reports are ready for viewing or export.

```
+------------------+    +------------------+    +------------------+    +-----------------
|                  |    |                  |    |                  |    |
| Generate Report  +--> + Validate Request +--> + Store Report     +--> + Notify Admin
|                  |    |     Data         |    |  Info in DB      |    |
+--------+---------+    +--------+---------+    +--------+---------+    +--------+--------
         ^                          |                          |                  |
         |


v                          v                   v
+--------+---------+    +-----------+----------+    +--------+---------+    +--------+----
|                  |    |                      |    |                  |    |
|    Admin         +--> +  Process Report Data +--> +  Report Database +--> + Generate Repo
|                  |    |                      |    |                  |    |
+------------------+    +----------------------+    +------------------+    +-------------
```

Notification Service   Detailed Steps:

1. **Create Notification:**
   - Admins or system processes create notification messages.
   - Notification data is validated.
2. **Validate Data:**
   - Check for completeness and correctness of notification data.
3. **Store Notification Info:**
   - Validated notification data is stored in the Notification Database.

4. **Send Notification:**
    - Notifications are sent to the intended recipients (residents, staff, etc.).

```
+-------------------+    +-------------------+    +-------------------+    +------------------
|                   |    |                   |    |                   |    |
| Create Notification+--> + Validate Data   +--> + Store Notification+--> + Send Notificati
|                   |    |                   |    |  Info in DB       |    |
+--------+---------+    +--------+---------+    +--------+---------+    +--------+--------
         ^                        |                        |                    |
         |                        v                        v                    v
+--------+---------+    +-----------+----------+    +--------+---------+    +--------+----
|                   |    |                       |    |                   |    |
|  User/Admin      +--> +  Process Notification+--> + Notification DB  +--> +     Users
|                   |    |                       |    |                   |    |
+-------------------+    +-----------------------+    +-------------------+    +------------
```

7. Message Queue

The message queue system is crucial for handling asynchronous communication between various microservices within the Property Management Platform. It ensures reliable message delivery, decouples microservices, and enhances system scalability and resilience.

Overview **Purpose:** - To facilitate asynchronous communication between different services. - To decouple service dependencies. - To handle high throughput and ensure reliable message delivery.

Components: - **Message Broker:** Manages the message queues. - **Producers:** Services that send messages. - **Consumers:** Services that receive and process messages.

Technology: RabbitMQ (chosen for its robustness and extensive support in the industry)

Key Message Queues

1. **User Registration Queue:**
    - **Producer:** User Service
    - **Consumer:** Email Notification Service
    - **Purpose:** To send a confirmation email after a user registers.
2. **Asset Management Queue:**
    - **Producer:** Asset Service
    - **Consumer:** Reporting Service
    - **Purpose:** To update reports when new assets are added or updated.
3. **Visitor Management Queue:**
    - **Producer:** Visitor Service
    - **Consumer:** Security Notification Service
    - **Purpose:** To notify security staff about visitor arrivals.

4. **Repair Request Queue:**
    - **Producer:** Repair Service
    - **Consumer:** Maintenance Staff Notification Service
    - **Purpose:** To inform maintenance staff about new repair requests.
5. **Complaint Management Queue:**
    - **Producer:** Complaint Service
    - **Consumer:** Customer Service Notification Service
    - **Purpose:** To notify customer service about new complaints filed by residents.
6. **Reporting Queue:**
    - **Producer:** Various Services (User, Asset, Visitor, etc.)
    - **Consumer:** Reporting Service
    - **Purpose:** To aggregate data for generating reports.
7. **Notification Queue:**
    - **Producer:** Various Services (User, Asset, Repair, Complaint, etc.)
    - **Consumer:** Notification Service
    - **Purpose:** To send notifications to users, admins, and staff about various events and updates.

Detailed Steps for Each Queue

User Registration Queue Detailed Steps:

1. **User Registers:**
    - User submits registration details.
2. **User Service Publishes Message:**
    - User Service validates data and publishes a message to the User Registration Queue.
3. **Email Notification Service Consumes Message:**
    - Email Notification Service retrieves the message from the queue.
4. **Send Confirmation Email:**
    - Email Notification Service sends a confirmation email to the user.

```
User Registers --> User Service Publishes Message --> Message Broker (User Registration Queu
```

Asset Management Queue Detailed Steps:

1. **Add/Update Asset Info:**
    - Community staff add or update asset information.
2. **Asset Service Publishes Message:**
    - Asset Service validates data and publishes a message to the Asset Management Queue.
3. **Reporting Service Consumes Message:**
    - Reporting Service retrieves the message from the queue.
4. **Update Reports:**
    - Reporting Service updates asset-related reports.

```
Add/Update Asset Info --> Asset Service Publishes Message --> Message Broker (Asset Manageme
```

Visitor Management Queue Detailed Steps:

1. **Register Visitor:**
   - Visitor provides details at the entrance.
2. **Visitor Service Publishes Message:**
   - Visitor Service validates data and publishes a message to the Visitor Management Queue.
3. **Security Notification Service Consumes Message:**
   - Security Notification Service retrieves the message from the queue.
4. **Notify Security Staff:**
   - Security Notification Service sends notifications to security staff about visitor arrivals.

```
Register Visitor --> Visitor Service Publishes Message --> Message Broker (Visitor Manageme
```

Repair Request Queue Detailed Steps:

1. **Submit Repair Request:**
   - Resident submits a repair request.
2. **Repair Service Publishes Message:**
   - Repair Service validates data and publishes a message to the Repair Request Queue.
3. **Maintenance Staff Notification Service Consumes Message:**
   - Maintenance Staff Notification Service retrieves the message from the queue.
4. **Notify Maintenance Staff:**
   - Maintenance Staff Notification Service notifies maintenance staff about the repair request.

```
Submit Repair Request --> Repair Service Publishes Message --> Message Broker (Repair Reques
```

Complaint Management Queue Detailed Steps:

1. **Submit Complaint:**
   - Resident files a complaint.
2. **Complaint Service Publishes Message:**
   - Complaint Service validates data and publishes a message to the Complaint Management Queue.
3. **Customer Service Notification Service Consumes Message:**
   - Customer Service Notification Service retrieves the message from the queue.
4. **Notify Customer Service:**
   - Customer Service Notification Service notifies customer service staff about the new complaint.

```
Submit Complaint --> Complaint Service Publishes Message --> Message Broker (Complaint Manag
```

Reporting Queue   Detailed Steps:

1. **Data Changes:**
   - Various services trigger data changes (User, Asset, Visitor, etc.).
2. **Service Publishes Message:**
   - Relevant service publishes a message to the Reporting Queue.
3. **Reporting Service Consumes Message:**
   - Reporting Service retrieves the message from the queue.
4. **Update Reports:**
   - Reporting Service updates reports based on the new data.

```
Data Changes --> Service Publishes Message --> Message Broker (Reporting Queue) --> Reportin
```

Notification Queue   Detailed Steps:

1. **Event Triggers:**
   - Various events (User registration, asset update, repair request, etc.) occur.
2. **Service Publishes Message:**
   - Relevant service publishes a message to the Notification Queue.
3. **Notification Service Consumes Message:**
   - Notification Service retrieves the message from the queue.
4. **Send Notification:**
   - Notification Service sends notifications to the intended recipients.

```
Event Triggers --> Service Publishes Message --> Message Broker (Notification Queue) --> Not
```

Benefits of Using Message Queues

1. **Decoupling:** Services can operate independently without needing to wait for each other's processes.
2. **Scalability:** The system can handle increased load by scaling the message queue infrastructure.
3. **Reliability:** Ensures messages are delivered even if some services are temporarily down.
4. **Flexibility:** New services can be added to consume messages from existing queues without impacting the producers.
5. **Resilience:** Provides fault tolerance, as messages are persisted in the queue until successfully processed.

8. Technical Challenges

Developing the Property Management Platform for CBRE Group, Inc. involves several technical challenges that need to be addressed to ensure a robust, scalable, and efficient system. Here are the primary challenges and the strategies to overcome them:

1. Scalability Challenge: - The platform needs to handle a growing number of users, properties, and transactions efficiently without compromising performance.

Solution: - Implement a microservices architecture to break down the system into smaller, independent services. - Use load balancers and auto-scaling groups to manage and distribute traffic. - Employ a distributed database system to handle large volumes of data.

2. Data Consistency Challenge: - Ensuring data consistency across different services and databases, especially in a distributed system.

Solution: - Use distributed transactions and eventual consistency models where appropriate. - Implement strong data validation and conflict resolution mechanisms. - Employ tools like Kafka or RabbitMQ for reliable message queuing and processing.

3. Security Challenge: - Protecting sensitive data and ensuring secure access control across the platform.

Solution: - Implement role-based access control (RBAC) to manage permissions. - Use encryption for data at rest and in transit. - Regularly update and patch the system to mitigate vulnerabilities.

4. Real-Time Data Processing Challenge: - Providing real-time updates and notifications to users.

Solution: - Use WebSocket or Server-Sent Events (SSE) for real-time communication. - Implement efficient caching mechanisms to reduce latency. - Utilize message queues (e.g., RabbitMQ) for real-time event handling and processing.

5. Integration with Third-Party Services Challenge: - Integrating the platform with various third-party services (e.g., payment gateways, email services, SMS services).

Solution: - Use standard APIs and SDKs provided by third-party services. - Implement an API gateway to manage and route requests to third-party services. - Ensure robust error handling and retry mechanisms for third-party API calls.

6. High Availability and Disaster Recovery Challenge: - Ensuring the platform remains available and operational during failures or disasters.

Solution: - Deploy the system across multiple availability zones and regions. - Implement database replication and automated backups. - Use monitoring and alerting tools to detect and respond to failures quickly.

7. Performance Optimization Challenge: - Maintaining optimal performance under high load conditions.

Solution: - Conduct regular performance testing and profiling. - Optimize database queries and indexing. - Use content delivery networks (CDNs) and edge caching for faster content delivery.

8. User Experience Challenge: - Providing a seamless and intuitive user experience.

Solution: - Conduct user testing and gather feedback to improve the user interface (UI) and user experience (UX). - Implement responsive design to support various devices and screen sizes. - Ensure fast load times and smooth navigation.

9. Regulatory Compliance Challenge: - Ensuring compliance with various legal and regulatory requirements (e.g., GDPR, HIPAA).

Solution: - Implement data protection and privacy policies. - Regularly review and update compliance procedures. - Work with legal experts to ensure the platform adheres to relevant regulations.

10. Legacy System Integration Challenge: - Integrating with existing legacy systems that CBRE Group, Inc. might be using.

Solution: - Use middleware or adapters to interface with legacy systems. - Plan a phased migration strategy to gradually move away from legacy systems. - Ensure backward compatibility during the transition period.

Strategies for Overcoming Challenges

1. **Adopt Agile Methodologies:**
    - Use agile development practices to iterate quickly and adapt to changing requirements.
    - Conduct regular sprints and reviews to ensure continuous improvement.
2. **Continuous Integration and Continuous Deployment (CI/CD):**
    - Implement CI/CD pipelines for automated testing and deployment.
    - Use tools like Jenkins, GitLab CI, or Travis CI for managing the CI/CD processes.
3. **Comprehensive Monitoring and Logging:**
    - Deploy monitoring tools like Prometheus, Grafana, and ELK stack for real-time monitoring and logging.
    - Set up alerts and dashboards to proactively manage system health and performance.
4. **Collaborative Development:**
    - Foster collaboration between development, operations, and security teams (DevOps and DevSecOps).

- Use tools like Jira, Confluence, and Slack for effective communication and project management.

By addressing these technical challenges with robust solutions and strategies, the Property Management Platform can achieve high performance, reliability, and user satisfaction, ensuring it meets the needs of CBRE Group, Inc. and its users effectively.

9. AWS Integration

Integrating AWS (Amazon Web Services) into the Property Management Platform provides a scalable, reliable, and cost-effective infrastructure. AWS offers various services that can be leveraged to meet the requirements of the platform, ensuring high availability, security, and performance.

Key AWS Services Utilized

1. **Amazon EC2 (Elastic Compute Cloud)**
   - **Purpose:** To provide scalable virtual servers for running the application.
   - **Integration:**
     - Use Auto Scaling groups to automatically adjust the number of EC2 instances based on demand.
     - Implement Elastic Load Balancing (ELB) to distribute incoming traffic across multiple instances for fault tolerance.
2. **Amazon RDS (Relational Database Service)**
   - **Purpose:** To manage relational databases in the cloud.
   - **Integration:**
     - Deploy databases like MySQL, PostgreSQL, or SQL Server.
     - Enable Multi-AZ (Availability Zone) deployments for high availability and automated backups for disaster recovery.
3. **Amazon S3 (Simple Storage Service)**
   - **Purpose:** To store and retrieve any amount of data at any time.
   - **Integration:**
     - Use S3 buckets to store user-uploaded files, reports, and backups.
     - Implement lifecycle policies to manage storage costs by transitioning objects to different storage classes.
4. **Amazon SNS (Simple Notification Service)**
   - **Purpose:** To send notifications and alerts.
   - **Integration:**
     - Use SNS to send email, SMS, or push notifications to users and admins for various events (e.g., new visitor arrivals, maintenance updates).
5. **Amazon SQS (Simple Queue Service)**
   - **Purpose:** To decouple and scale microservices, distributed systems, and serverless applications.
   - **Integration:**

    - Use SQS for message queuing between services to ensure reliable communication and processing.
    - Implement SQS for handling asynchronous tasks like sending notifications, processing reports, etc.

6. **AWS Lambda**
    - **Purpose:** To run code without provisioning or managing servers.
    - **Integration:**
        - Use Lambda functions for event-driven processing (e.g., triggering a function on file upload to S3, database changes).
        - Implement Lambda for lightweight background processing tasks.
7. **Amazon CloudFront**
    - **Purpose:** To deliver content with low latency and high transfer speeds.
    - **Integration:**
        - Use CloudFront as a content delivery network (CDN) to cache and deliver static content like images, CSS, and JavaScript files.
        - Implement SSL/TLS for secure data transfer.
8. **Amazon API Gateway**
    - **Purpose:** To create, publish, maintain, monitor, and secure APIs.
    - **Integration:**
        - Use API Gateway to expose RESTful APIs for various services.
        - Implement authentication and authorization mechanisms using AWS Cognito.
9. **Amazon CloudWatch**
    - **Purpose:** To monitor and manage operational health.
    - **Integration:**
        - Use CloudWatch for logging, monitoring, and alerting.
        - Set up CloudWatch Alarms to notify administrators of potential issues (e.g., high CPU usage, memory leaks).
10. **AWS IAM (Identity and Access Management)**
    - **Purpose:** To manage access to AWS services and resources securely.
    - **Integration:**
        - Define IAM roles and policies to control access to AWS resources.
        - Implement multi-factor authentication (MFA) for enhanced security.
11. **AWS Elastic Beanstalk**
    - **Purpose:** To deploy and manage applications in the AWS Cloud.
    - **Integration:**
        - Use Elastic Beanstalk for simplified deployment and scaling of web applications and services.
        - Manage the underlying infrastructure and focus on application code.

Integration Architecture

User Authentication and Management **Components:** - AWS Cognito - API Gateway - Lambda

Flow: 1. User authentication requests are directed to AWS Cognito via API Gateway. 2. Cognito handles user authentication and returns tokens. 3. API Gateway routes authenticated requests to appropriate Lambda functions or microservices.

Data Storage and Retrieval **Components:** - RDS - S3 - Lambda

Flow: 1. Application data is stored in RDS with read replicas for high availability. 2. User-generated content and backups are stored in S3. 3. Lambda functions trigger on specific events (e.g., new data in S3) to process or move data.

Notification Service **Components:** - SNS - SQS - Lambda

Flow: 1. Events from various services publish messages to SNS topics. 2. SNS topics fan out messages to SQS queues subscribed to them. 3. Lambda functions or other services consume messages from SQS and process notifications.

Monitoring and Logging **Components:** - CloudWatch - Lambda

Flow: 1. Application and infrastructure logs are sent to CloudWatch. 2. CloudWatch Alarms trigger Lambda functions or notifications when predefined thresholds are met.

Security and Compliance **Strategies:** - Implement IAM roles and policies to restrict access. - Use VPC (Virtual Private Cloud) to isolate network resources. - Encrypt data at rest (using AWS KMS) and in transit (using SSL/TLS). - Regularly review and update security groups and firewall settings. - Conduct regular security audits and compliance checks.

Benefits of AWS Integration

1. **Scalability:** Easily scale infrastructure up or down based on demand.
2. **Reliability:** Leverage AWS's global infrastructure for high availability and disaster recovery.
3. **Cost-Effectiveness:** Use pay-as-you-go pricing models to optimize costs.
4. **Security:** Benefit from AWS's robust security features and compliance certifications.
5. **Performance:** Ensure low latency and high performance with services like CloudFront and RDS read replicas.
6. **Flexibility:** Quickly deploy and manage resources using Elastic Beanstalk, Lambda, and other services.

By integrating these AWS services, the Property Management Platform can achieve its goals of scalability, reliability, security, and performance, ensuring a robust solution for CBRE Group, Inc.

Deployment Strategy

Deploying the Property Management Platform efficiently and reliably involves a well-planned strategy that ensures minimal downtime, seamless updates, and robust rollback mechanisms. The deployment strategy leverages AWS services to automate and streamline the deployment process.

Key Components of the Deployment Strategy

1. **CI/CD Pipeline**
   - **Tools:** AWS CodePipeline, AWS CodeBuild, AWS CodeDeploy, GitHub
   - **Flow:**
     1. Developers commit code to a GitHub repository.
     2. CodePipeline detects the change and triggers the build process in CodeBuild.
     3. CodeBuild compiles the code, runs tests, and creates deployment artifacts.
     4. CodeDeploy deploys the artifacts to the target environments (e.g., development, staging, production).
2. **Infrastructure as Code (IaC)**
   - **Tools:** AWS CloudFormation, AWS CDK (Cloud Development Kit), Terraform
   - **Flow:**
     1. Define infrastructure resources (e.g., EC2 instances, RDS databases, S3 buckets) in code using CloudFormation templates or Terraform scripts.
     2. Version control the infrastructure code in a Git repository.
     3. Use CloudFormation or Terraform to provision and manage infrastructure resources consistently across environments.
3. **Blue/Green Deployment**
   - **Purpose:** To minimize downtime and reduce risks during deployment.
   - **Flow:**
     1. Deploy the new version of the application to a separate environment (green).
     2. Run tests and validate the new version in the green environment.
     3. Gradually shift traffic from the old version (blue) to the new version (green) using AWS Elastic Load Balancing.
     4. Monitor the new version for any issues.
     5. Fully switch to the new version and decommission the old environment if everything is stable.
4. **Canary Deployment**
   - **Purpose:** To test new releases with a subset of users before a full rollout.
   - **Flow:**
     1. Deploy the new version to a small subset of instances or users

   (canary).
   2. Monitor the performance and gather feedback from the canary deployment.
   3. Gradually increase the number of users or instances running the new version if no issues are detected.
   4. Fully deploy the new version to all users once confidence is established.

5. **Rolling Updates**
   - **Purpose:** To update the application incrementally with no downtime.
   - **Flow:**
     1. Deploy the new version to a few instances at a time.
     2. Ensure each instance is healthy before proceeding to the next set of instances.
     3. Continue the process until all instances are running the new version.
     4. Roll back to the previous version if any issues

10. Additional Project Insights

Multithreading Usage

In the Property Management Platform, we used multithreading to handle concurrent processing of visitor check-ins and asset status updates. This ensured that our system could efficiently manage multiple operations simultaneously without performance degradation.

Builder Design Pattern

The Builder design pattern was employed in the Report Service to construct complex report objects. This pattern helped in managing the creation process, especially when reports required multiple optional parameters and complex configurations.

Team Composition

As the team leader, I oversaw a team of six members: - **Backend Developers (2):** Focused on developing and maintaining the microservices. - **Frontend Developer (1):** Developed the user interface and ensured seamless user experience. - **QA Engineer (1):** Responsible for testing the platform and ensuring high-quality releases. - **DevOps Engineer (1):** Managed our CI/CD pipeline, AWS infrastructure, and ensured system reliability. - **Product Manager (1):** Coordinated requirements and worked with stakeholders to define features.

Jenkins Pipeline Design

Our Jenkins pipeline for AWS included the following stages: 1. **Code Checkout:** Retrieve the latest code from the repository. 2. **Build:** Compile the code and run unit tests. 3. **Docker Build:** Build Docker images for each microservice. 4. **Security Scan:** Perform security checks on the Docker images. 5. **Deploy to Staging:** Deploy the Docker images to the staging environment on AWS ECS. 6. **Integration Tests:** Run integration tests in the staging environment. 7. **Deploy to Production:** Upon successful tests, deploy the Docker images to the production environment on AWS ECS. 8. **Notification:** Send deployment notifications to the team via Slack.

Monitoring Strategy

Our monitoring setup included: - **AWS CloudWatch:** Used for monitoring the health and performance of our AWS resources. - **Prometheus and Grafana:** Implemented for detailed application metrics and dashboard visualizations. - **ELK Stack:** Utilized for log aggregation, searching, and analysis.

Usage Metrics

- **Daily Users:** Approximately 2,000 active users daily.
- **Transactions Per Second (TPS):** Average TPS of 50 during peak hours.
- **Queries Per Second (QPS):** Average QPS of 100 for our database.

Frontend Story

Frontend Development Story

Technologies Used

- **Framework:** React.js
- **State Management:** Redux
- **Styling:** SCSS
- **API Requests:** Axios
- **Code Quality:** ESLint, Prettier
- **Testing:** Jest, React Testing Library

Development Process

1. **Planning and Design**
    - Collaborated with UI/UX designers to create wireframes and mockups.
    - Defined the application's components and their interactions.
2. **Setting Up the Project**
    - Initialized a new React project using Create React App.
    - Configured Redux for state management.
3. **Implementing Components**
    - Developed reusable components for the application's UI elements.
    - Integrated Redux for managing state across the application.
4. **API Integration**
    - Used Axios to make API requests to the backend services.

    - Implemented logic to handle API responses and update the UI accordingly.
5. **Styling**
    - Used SCSS for styling to maintain a modular and scalable design.
    - Ensured consistency in UI elements across the application.
6. **Testing**
    - Wrote unit tests using Jest and React Testing Library for components and Redux actions/reducers.
    - Conducted integration tests to ensure the components work together as expected.
7. **Deployment**
    - Configured deployment settings for different environments (development, staging, production).
    - Used AWS Amplify or similar services for continuous deployment.
8. **Performance Optimization**
    - Implemented lazy loading for components and images to improve initial loading times.
    - Minimized bundle size by code splitting and tree shaking.
9. **Accessibility**
    - Ensured the application is accessible to users with disabilities by following WCAG guidelines.
    - Used ARIA attributes and semantic HTML elements appropriately.
10. **Code Reviews and Refactoring**
    - Conducted regular code reviews to maintain code quality and consistency.
    - Refactored code to improve performance, readability, and maintainability.
11. **Collaboration**
    - Used version control (Git) and collaborated with backend developers, QA engineers, and designers using tools like GitHub and Slack.
    - Communicated regularly to ensure alignment with project requirements and timelines.

Future Improvements

- **Enhanced User Experience:** Implement more interactive features and animations to improve user engagement.
- **Optimized Performance:** Further optimize performance by analyzing and addressing bottlenecks.
- **Internationalization:** Add support for multiple languages to make the application accessible to a broader audience.
- **Progressive Web App (PWA):** Convert the application into a PWA for offline access and improved performance on mobile devices.
- **Continuous Improvement:** Continuously gather user feedback and iterate on the application to meet evolving user needs.