Functional Documentation

Mohammed Nawaz, Shaik Aseeruddin

November 15, 2024

Contents

1 **Scope**

This document describes the structure and design of a community notification system built using microservices. The project aims to replace traditional communication methods in residential communities with a digital platform, enabling residents and admins to share and access important updates, maintenance tasks, and payments conveniently. Each function is divided into a separate microservice, allowing independent development and operation. These microservices communicate via RESTful APIs for seamless integration.

2 **Functional Requirements**

The system's main functional requirements include:

- **Authentication and User Management**: Users (both residents and admins) should be able to sign up, log in, log out, and manage their profiles.
- **Notification Management**: Admins should post community notifications, while residents view these updates.
- **Maintenance Task Management**: Admins post details about maintenance tasks, such as repair schedules or utility outages, and residents view task statuses.
- **Payment Management**: Residents can view maintenance fees and make payments through an integrated payment system.
- **Feedback Management**: Residents can submit feedback about services, and admins can view and respond.

3 **Non-Functional Requirements**

The non-functional requirements include:

- **Performance**: The system should handle up to 1,000 concurrent users with a response time under 200 milliseconds.
- **Scalability**: The design should support horizontal scaling as more users join.
- **Security**: All data must be encrypted in transit and at rest, with JWT for authentication.
- **Availability**: The system should achieve 99.9% uptime with reliable fallback options.
- **Maintainability**: Code should follow SOLID principles, ensuring high modularity and loose coupling for easy updates and troubleshooting.

4 **Technologies Used**

The system uses:

- **Programming Language**: Java 17 for backend services.
- **Frameworks**:
    - Spring Boot for building microservices.
    - Spring Security for authentication and authorization.
    - Spring Cloud Netflix Eureka for service discovery.
    - Spring Cloud Gateway for API routing.

- **Database**: MySQL or MongoDB for data storage.
- **Frontend**: Angular for a responsive interface.

5 **High-Level Design**

5.1 **Architectural Overview**

This system is based on a microservices architecture, where each service focuses on a specific function. Each microservice operates independently, with REST APIs enabling communication. The core components include:

- **API Gateway**: Serves as a single entry point for all requests, routing them to appropriate microservices.
- **Authentication Service**: Manages resident and admin authentication with JWT.
- **User Service**: Handles user-related operations, such as registration, login, profile management, and role assignments.
- **Notification Service**: Allows admins to create notifications for residents.
- **Maintenance Task Service**: Manages and updates details related to community maintenance.
- **Payment Service**: Manages payment processing using Razorpay API integration.
- **Feedback Service**: Captures feedback submitted by residents and provides an interface for admin responses.

5.2 **Functional Specifications**

Each component has specific responsibilities. An example sequence diagram is provided below to illustrate a typical notification posting workflow:

Sequence Diagram: Posting a Notification

1. Admin logs in and accesses the notification dashboard.
2. Admin creates and submits a notification.
3. Notification Service saves the notification to the database.
4. Residents view the notification via their dashboard.

Sequence Diagram: Resident Login

1. Resident submits login credentials.
2. Authentication Service validates the credentials.
3. If valid, a JWT token is generated and returned.
4. Resident uses the token for accessing other services.

5.3 **Performance Considerations**

The system is optimized for performance with several key practices:

- **Caching**: Notifications and other frequently accessed data are cached.

- **Load Balancing**: Requests are balanced across instances for even load distribution.
- **Asynchronous Processing**: Non-critical tasks, like sending feedback confirmations, are processed asynchronously to enhance responsiveness.
- **Database Optimization**: Indexing is implemented on commonly queried fields.

5.4 **Technology Stack**

The technology stack includes:

- **Backend**: Java, Spring Boot, Spring Security.
- **Frontend**: Angular, Bootstrap, HTML, CSS.
- **Database**: MySQL or MongoDB.
- **Containerization**: Docker for consistent deployment.
- **Service Discovery**: Spring Cloud Netflix Eureka.
- **API Gateway**: Spring Cloud Gateway.
- **CI/CD**: Jenkins and GitHub Actions for automated integration and deployment.

6 **Low-Level Design**

6.1 **Detailed Component Specification**

Each microservice is composed of controllers, services, and repositories:

- **Controller**: Exposes RESTful endpoints.
- **Service**: Contains business logic and manages interactions with repositories.
- **Repository**: Manages data persistence and retrieval.

For example, in the Notification Service:

- **NotificationController**: Handles API requests for notifications.
- **NotificationService**: Contains logic for posting, viewing, and managing notifications.
- **NotificationRepository**: Interacts with the database to manage notification data.

6.2 **Interface Definitions**

Each service exposes a RESTful API that other services or the frontend can consume. The endpoints include:

- **Authentication Service API**:
  - `POST /api/auth/register`: Registers a new user.
  - `POST /api/auth/login`: Authenticates a user and returns a JWT.
- **Notification Service API**:
  - `POST /api/notifications`: Creates a new notification.
  - `GET /api/notifications`: Retrieves all notifications for viewing.
- **Maintenance Task Service API**:
  - `POST /api/maintenance/tasks`: Adds a new maintenance task.
  - `GET /api/maintenance/tasks`: Lists all maintenance tasks.

- **Payment Service API**:
  - `POST /api/payments`: Processes a new payment.
  - `GET /api/payments/status`: Checks the status of a payment.

6.3 **Resource Allocation**

Resource distribution is optimized based on anticipated load for each microservice:

- **Notification Service**: Allocated extra memory to handle frequent data retrieval.
- **Payment Service**: Allocated higher CPU for processing secure transactions.

6.4 **Error Handling and Recovery**

Error management strategies include:

- **Try-Catch Blocks**: To handle exceptions gracefully.
- **Fallback Mechanisms**: To provide continuity during service failures.
- **Logging**: All errors are logged for monitoring and debugging.
- **Circuit Breaker Pattern**: Prevents cascading failures if a service goes down.

6.5 **Security Considerations**

Security practices include:

- **Authentication**: JWT for secure authentication.
- **Authorization**: Role-based access control to limit resources based on user roles.
- **Data Encryption**: Ensures sensitive data is encrypted in transit and at rest.
- **Input Validation**: Prevents injection attacks by validating all inputs.

6.6 **Code Structure**

Each microservice has a clean structure with modular components:

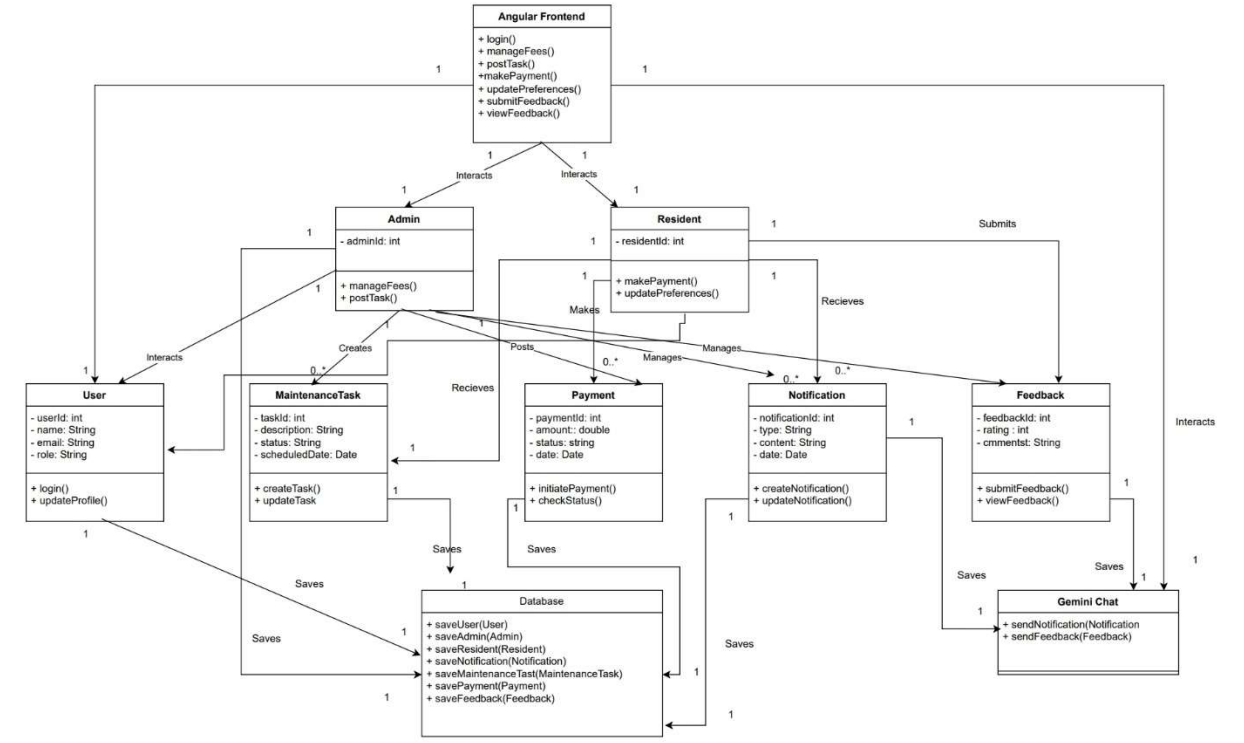
- **Controller Layer**: Manages HTTP requests.
- **Service Layer**: Contains business logic.
- **Repository Layer**: Manages data operations.
- **DTOs**: Encapsulates data for transfer between layers.
- **Utils**: Common utilities for various services.

7 **Diagrams**

7.1 **ER Diagrams**

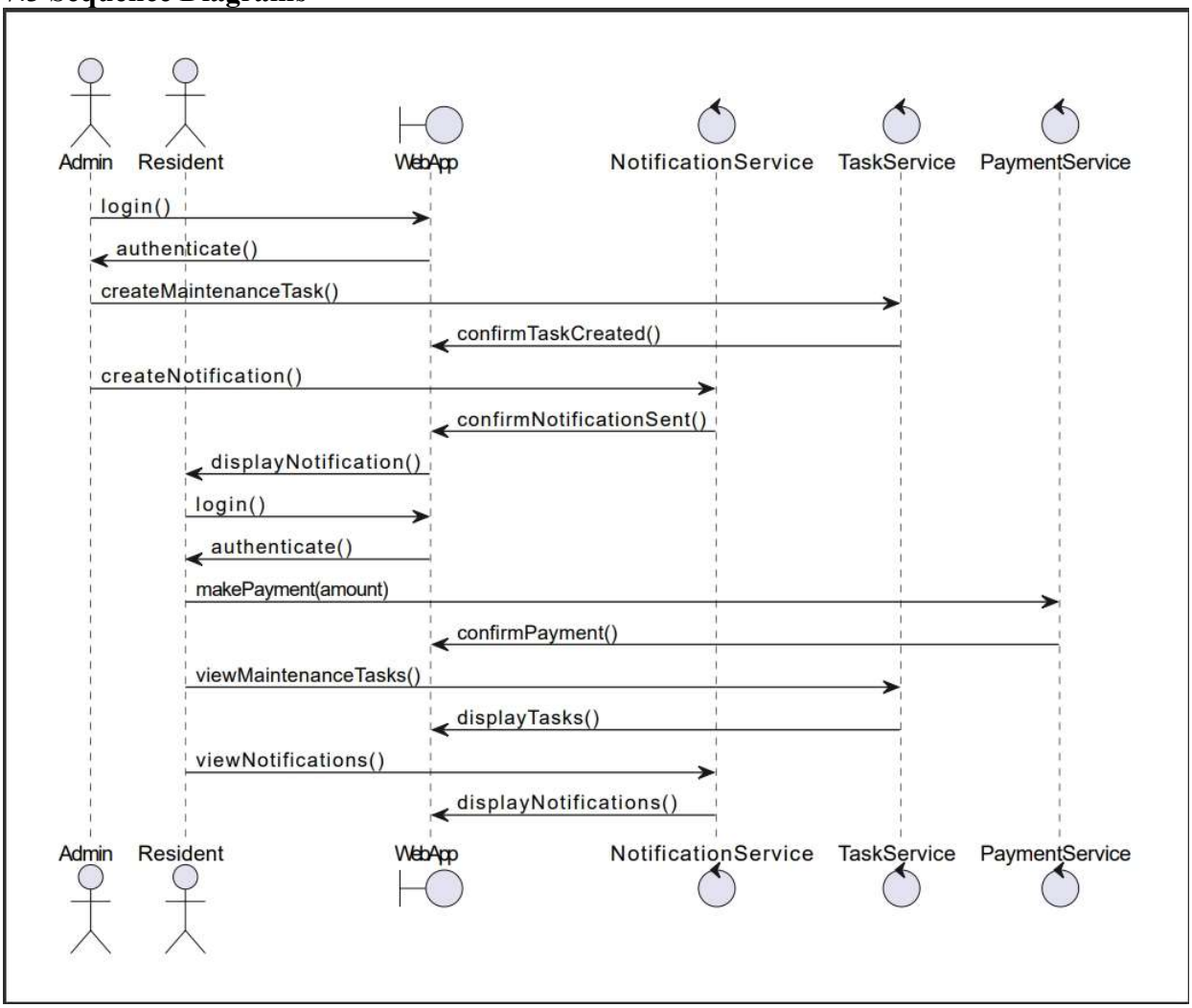
Shows the relationships among entities such as User, Notification, MaintenanceTask, Payment, and Feedback.

7.2 **Class Diagram**



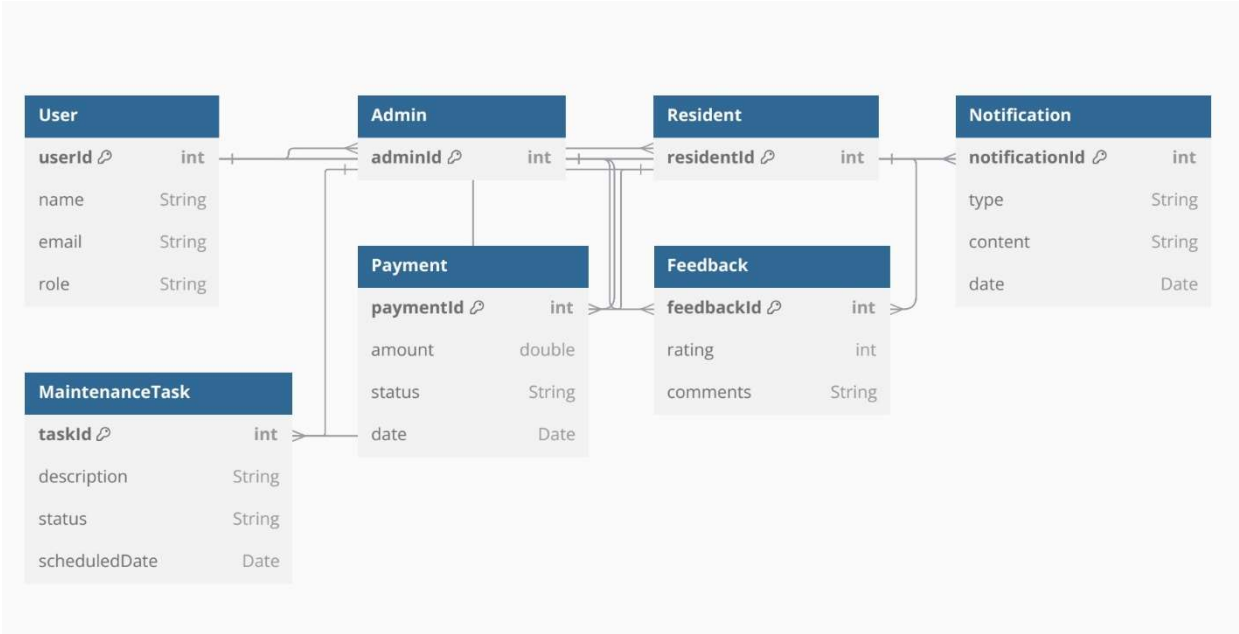
Depicts the class structure of each microservice, showing attributes, methods, and relationships.

7.3 **Sequence Diagrams**



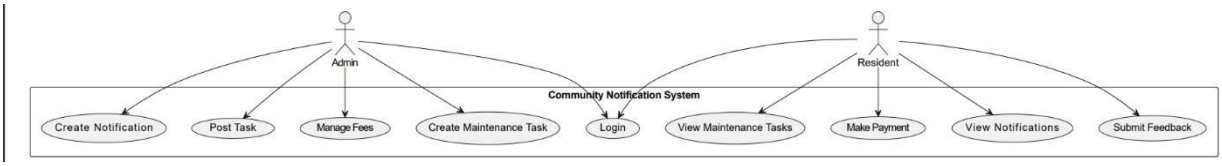
Illustrates the workflow for processes, such as posting a notification and user authentication.

7.4 **Deployment Diagram**



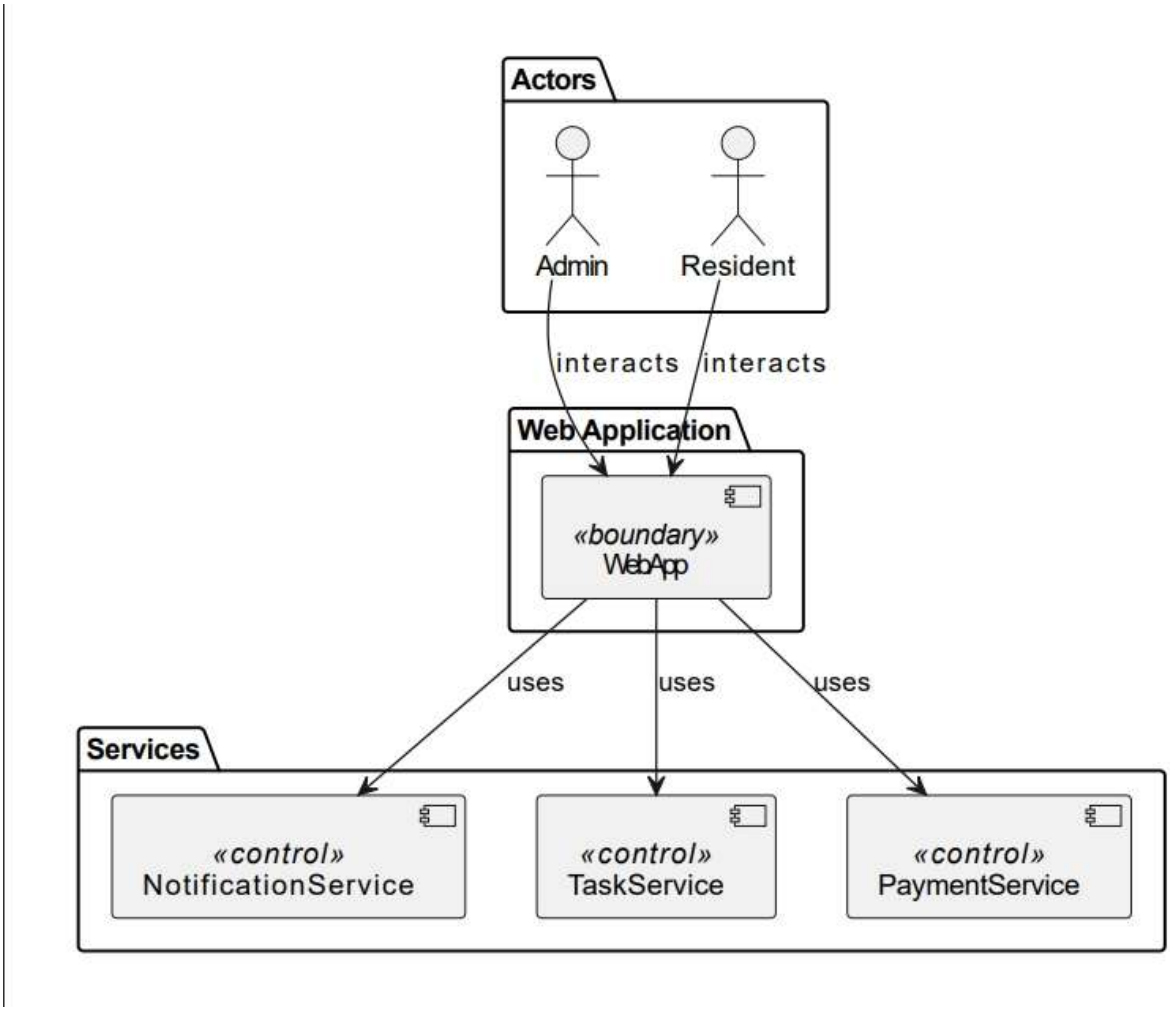
Depicts the system's physical deployment setup, including servers, applications, and other components.

7.5 **Use Case Diagram**



Visualizes interactions between users and the system.

7.6 **Package Diagram**



Shows the grouping of different components, making the system's dependencies clear.

8 **Conclusion**

This microservices-based system is scalable, maintainable, and secure, covering the core needs of a community notification platform. By separating functionality into services like Authentication, Notification, Maintenance, Payment, and Feedback, the system can grow and adapt over time. The backend is powered by Spring Boot and MySQL/MongoDB, while Angular provides a dynamic user interface. The architecture supports scalability with Docker and JWT ensures secure user interactions. This system's modular design is prepared to accommodate future updates and new features, providing efficient and reliable communication for residential communities.