CAPSTONE PROJECT

Digital Insurance Management System

Team Members :

1. Sai Chakradhar Rao Mahendrakar (ZE2916)
2. Swarnim Srijan (ZE2923)
3. Sai Teja Ambati (ZE2908)
4. Arya Makadia (ZE2910)
5. Sharath T N (ZE2920)
6. Tushar Raikar (ZE2924)

Date : 22 August 2025

Organization : Better World Technology Pvt. Ltd.

zeta

upGrad

Table Of Contents

Abstract

The Digital Insurance Management System is a comprehensive, full-stack application designed to modernize and automate the core processes of insurance administration. Leveraging a robust backend built with Java and Spring Boot, the system ensures scalable and secure management of insurance policies, claims, and customer information. PostgreSQL serves as the primary database, providing reliable data storage and efficient querying capabilities. The backend is complemented by a dynamic frontend developed using Vue.js and TypeScript, delivering a responsive and intuitive user experience for both administrators and customers.

The project employs Docker for containerization, enabling seamless deployment and consistent environments across development, testing, and production stages. With Docker Compose, developers can easily orchestrate multiple services, including the application server and database, simplifying setup and maintenance. The use of Maven and npm streamlines dependency management and build processes for the backend and frontend, respectively.

Security is a key focus, with authentication and authorization mechanisms in place to protect sensitive data and ensure that only authorized users can access specific features. The system is designed to handle complex insurance workflows, from policy issuance and renewal to claims processing and customer support. Its modular architecture allows for easy integration with third-party services and future scalability.

Overall, the Digital Insurance Management System provides a unified platform that enhances operational efficiency, reduces manual effort, and improves customer satisfaction. By embracing modern technologies and best practices, it offers a reliable and future-proof solution for digital insurance management.

Introduction

The Digital Insurance Management System is a modern, full-stack application engineered to transform the way insurance operations are managed and delivered. In an industry where efficiency, accuracy, and customer satisfaction are paramount, this system leverages a suite of contemporary technologies to automate and streamline the end-to-end processes involved in insurance administration.

At its core, the backend is developed using Java and the Spring Boot framework, which together provide a robust, scalable, and secure foundation for business logic and data management. Spring Boot's modularity and extensive ecosystem enable rapid development and integration of features such as policy management, claims processing, and customer data handling. The backend communicates with a PostgreSQL database, chosen for its reliability, performance, and advanced querying capabilities, ensuring that all insurance data is stored securely and can be accessed efficiently.

The frontend is built with Vue.js and TypeScript, offering a responsive, dynamic, and user-friendly interface. This combination allows for the creation of interactive dashboards and forms that cater to both administrators and end-users, making complex insurance workflows intuitive and accessible. TypeScript's static typing enhances code quality and maintainability, while Vue.js's component-based architecture supports rapid UI development and scalability.

To facilitate seamless development, deployment, and scaling, the project utilizes Docker for containerization. Docker ensures that the application and its dependencies run consistently across different environments, from local development to production. The use of Docker Compose further simplifies orchestration, allowing multiple services—such as the application server and database—to be managed together with minimal configuration. This approach not only accelerates onboarding for new developers but also reduces the risk of environment-specific issues.

Build and dependency management are handled by Maven for the backend and npm for the frontend, streamlining the process of integrating third-party libraries and ensuring reproducible

builds. This setup supports continuous integration and delivery pipelines, enabling rapid iteration and deployment of new features.

Security is a fundamental aspect of the system. Authentication and authorization mechanisms are implemented to safeguard sensitive information and ensure that only authorized users can access specific functionalities. The architecture is designed to comply with industry best practices, supporting data privacy and regulatory requirements.

The Digital Insurance Management System is engineered to handle the full lifecycle of insurance operations, from policy issuance and renewals to claims submission and resolution. Its modular design allows for easy integration with external services, such as payment gateways and third-party verification systems, and supports future expansion as business needs evolve.

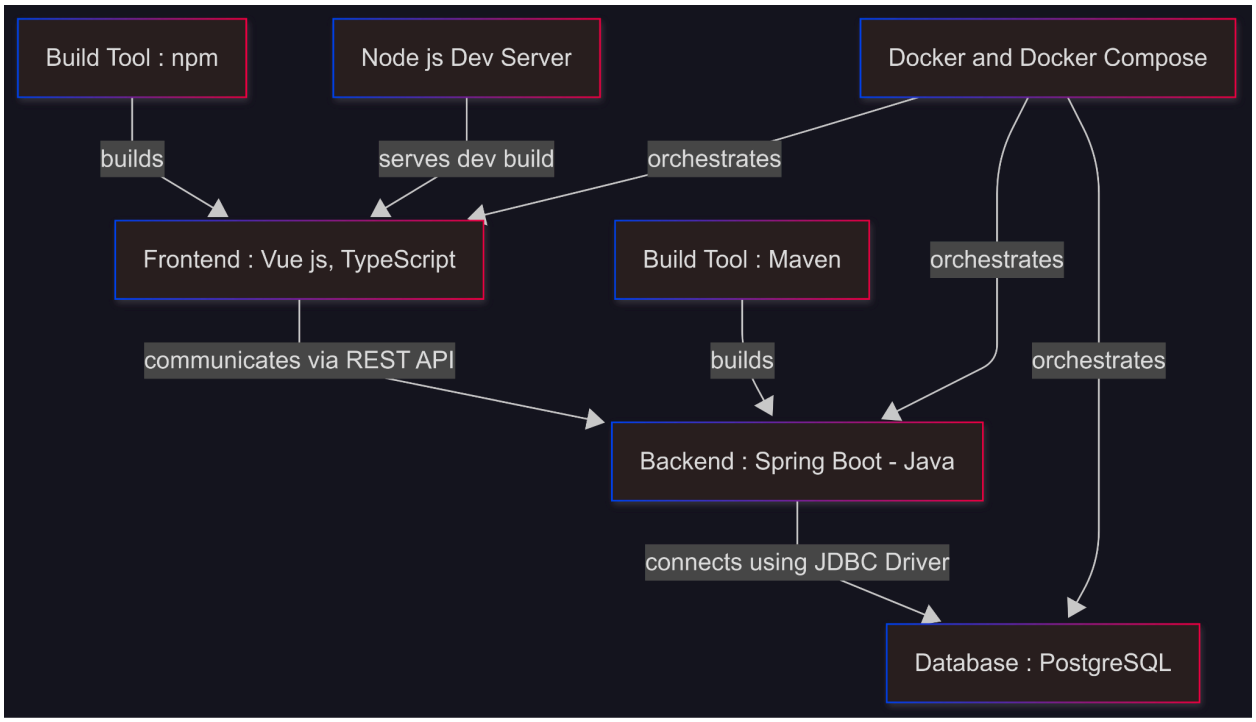
In summary, this project delivers a comprehensive, future-proof solution for digital insurance management. By integrating proven backend and frontend technologies, containerization, and best practices in security and deployment, it empowers insurance providers to enhance operational efficiency, reduce manual workloads, and deliver superior service to their customers. The result is a unified platform that not only meets the current demands of digital insurance but is also adaptable to the changing landscape of the industry.

System Architecture

High Level Description :

The Digital Insurance Management System is a full-stack web application designed to automate and streamline insurance operations. It features a backend built with Java and Spring Boot, using PostgreSQL for secure and scalable data storage. The frontend is developed with Vue.js and TypeScript, providing a responsive and user-friendly interface for both administrators and customers. The system supports essential insurance workflows, including policy management, claims processing, and customer data handling. Docker is used for containerized deployment, ensuring consistent environments across development and production. The architecture is modular and extensible, allowing for easy integration with third-party services and future scalability. Security, efficiency, and user experience are central to the system's design, making it a comprehensive solution for digital insurance administration.

Architecture Diagram



- The frontend is built with Vue.js and TypeScript, using npm as the build tool and Node.js Dev Server for development.
- The backend is developed with Spring Boot (Java), built using Maven.
- The frontend communicates with the backend via REST API.
- The backend connects to the PostgreSQL database using a JDBC driver.
- Docker and Docker Compose orchestrate the frontend, backend, and database services, enabling easy deployment and management.

Technology Stack

Technology Stack for Digital Insurance Management System:

- Frontend:
    - Vue.js
    - TypeScript
    - JavaScript
    - npm (build tool)
    - Node.js (development server)
- Backend:
    - Java
    - Spring Boot
    - Maven (build tool)
- Database:
    - PostgreSQL
- Containerization & Orchestration:
    - Docker
    - Docker Compose
- Other:
    - SQL (for database migrations and queries)

Requirements

Functional Requirements :

Functional Requirements (User Stories / Features) for Digital Insurance Management System:

- User Registration and Authentication
    - Enables secure registration and login for users to access insurance accounts.
- Policy Management
    - Allows users to view, purchase, renew, and cancel insurance policies for comprehensive coverage management.
- Claims Management
    - Provides functionality to file, view, and track insurance claims for timely support.
- Admin Dashboard
    - Grants administrators the ability to manage users, policies, and claims for effective system oversight.
- Notifications
    - Delivers notifications regarding policy renewals, claim updates, and payment reminders to users.
- Password Management
    - Permits users to change passwords for account maintenance.
- Search and Filter
    - Enables efficient search and filtering of policies, claims, and users for quick access.
- Role-Based Access Control
    - Restricts feature access based on user roles (user, admin) to ensure security and proper authorization.

Non-Functional Requirements

Non-Functional Requirements (as present in the project):

- Containerization and Portability
    - The system is containerized using Docker, enabling deployment on any environment supporting Docker.

- Modular Architecture
    - The codebase is organized into frontend and backend modules, supporting maintainability and scalability.
- Documentation
    - Setup and usage instructions are provided in README.md files for both frontend and backend.
- Database Initialization
    - Automated database setup is available via Docker Compose and initialization scripts.
- Testing
    - Unit tests written for all Controllers, Services and Repositories.
- Configuration Management
    - Environment-specific configurations are managed through properties files and Docker Compose.
- Code Quality
    - Linting and type checking are set up for the frontend (ESLint, TypeScript).
- Separation of Concerns
    - Clear separation between user and admin workflows in both frontend and backend.
- Basic Security
    - Role-based access control is implemented in the application logic.
- Scalability (Basic)
    - The architecture allows for independent scaling of frontend, backend, and database services via Docker Compose.

Hardware Requirements:

- CPU: At least a dual-core processor (Intel i3, AMD equivalent, or better)
- RAM: Minimum 4 GB (8 GB recommended for smoother performance)
- Storage: At least 2 GB of free disk space (more may be needed for Docker images and database data)
- Display: Minimum resolution of 1366x768

Software Requirements:

- Operating System (any one of the following):
- Windows 10 or 11 (64-bit)
- macOS 11 (Big Sur) or later
- Ubuntu 20.04 or newer (or other recent Linux distributions)

Backend Requirements:

- Java JDK 17 (OpenJDK or Oracle)
- Maven 3.6 or later (for building and managing dependencies)
- PostgreSQL 13 or later (or another supported relational database, if configured)
- Docker 20.10 or later (optional, for containerized deployments)

Frontend Requirements:

- Node.js 18.x LTS or later
- npm 9.x or later (usually included with Node.js)
- Modern web browser (latest version of Chrome, Firefox, Edge, or Safari)

Default Ports Used:

- Backend API: 8080
- Frontend development server: 5173
- Database (PostgreSQL): 5432

Design

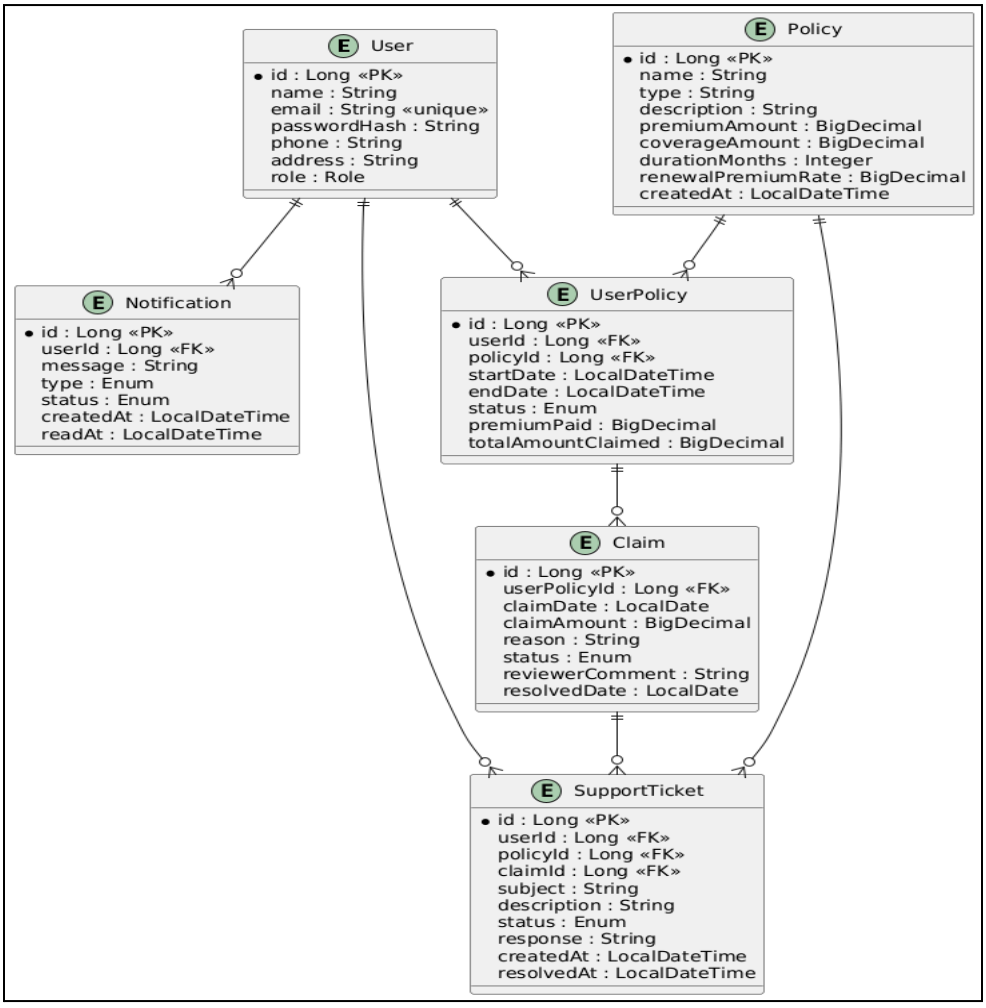
System Design Details

System Design Details and Decisions

- Architecture
    - Type: Modular, containerized, multi-tier architecture.
    - Layers:
        - Presentation (Frontend: Vue.js + TypeScript)
        - Application (Backend: Spring Boot, Java)
        - Data (PostgreSQL)
- Frontend
    - Framework: Vue.js with TypeScript for type safety and maintainability.
    - State Management: Pinia/Vuex stores for modular state handling.
    - Routing: Vue Router for SPA navigation.
    - Styling: Tailwind CSS for utility-first, responsive design.
    - Testing: Playwright for end-to-end tests.
    - Build Tool: Vite for fast development and optimized builds.
- Backend
    - Framework: Spring Boot (Java) for RESTful API development.
    - Dependency Management: Maven.
    - Security:
        - JWT-based authentication.
        - Role-based access control (USER, ADMIN).
    - Configuration: Externalized via application-docker.properties for environment flexibility.
- Database
    - Type: PostgreSQL, chosen for reliability and ACID compliance.
    - Initialization: Automated with SQL scripts (init-db.sql) via Docker Compose.
- Containerization & Deployment
    - Tool: Docker for consistent environment setup.

    - Orchestration: Docker Compose to manage frontend, backend, and database services.
    - Profiles: Support for migrations and environment-specific builds.
- Scalability & Maintainability
    - Separation of Concerns: Clear split between frontend, backend, and database.
    - Modularity: Components, stores, and services are organized by feature.
    - Stateless Backend: Enables horizontal scaling.
- Security Decisions
    - Transport Security: Enforced via HTTPS (configurable at deployment).
    - Sensitive Data: JWT for session management, password management features.
- Documentation & Usability
    - README.md: Setup, usage, and operational instructions for all modules.
    - User/Admin Workflows: Separate views and components for each role.

Database Schema



- User Entity:
    - Represents system users, storing details like name, email (unique), hashed password, phone, address, and role.
- Policy Entity:
    - Defines insurance policies with attributes such as name, type, description, premium, coverage amount, duration, renewal rate, and creation date.
- UserPolicy Entity:
    - Links users to policies they have purchased, recording payment, coverage period, claim totals, and current status.

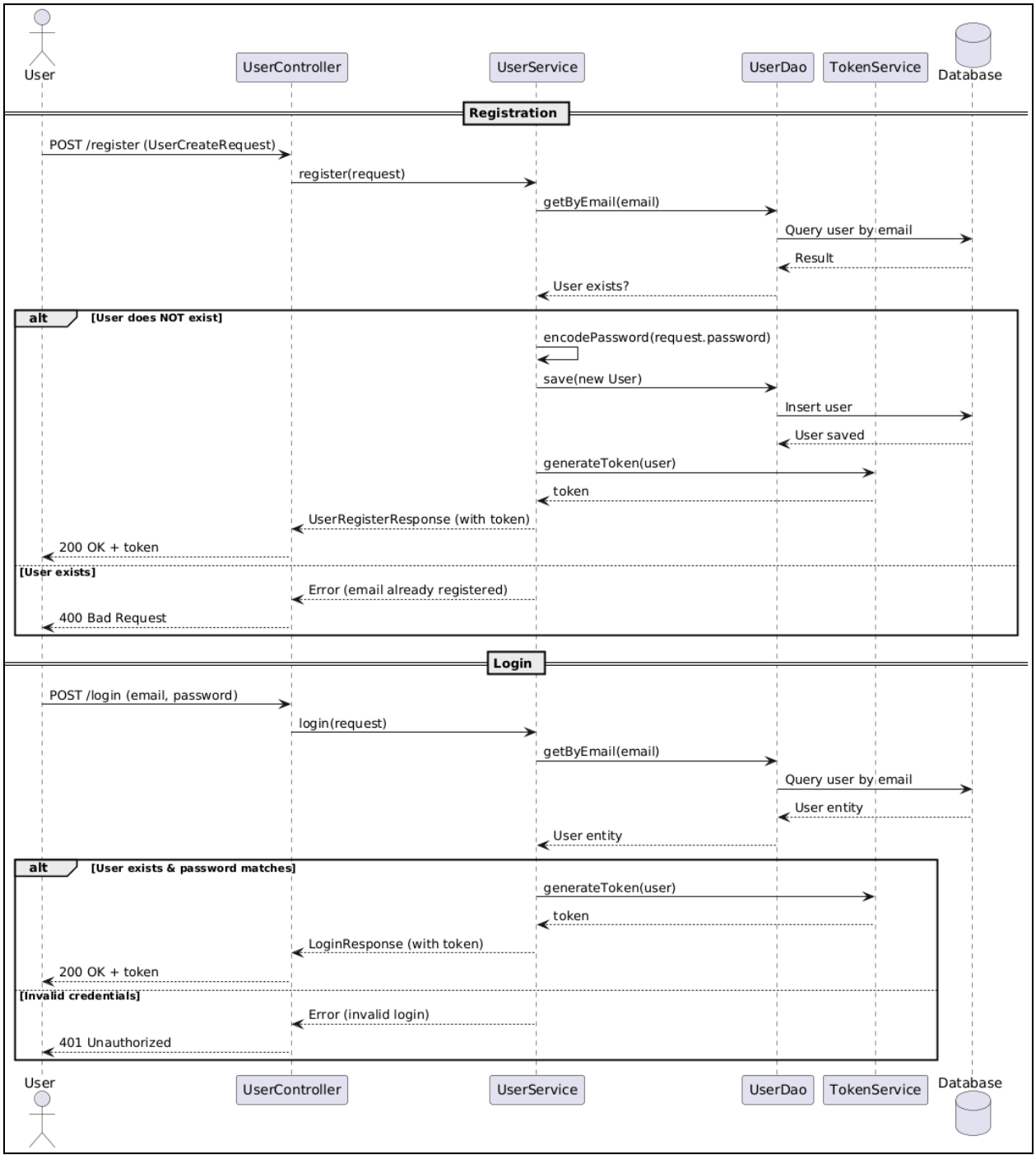
- Claim Entity:
    - Represents insurance claims made by users on their policies, detailing claim date, amount, reason, status, reviewer comments, and resolution date.
- SupportTicket Entity:
    - Allows users to raise support tickets related to policies or claims, including subject, description, status, response, and timestamps for creation and resolution.
- Notification Entity:
    - Stores messages sent to users (such as policy updates or claim statuses), including type, status, and timestamps for creation and when read.

Relationships:

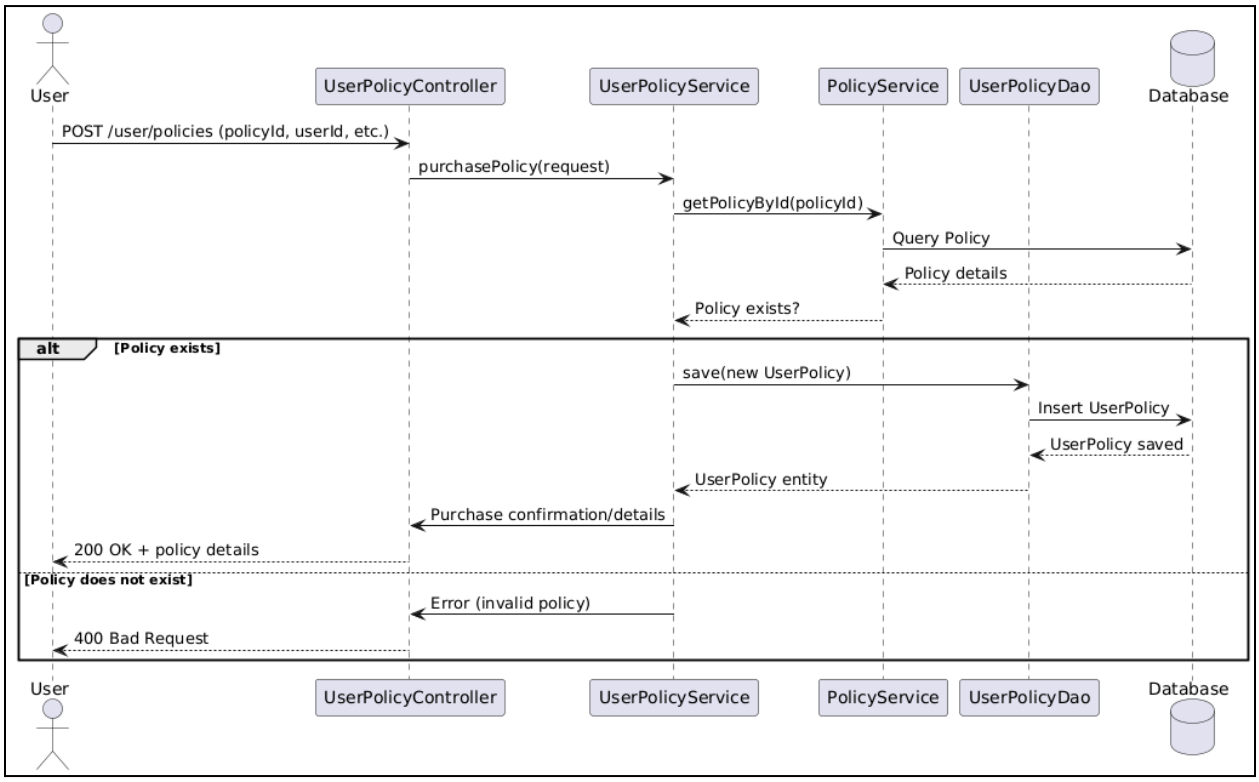
- Users can have multiple policies, notifications, claims, and support tickets.
- Policies can be purchased by multiple users.
- Claims are linked to specific user-policy relationships.
- Support tickets can reference both policies and claims.
- Data Integrity: The use of primary keys (PK) and foreign keys (FK) ensures referential integrity between users, policies, claims, notifications, and support tickets.

Sequence Diagrams

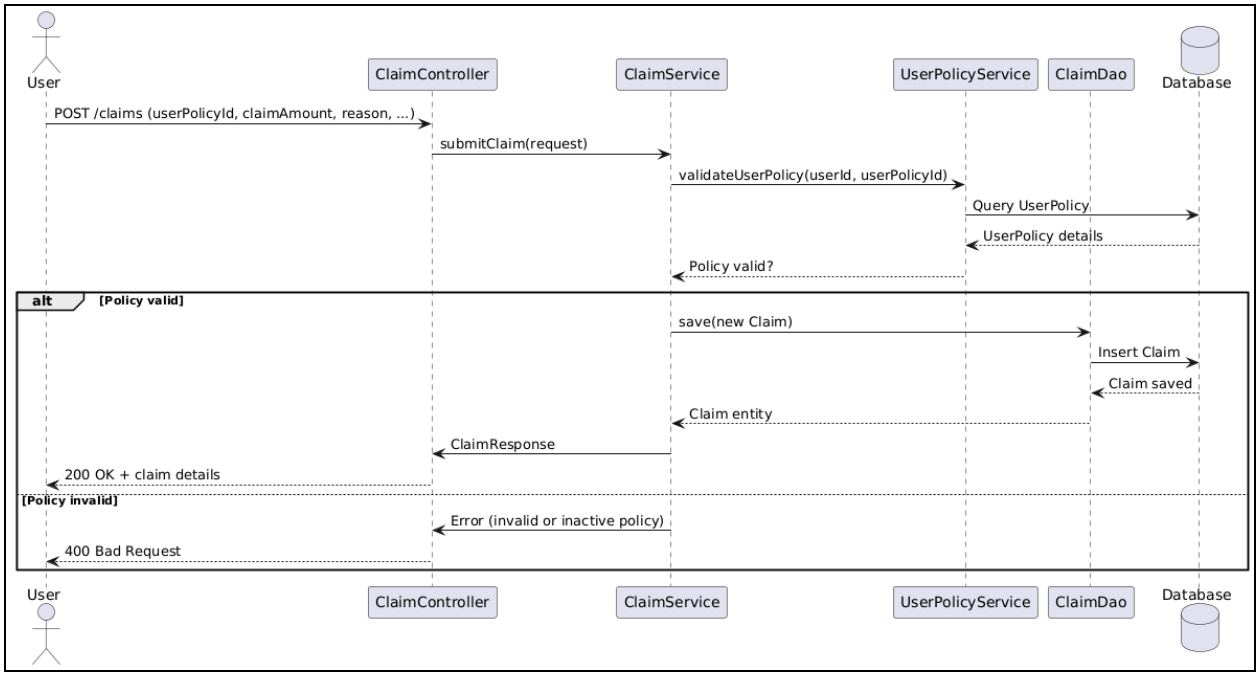
1. Register and Login Sequence



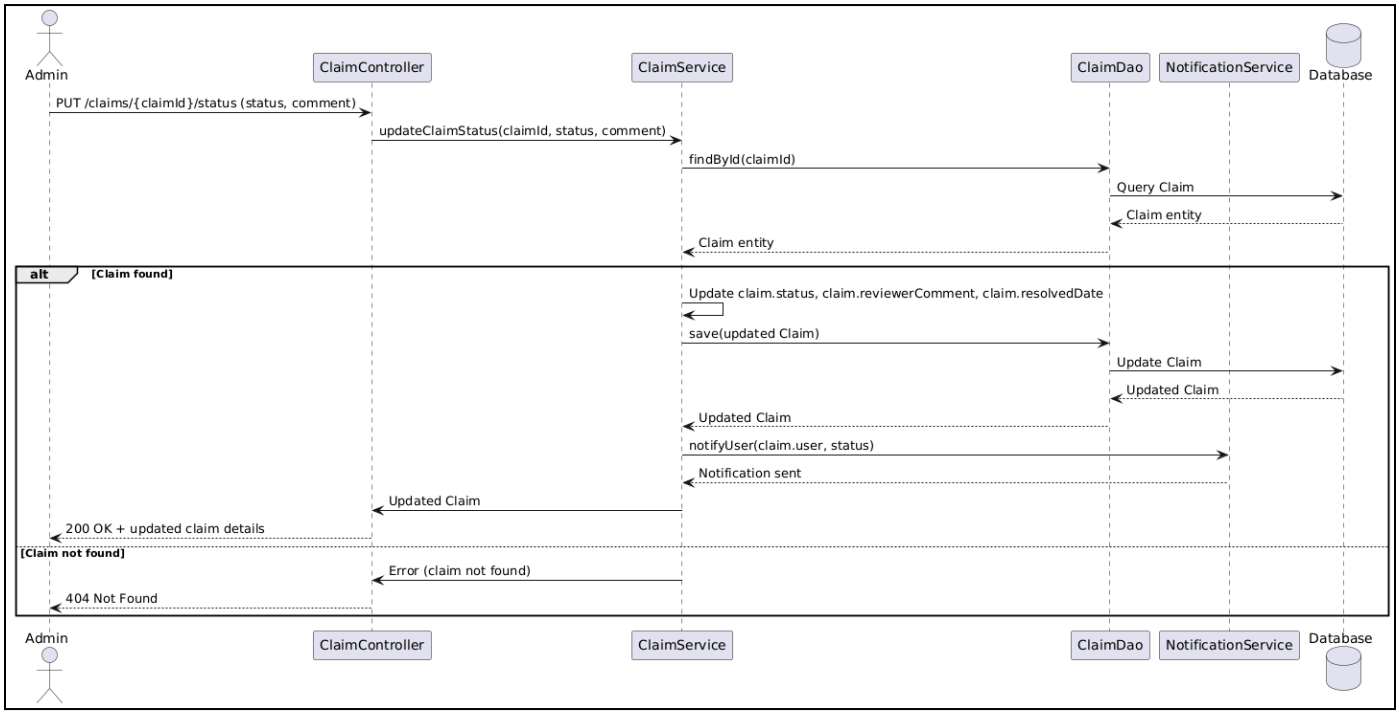
2. Policy purchase Sequence Diagram



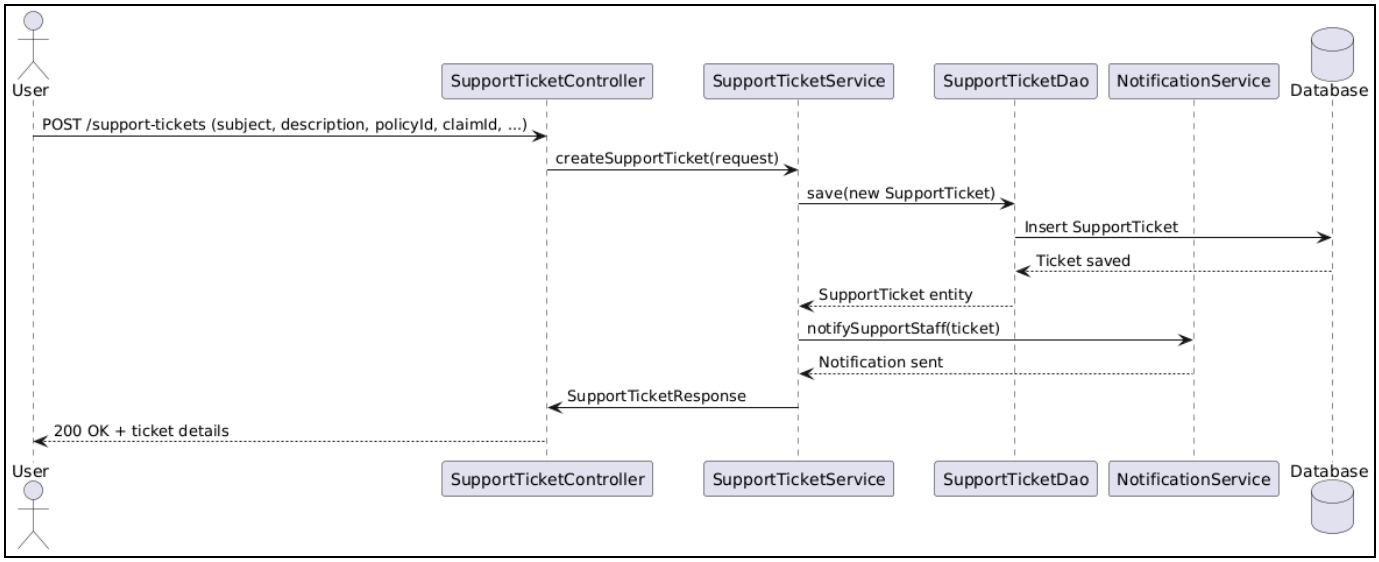
3. Claim Submission Sequence Diagram



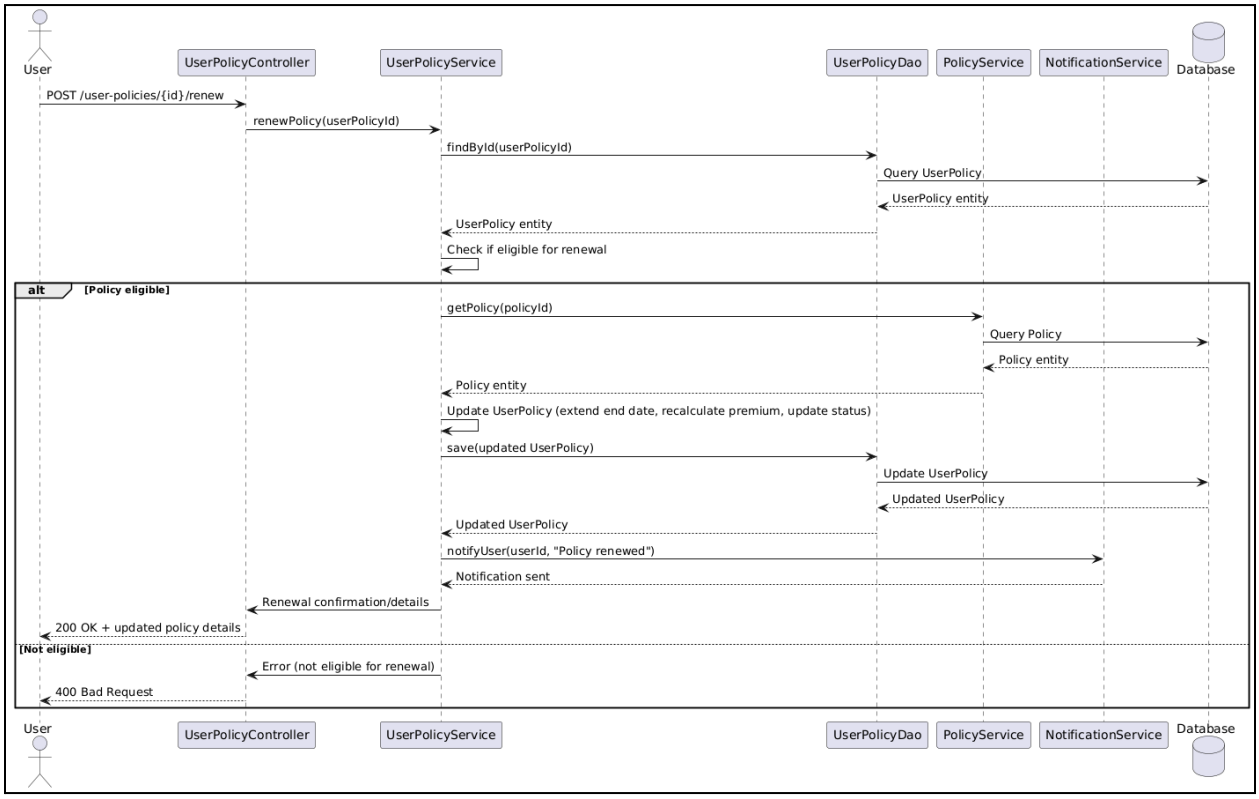
4. Approve/Reject Claim Sequence Diagram



5. Support Ticket

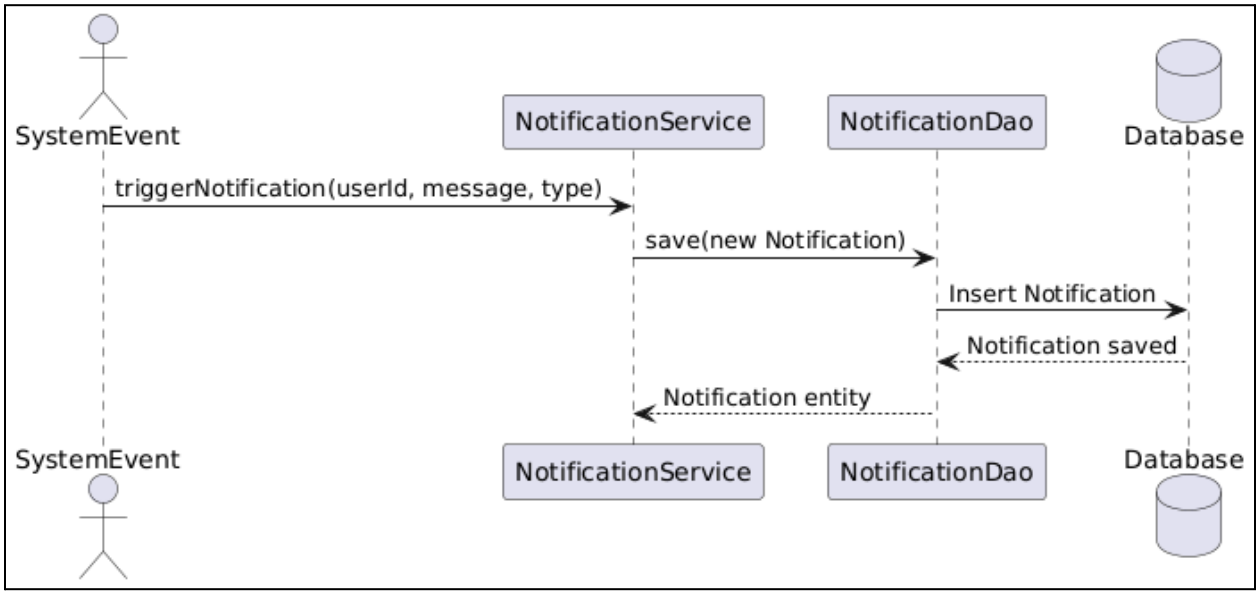


6. Renew Policy Sequence Diagram

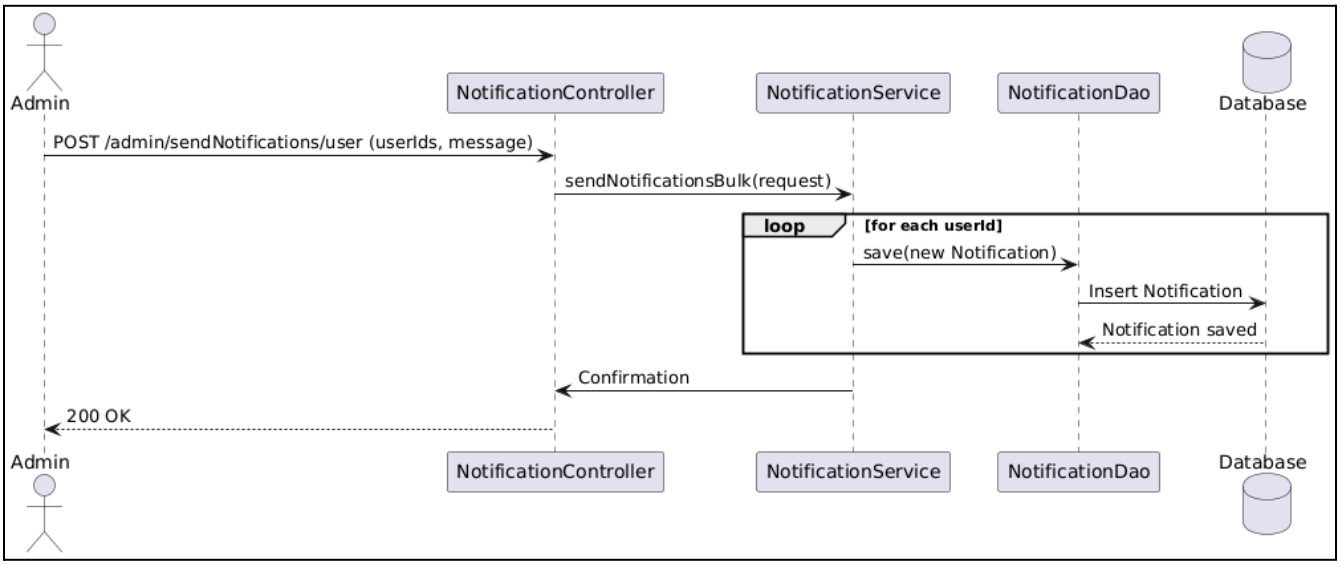


7. Notification

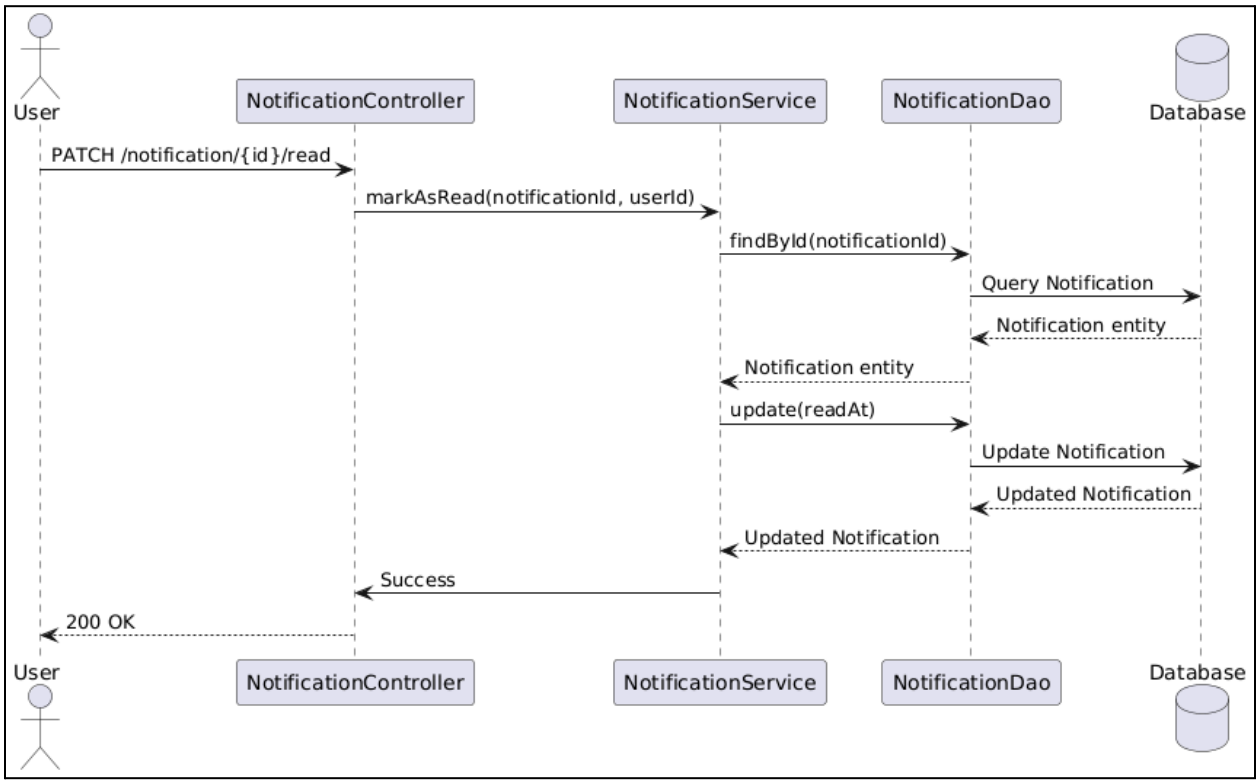
a. Trigger



b. Admin Sends Bulk Notifications



c. User Reads Notification



Implementation

Backend Implementation Details

Spring Boot Modules :

The backend is organized into modular packages, typically following:

- controller: REST API controllers
- service: Business logic and core services
- repository: Data access (Spring Data JPA)
- model or entity: JPA entities
- config: Configuration classes (security, CORS, etc.)

File structure :

Key Dependencies

- Spring Boot Starter Web
- Spring Boot Starter Data JPA
- Spring Boot Starter Security
- Database connector (e.g., PostgreSQL/MySQL)

Key Classes and Services :

- Controllers
    - Handle incoming HTTP requests, map endpoints, and delegate to services.
- Services
    - Contain business logic, orchestrate repository calls, and may include transactional boundaries.
- Repositories
    - Interface with the database using JPA repositories.
- Security Configuration
    - Defines authentication (JWT), authorization, and CORS rules.
- DTOs (Data Transfer Objects)
    - Used to transfer data between client and server, and to decouple internal models from API contracts.

Backend File Structure :

```
backend/
├── insurance/
│   ├── .idea/
│   ├── .mvn/
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   │   └── com/
│   │   │   │       └── innov8ors/
│   │   │   │           └── insurance/
│   │   │   │               ├── config/
│   │   │   │               ├── controller/
│   │   │   │               ├── entity/
│   │   │   │               ├── enums/
│   │   │   │               ├── error/
│   │   │   │               ├── exception/
│   │   │   │               ├── mapper/
│   │   │   │               ├── repository/
│   │   │   │               ├── request/
│   │   │   │               ├── response/
│   │   │   │               ├── service/
│   │   │   │               ├── util/
│   │   │   │               └── InsuranceApplication.java
│   │   │   └── resources/
│   │   └── test/
│   ├── target/
│   ├── .gitattributes
│   ├── .gitignore
│   ├── mvnw
│   └── mvnw.cmd
```

Frontend Implementation Details

Vue.js Structure

Directory Layout

- src/
    - components/: Reusable Vue components (buttons, modals, etc.)
    - views/: Page-level components (mapped to routes)
    - router/: Vue Router configuration
    - store/: Vuex store (state management)
    - assets/: Static files (images, styles)
    - services/: API interaction and business logic
    - App.vue: Root component
    - main.js: Entry point

Key Components

- Main Layout Components :
    - Header, Sidebar, Footer, etc.
- Page Components :
    - Dashboard, Authentication (Login/Register), CRUD pages for main entities.
- Reusable Components :
    - Forms, Dialogs, Data Tables, etc.
- Service Layer :
    - Services for API interaction.

Integration Approach

Communication Flow

Frontend ↔ Backend

- Communicates via RESTful APIs (typically JSON over HTTP).
- Authentication often handled by JWT tokens stored in localStorage or cookies.

Backend ↔ Database

- Uses Spring Data JPA to interact with the relational database.
- Transactions and queries managed via repository interfaces.

Dockerization

Dockerfiles

Backend Dockerfile :

- Based on OpenJDK image
- Copies and builds the Spring Boot JAR
- Exposes port (e.g., 8080)

Frontend Dockerfile :

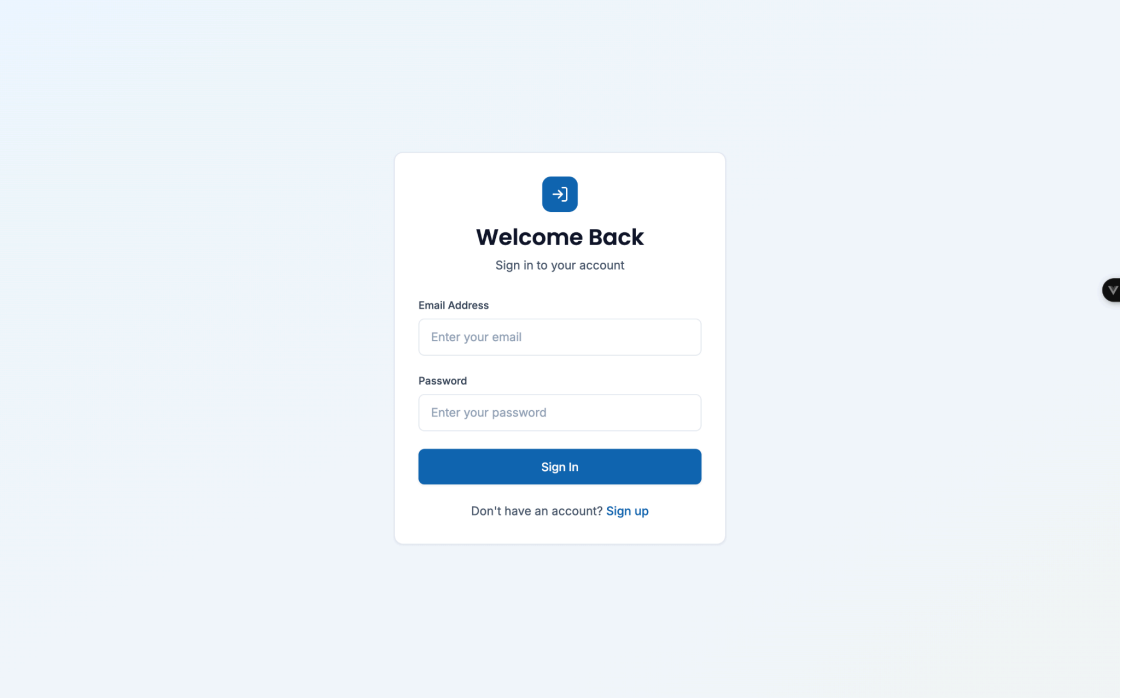
- Based on Node for build, then uses Nginx or lighter image for serving static files
- Copies source, installs dependencies, builds, and serves dist/ folder

docker-compose

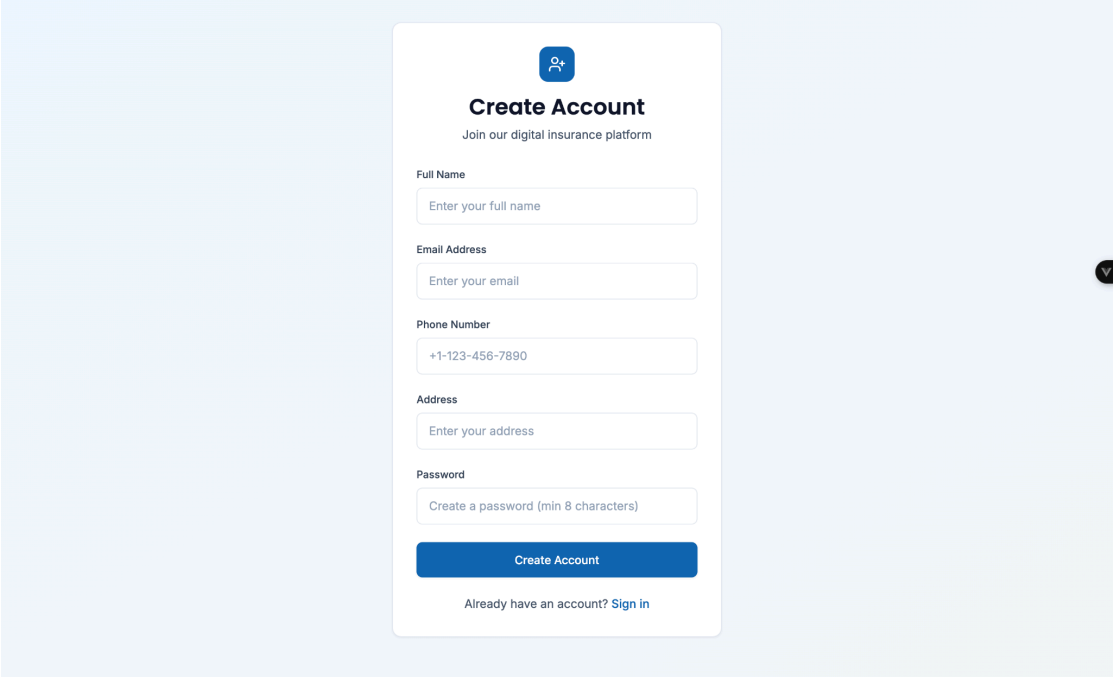
- Orchestrates multiple containers:
- backend: Runs Spring Boot
- frontend: Serves Vue.js build
- db: Runs the database (e.g., PostgreSQL)
- Defines network, volume mounts, and environment variables

Implementation Screenshots

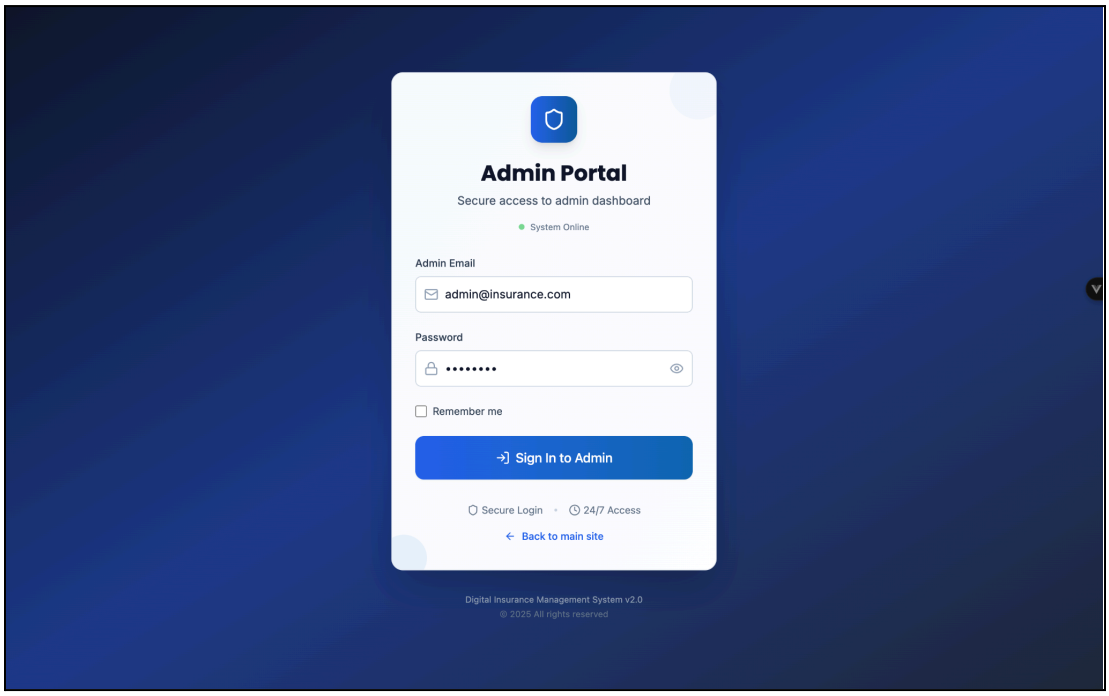
User Login



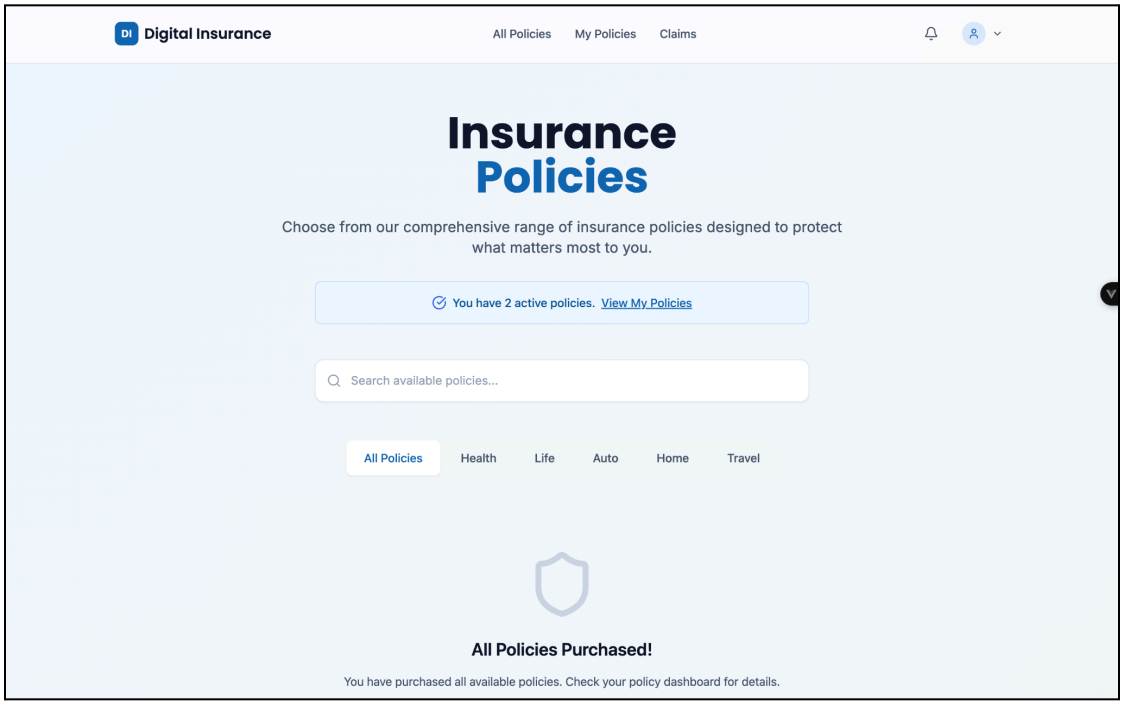
User Register



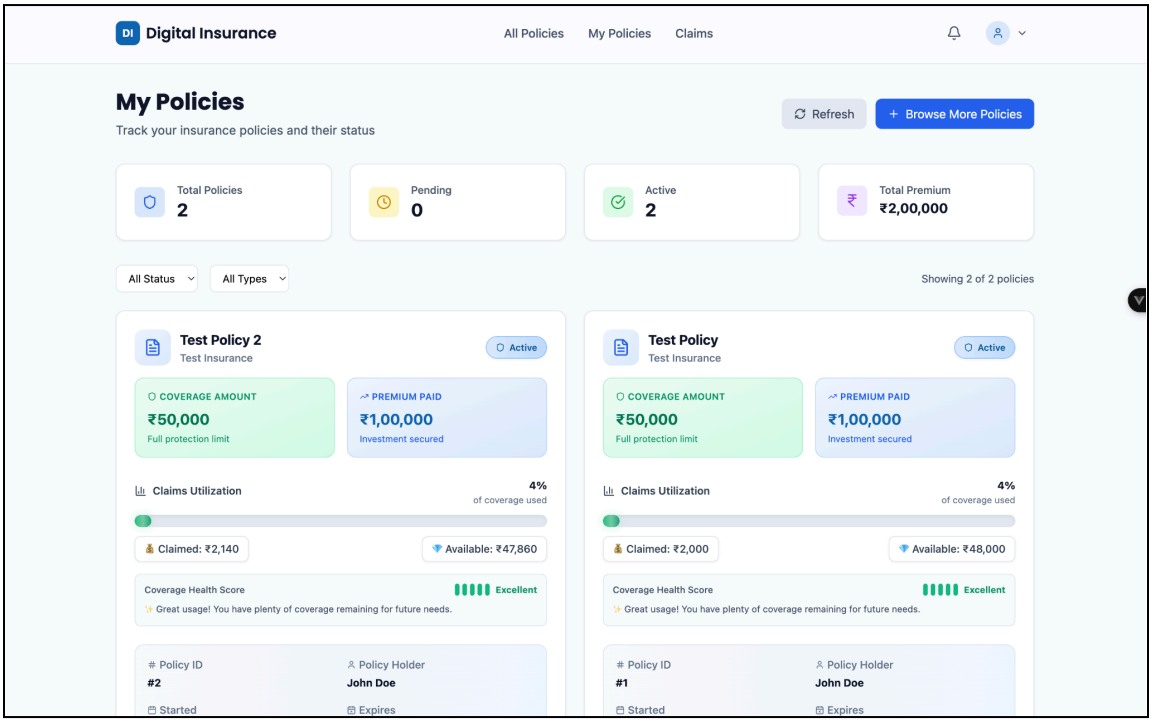
Admin Login



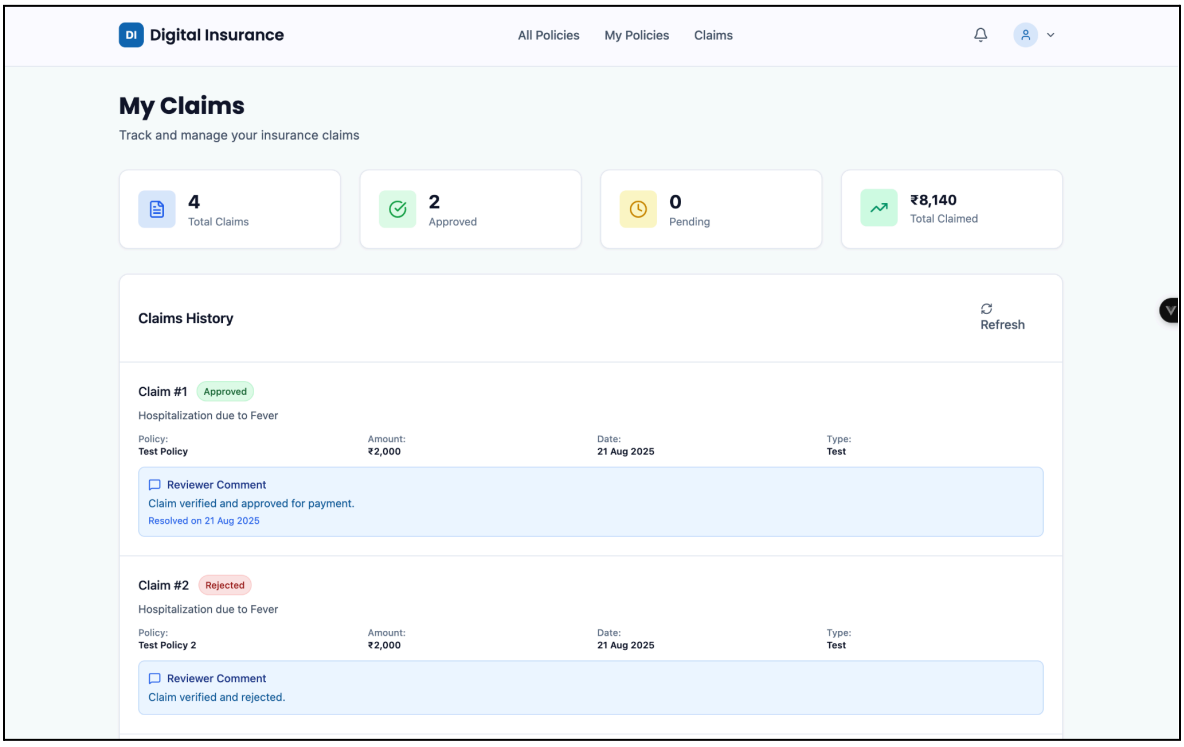
User Dashboard



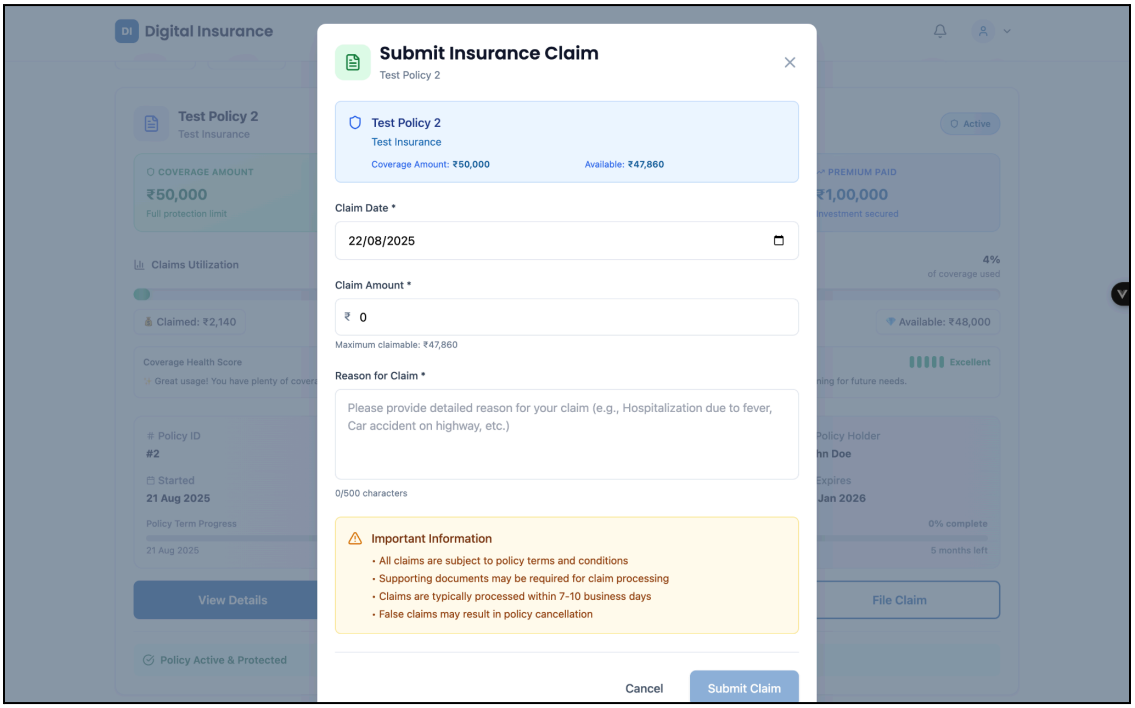
User Policies



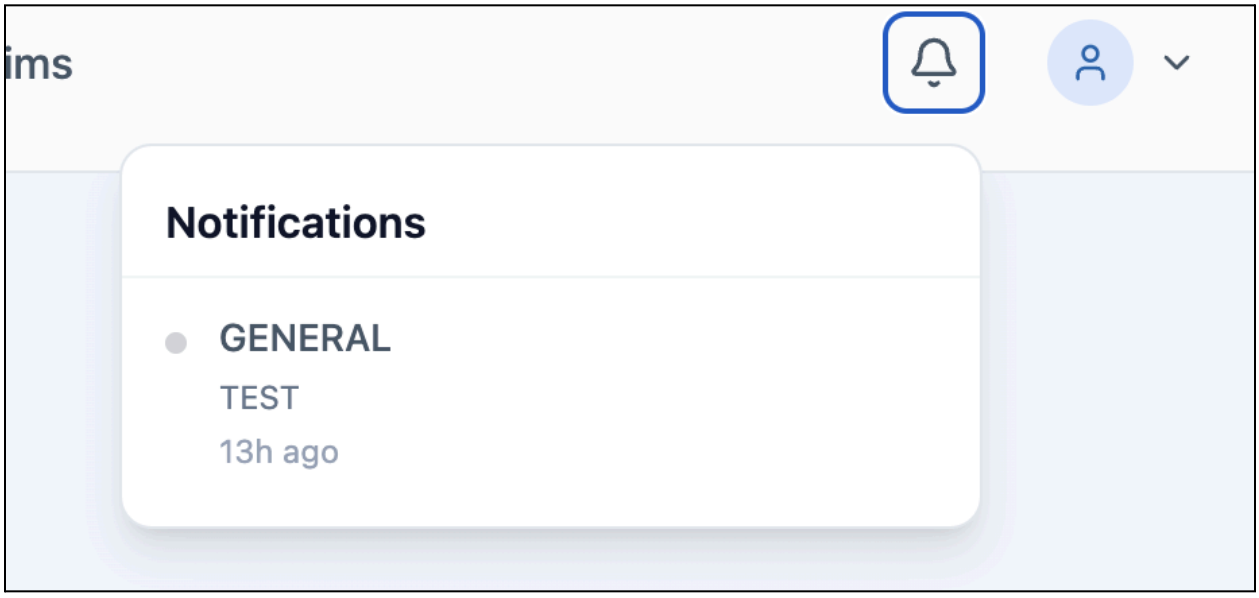
User Claims



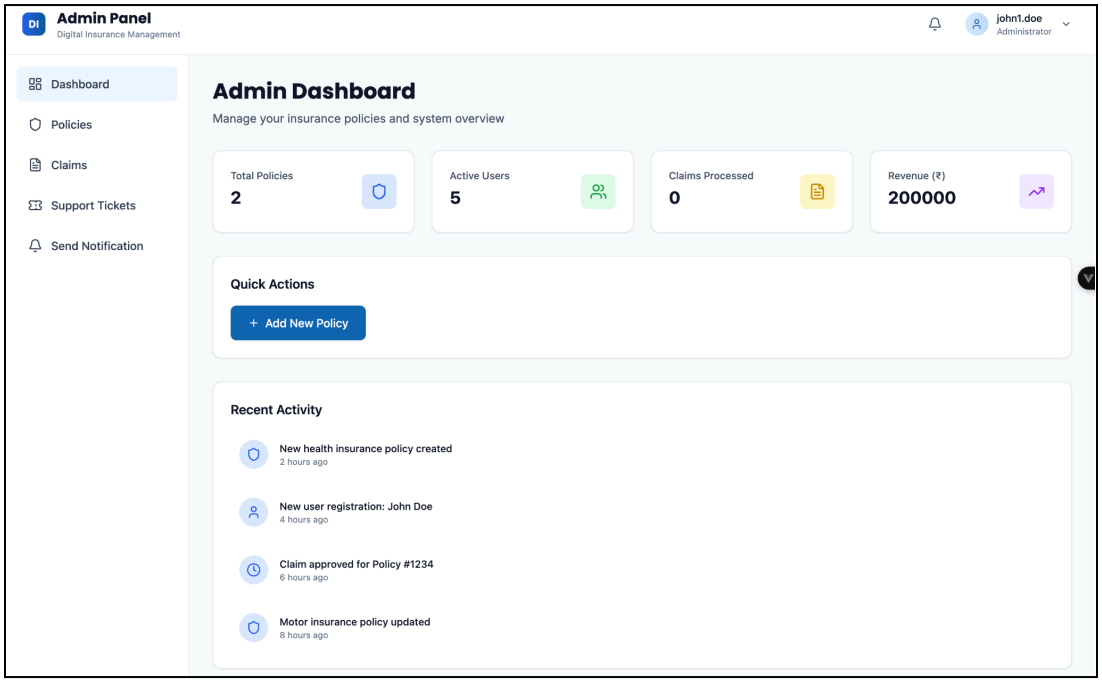
User Claim Submission



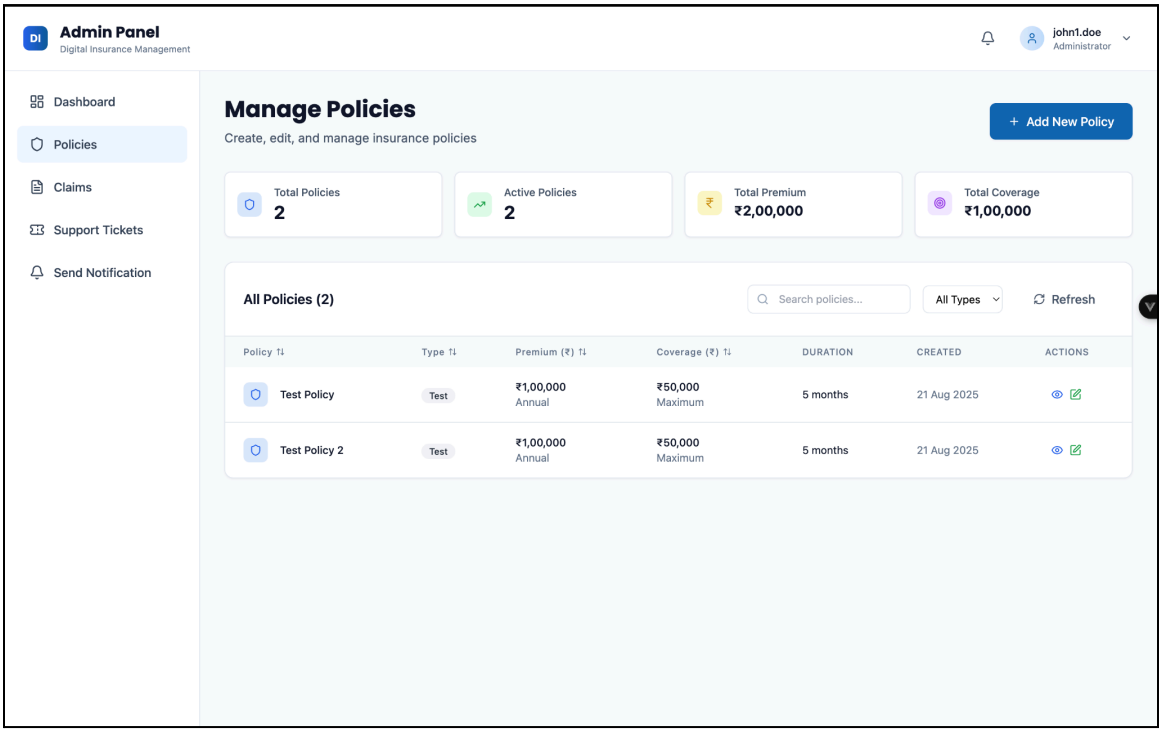
User Notifications



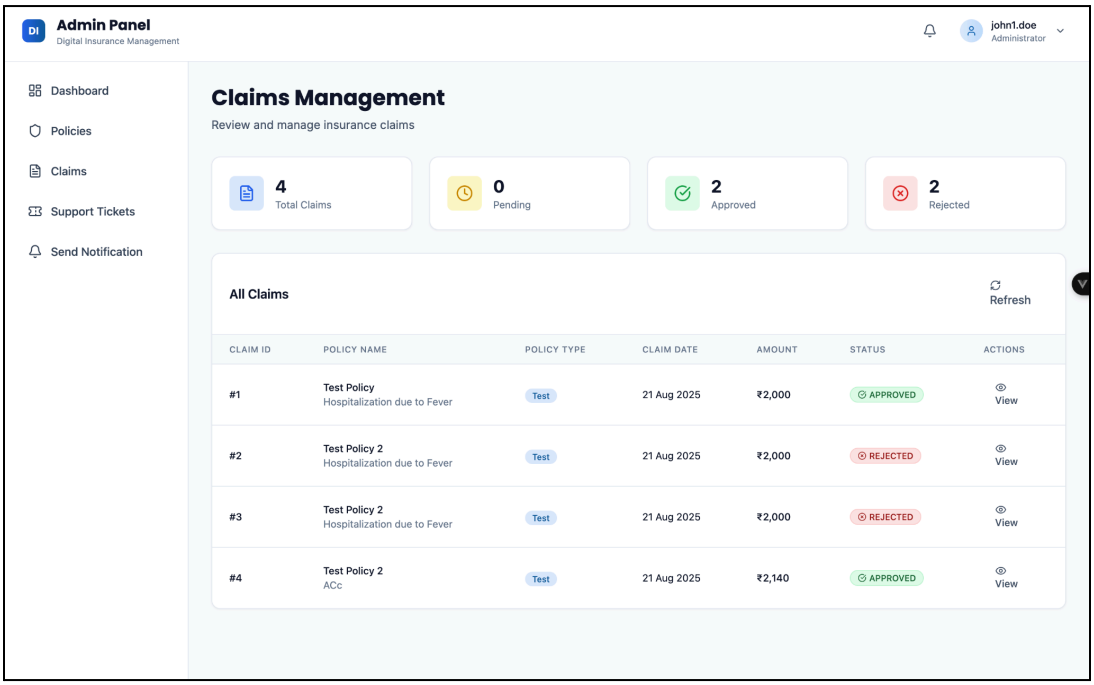
Admin Dashboard



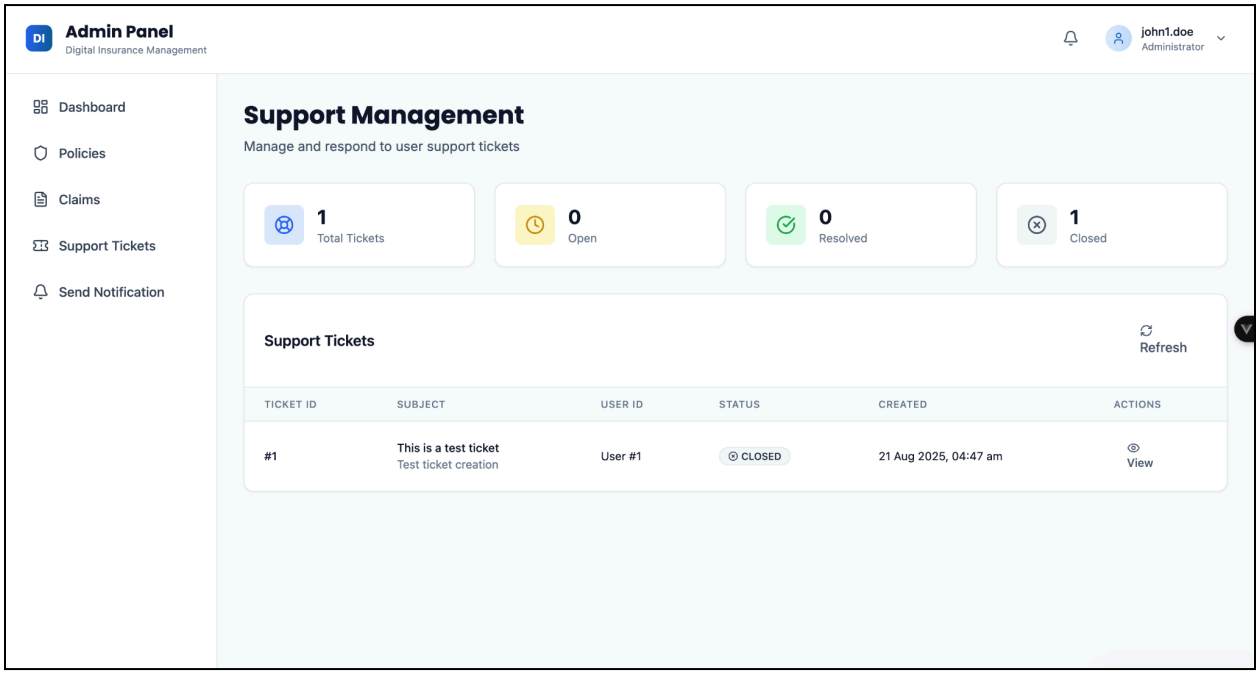
Admin Policy Management



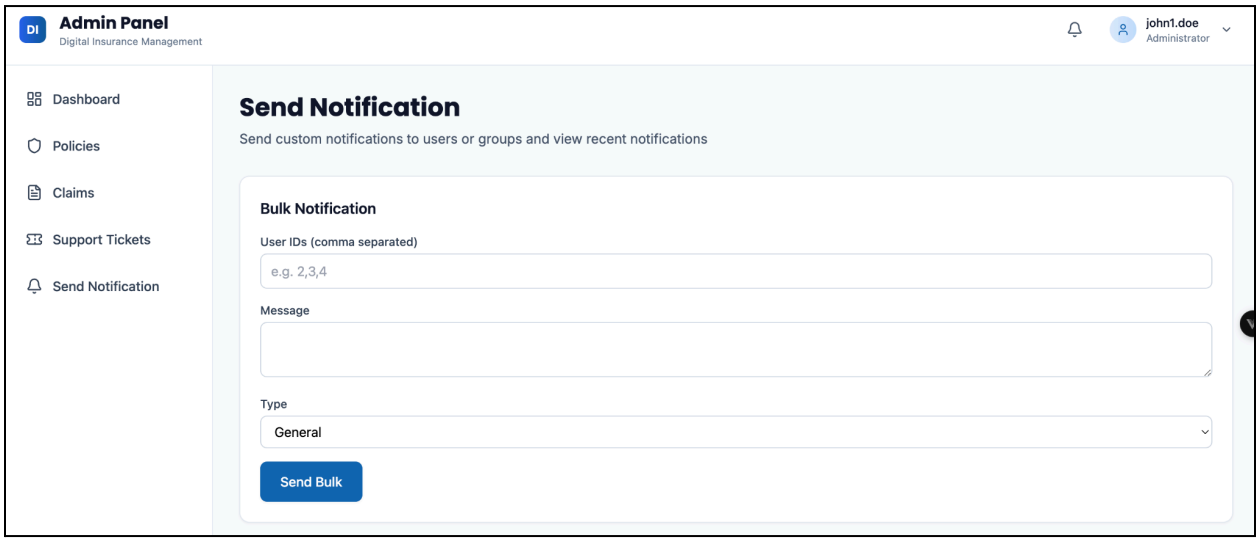
Admin Claims Management



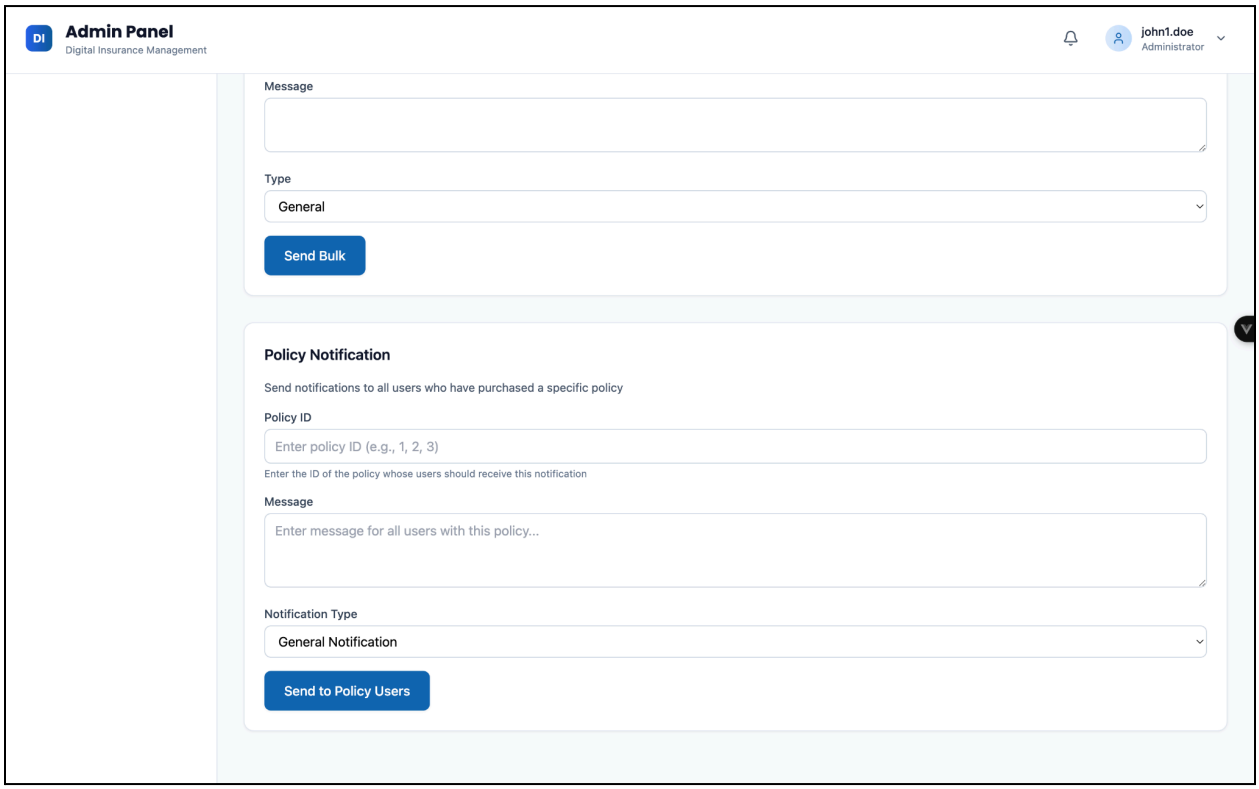
Admin Support Ticket Management



Admin Send Notification



Admin Send Notification By Policy



Testing

1. Testing Strategy

The project employs a layered testing approach to ensure both individual module reliability and overall system integrity. The main strategies used are:

1. **Unit Testing:**
    a. Each module, class, or function is tested in isolation to verify correct behavior.
    b. Backend: Unit tests cover service and utility classes.
    c. Frontend: Unit tests target Vue.js components, stores, and utility functions.
2. **Integration Testing:**
    a. Focuses on the interaction between multiple modules or components.
    b. Backend: Tests API endpoints, service-repository interaction, and authentication flows.
    c. Frontend: Tests end-to-end flows across multiple components (using stubs/mocks for backend).
3. **Manual Testing:**
    a. Exploratory and scenario-based testing to catch edge cases and usability issues.
    b. Performed before each release to verify user stories and workflow correctness.

2. Tools Used

1. **Backend:**
    a. **JUnit**: For unit and integration tests of Spring Boot services, controllers, and repositories.
    b. **Mockito**: For mocking dependencies in service layer tests.
    c. **Spring Boot Test**: For full-stack integration tests (API endpoints, database interaction).
2. **Frontend:**
    a. **Vitest**: For unit testing Vue.js components, composables, and utility functions.
    b. **@vue/test-utils**: For mounting and interacting with Vue components in tests.

Conclusion

The digital insurance management system has been designed and implemented with a strong focus on modularity, security, and maintainability. By leveraging modern frameworks such as Spring Boot for the backend and Vue.js for the frontend, the project ensures a clear separation of concerns, efficient data handling, and a responsive user experience.

Key best practices have been followed throughout development, including rigorous unit and integration testing, comprehensive documentation, and adherence to security standards such as JWT authentication, RBAC, and OWASP guidelines. The integration of Docker enables seamless deployment and scaling across diverse environments.

In summary, the system provides a robust, secure, and scalable platform for digital insurance management, supporting current business needs while being adaptable for future enhancements. Continued attention to testing, security, and code quality will help maintain the system's reliability and effectiveness as it evolves.