CAPSTONE PROJECT

InsureCore

A Digital Insurance Management System

Team Members

1. **Soumya Behura** (ZE3061)
2. **Deep Parekh** (ZE3058)
3. **S.K. Hussain** (ZE3065)
4. **Parth Verma** (ZE3075)

Organization

Better World Technology Pvt. Ltd.

GitHub Repository Link :
https://github.com/SoumyaBehura18/Digital-Insurance-Management-System

1. Abstract

The growing demand for digital-first insurance solutions has highlighted the need for a unified, transparent, and efficient platform that simplifies policy management for both customers and providers. InsureCore is a comprehensive digital insurance management system designed to address this challenge by providing seamless end-to-end functionality for users and administrators.

For customers, the platform enables quick registration, secure login, and role-based access to services. Users can browse available insurance products across Life, Health, and Vehicle domains, purchase policies, renew or cancel existing ones, and track their validity in real time. The system further integrates claims management, allowing users to raise claims, monitor their status, and receive timely updates. Additionally, a support ticket system ensures smooth resolution of user queries, improving customer trust and engagement.

For administrators, InsureCore offers robust dashboards to manage users, approve claims, monitor policies, and oversee support tickets. By digitizing workflows, the platform eliminates manual intervention, reduces processing delays, and ensures higher transparency.

The system is engineered with a Spring Boot (Java) backend providing RESTful APIs, integrated with PostgreSQL for reliable data persistence, and a Vue.js 3 frontend for a responsive and intuitive user experience. Security is ensured through Spring Security with JWT-based authentication, while Vuex manages client-side state. The use of Tailwind CSS and Lucide Vue Next ensures a modern and consistent UI.

With built-in testing support via JUnit, Mockito, and Vitest, InsureCore ensures quality and reliability. The application is deployable across local, staging, and production environments, with cloud-ready configurations for PostgreSQL (AWS RDS, GCP Cloud SQL, Azure).

By combining usability, scalability, and robustness, InsureCore transforms insurance management into a transparent, accessible, and efficient digital ecosystem, benefitting both customers and providers.

2. Project Goal

The goal of **InsureCore** is to design and develop a unified digital platform where users can **purchase, manage, and claim insurance policies** (Life, Health, and Vehicle) with ease. The system eliminates manual processes by providing a **digital-first solution** that ensures transparency, faster policy management, and seamless interaction between customers and insurance providers.

Key objectives include:

- **Simplifying user onboarding** → registration, login, and profile management with role-based access (User and Admin).
- **Enabling complete policy lifecycle management** → purchase, renewals, and cancellations.
- **Supporting claims management** → submission, approvals, status tracking, and resolution.
- **Providing a support ticket system** → to handle queries related to policies and claims.
- **Offering admin workflows** → for monitoring users, managing policies, approving claims, and resolving user tickets.
- **Delivering an intuitive and responsive UI/UX** → ensuring a smooth experience for both customers and admins.

Technologies Used

Frontend

- **Vue.js 3 (Composition API)** → Reactive, modular UI components
- **Vuex** → Centralized state management for users, policies, and claims
- **Vue Router** → Navigation, authentication flow, and route guards
- **Tailwind CSS** → Modern, utility-first styling & responsive layouts
- **Lucide Vue Next** → Lightweight, scalable icons
- **Axios** → Communication with backend APIs

Backend

- **Spring Boot (Java)** → RESTful APIs for users, policies, claims, and tickets
- **Spring Security + JWT** → Authentication and role-based access control
- **Hibernate (JPA)** → ORM & persistence layer

Database

- **PostgreSQL** → Primary relational database for users, policies, claims, tickets
- **Supabase** → Managed Postgres service (with optional authentication & storage support)

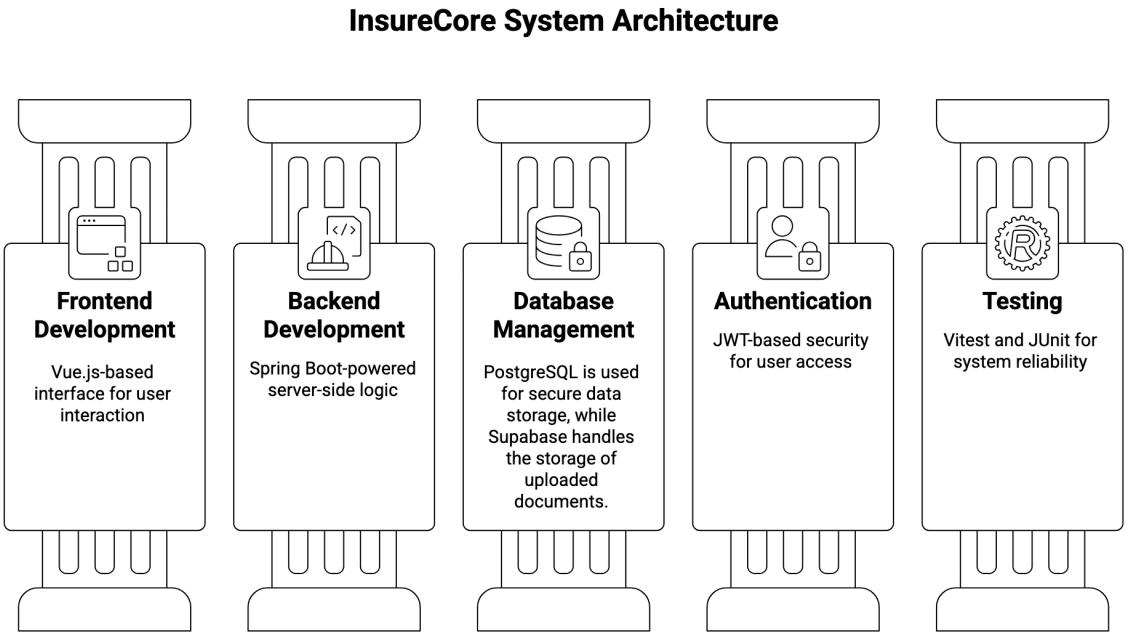
Testing

- **JUnit** → Backend unit testing
- **Mockito** → Mocking dependencies in backend tests
- **Vitest + Vue Test Utils** → Frontend unit & component testing

Build & Dependency Management

- **Maven** → Backend build & dependency management
- **npm** → Frontend dependency management

3. System Architecture



Backend (Spring Boot)

REST API Endpoints & Functionalities

This document provides a summary of all available API endpoints.

Authentication & User Management

- **POST /register** – Register a new user.
- **POST /login** – Authenticate user and return JWT token.
- **PUT /updateUserDetails/{userId}** – Update an existing user's details.

Policies

- **POST /policies/allPolicies** – Fetch all available policies (vehicle, life, health) based on user profile.
- **POST /policies/vehiclePolicies** – Fetch vehicle-related policies.
- **POST /policies/lifePolicies** – Fetch life-related policies.
- **POST /policies/healthPolicies** – Fetch health-related policies.

User Policies

- **POST /user/policy/purchase** – Purchase a policy for a user.
- **GET /user/policies/{userId}** – Get all policies purchased by a user.
- **GET /user/policy/{policyRecordId}** – Get details of a specific purchased policy.
- **PATCH /user/policy/ncb/{policyRecordId}** – Apply No Claim Bonus (NCB) to a user's policy.
- **PATCH /user/policy/status/{policyRecordId}** – Update the status of a user's policy (e.g., Renew).

Claims

- **POST /claim** – Create a new claim for a user's policy.
- **GET /claim/claims** – Get all claims (admin view).
- **GET /claim/user/{userId}** – Get all claims raised by a specific user.
- **PUT /claim/{claimId}/review** – Review a claim (approve/reject) with admin comments.
- **GET /claim/policy/{policyId}** – Get claims associated with a specific policy.
- **GET /claim/policy/remaining-amount/{policyRecordId}** – Get the remaining claimable amount for a user's policy

Tickets

- **GET /tickets/all – Retrieve all support tickets (admin view).**
- **GET /tickets/{ticketId} – Retrieve details of a specific ticket by ID.**
- **GET /tickets/user/{userId} – Retrieve all tickets submitted by a specific user.**
- **POST /tickets – Create a new support ticket (linked to a user, and optionally a policy/claim).**
- **POST /tickets/{ticketId}/messages – Add a message to a specific ticket.**
- **PATCH /tickets/{ticketId} – Update an existing support ticket (e.g., subject, description, status, policy/claim link).**

Data Models (Entities)

This document outlines the data models and their relationships within the Digital Insurance Management System.

1. User

Represents all system users, including customers, administrators, and agents.

Fields

- **id (Long)** – Unique identifier for the user.
- **name (String)** – The user's full name.
- **email (String)** – The user's unique email address.
- **password (String)** – The user's encrypted password.
- **roleType (Enum: CUSTOMER, ADMIN, AGENT)** – Defines the user's role and permissions.
- **age (Integer)** – The user's age.
- **phone (String)** – The user's contact phone number.
- **address (String)** – The user's physical address.
- **smokingDrinking (Boolean)** – Lifestyle factor used for risk evaluation in policy pricing.
- **preexistingConditions (List<String>)** – A list of the user's relevant medical history.
- **vehicleType (String)** – The type of vehicle owned by the user, if applicable.
- **vehicleAge (Integer)** – The age of the user's vehicle.
- **createdAt, updatedAt (Timestamps)** – Timestamps for record creation and last update.

Relationships

- **One-to-Many** with UserPolicy.
- **One-to-Many** with SupportTicket.
- **One-to-Many** with Claim.

2. Policy

Represents the master definitions for all insurance policies offered.

Fields

- **id (Long)** – Unique identifier for the policy.

- **policyName (String)** – The name of the policy.
- **policyType (Enum: HEALTH, LIFE, VEHICLE)** – The category of the policy.
- **coverageAmt (Double)** – The total coverage amount offered by the policy.
- **durationMonths (Integer)** – The duration of the policy term in months.
- **premiumRate (Double)** – The base premium rate.
- **renewalRate (Double)** – The rate for renewing the policy.
- **termsAndConditions (String)** – The detailed terms and conditions of the policy.
- **createdAt (Timestamp)** – Timestamp for when the policy was created.

Relationships

- **One-to-Many** with UserPolicy.
- **One-to-One** with HealthPolicyPremium, LifePolicyPremium, or VehiclePolicyPremium.
- **One-to-Many** with Claim.

3. UserPolicy

A junction entity that links a specific user to a purchased policy.

Fields

- **id (Long)** – Unique identifier for the user-policy record.
- **userId (FK → User)** – Foreign key referencing the User.
- **policyId (FK → Policy)** – Foreign key referencing the Policy.
- **startDate (Date)** – The date the policy coverage begins.
- **endDate (Date)** – The date the policy coverage ends.
- **status (Enum: ACTIVE, RENEWED, EXPIRED, CANCELLED)** – The current status of the policy.
- **premiumPaid (Double)** – The amount of premium the user has paid.
- **noClaimBonus (Boolean)** – Indicates if a no-claim bonus is applicable.
- **coverageAmount (Double)** – The specific coverage amount for this user's policy.

Relationships

- **Many-to-One** with User.
- **Many-to-One** with Policy.
- **One-to-Many** with Claim.

4. HealthPolicyPremium

Contains health-specific details that affect the premium calculation.

Fields

- **id (Long)** – Unique identifier.
- **policyId (FK → Policy)** – Foreign key referencing the Policy.
- **baseAmount (Double)** – The base premium amount for the health policy.
- **finalAmount (Double)** – The final calculated premium amount.
- **ageFactor (Double)** – A factor applied to the premium based on age.
- **medicalHistoryFactor (Double)** – A factor applied based on medical history.
- **coverageDetails (String)** – Specific details about what is covered.
- **deductibles (Double)** – The deductible amount for the policy.

Relationships

- **One-to-One** with Policy.
- **One-to-Many** with HealthPreexistingCondition.

5. LifePolicyPremium

Contains life insurance-specific premium details.

Fields

- **id (Long)** – Unique identifier.
- **policyId (FK → Policy)** – Foreign key referencing the Policy.
- **sumAssured (Double)** – The total amount assured to the beneficiary.
- **premiumAmount (Double)** – The premium amount for the policy.
- **beneficiaryDetails (String)** – Information about the policy's beneficiary.
- **termPeriod (Integer)** – The term period of the life insurance policy.
- **riskFactors (String)** – Specific risk factors considered for the premium.

6. VehiclePolicyPremium

Contains vehicle insurance-specific premium details.

Fields

- **id (Long)** – Unique identifier.
- **policyId (FK → Policy)** – Foreign key referencing the Policy.
- **vehicleDetails (make, model, year)** – Details of the insured vehicle.
- **coverageType (String)** – The type of vehicle coverage (e.g., comprehensive).
- **premiumAmount (Double)** – The calculated premium for the vehicle policy.

- **driverHistoryFactor (Double)** – A factor based on the driver's history.

7. HealthPreexistingCondition

Tracks specific health conditions that affect a health policy's premium.

Fields

- **id (Long)** – Unique identifier.
- **healthPremiumId (FK → HealthPolicyPremium)** – Foreign key to the health premium record.
- **condition (String)** – The name of the pre-existing condition.
- **additionalPremium (Double)** – The additional premium charged for this condition.
- **policyId (FK → Policy)** – Foreign key referencing the Policy.

8. SupportTicket

Manages customer support issues and inquiries.

Fields

- **id (Long)** – Unique identifier for the ticket.
- **userId (FK → User)** – The user who created the ticket.
- **subject (String)** – The subject of the support ticket.
- **description (String)** – A detailed description of the issue.
- **priority (Enum: LOW, MEDIUM, HIGH)** – The priority level of the ticket.
- **status (Enum: OPEN, IN_PROGRESS, RESOLVED, CLOSED)** – The current status of the ticket.
- **createdAt, resolvedAt (Timestamps)** – Timestamps for ticket creation and resolution.
- **assignedAgentId (FK → User)** – The agent assigned to handle the ticket.

Relationships

- **One-to-Many** with Message.

9. Message

Represents a single message within a support ticket's communication thread.

Fields

- **id (Long)** – Unique identifier for the message.
- **ticketId (FK → SupportTicket)** – The ticket this message belongs to.

- **senderId, receiverId (FK → User)** – The sender and receiver of the message.
- **content (String)** – The text content of the message.
- **timestamp (DateTime)** – When the message was sent.
- **messageType (Enum: TEXT, SYSTEM)** – The type of message.
- **isRead (Boolean)** – Indicates if the message has been read by the receiver.

10. Claim

Manages the lifecycle of an insurance claim from submission to resolution.

Fields

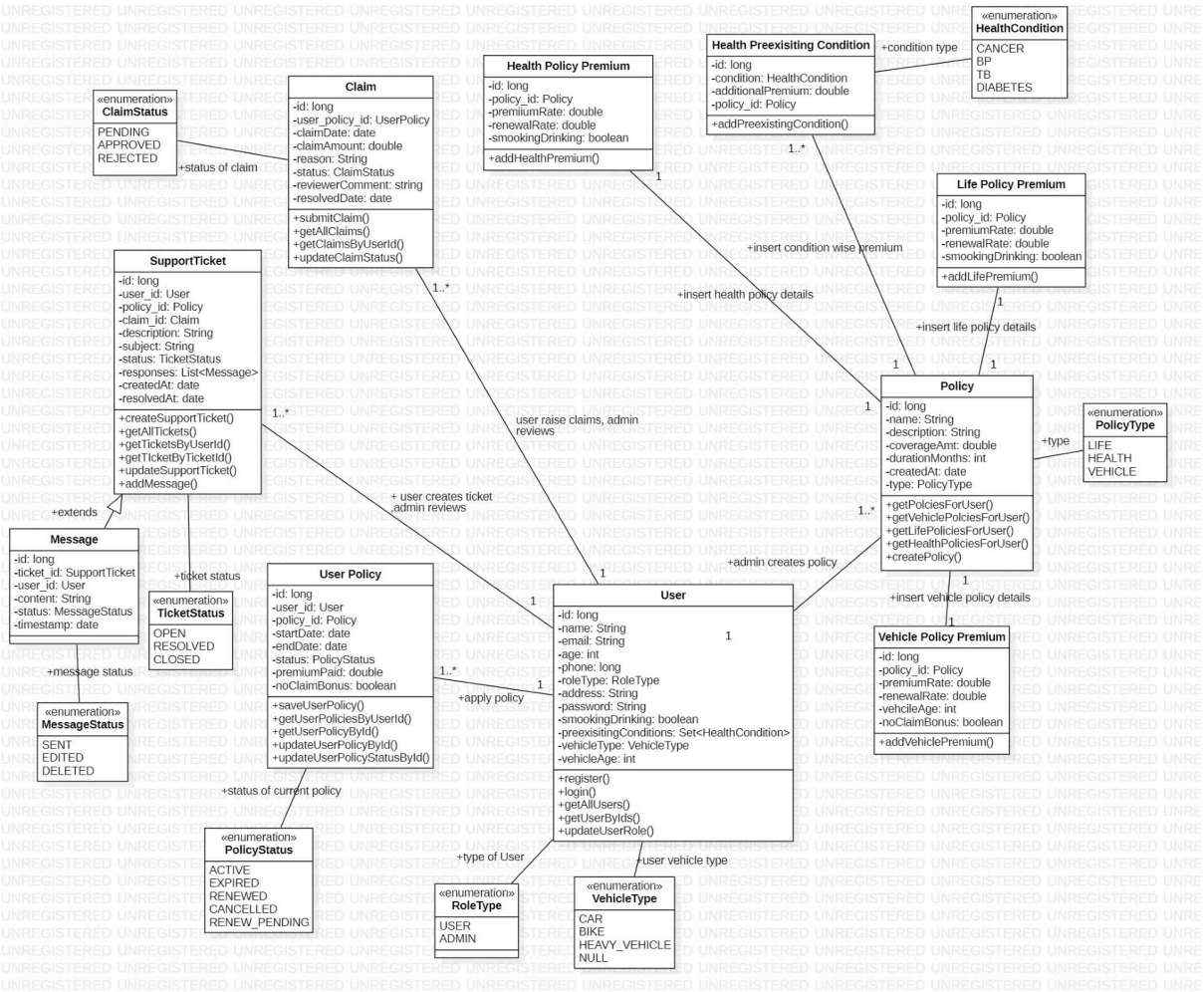
- **id (Long)** – Unique identifier for the claim.
- **userPolicyId (FK → UserPolicy)** – The specific user policy being claimed against.
- **userId (FK → User)** – The user who filed the claim.
- **policyId (FK → Policy)** – The master policy associated with the claim.
- **claimAmount (Double)** – The amount requested by the user.
- **approvedAmount (Double)** – The amount approved by the admin.
- **reason (String)** – The reason for the claim.
- **status (Enum: SUBMITTED, PENDING, UNDER_REVIEW, APPROVED, REJECTED)** – The current status of the claim.
- **reviewerComment (String)** – Comments from the admin who reviewed the claim.
- **claimDate, resolvedDate (Date)** – Dates for when the claim was filed and resolved.

11. UserPrincipal

A Spring Security entity used for handling user authentication and authorization.

- Implements the UserDetails interface.
- Stores username, password, roles, and authorities for the authenticated user.
- Used in JWT authentication for user login and subsequent API request authorization.

Class Diagram



Services & Business Logic Implementation

This document outlines the core services and their responsibilities within the Digital Insurance Management System.

UserService

- Manages user registration, login, and authentication.
- Handles profile updates and password management.
- Enforces role-based access control (USER, ADMIN, AGENT).

PolicyService

- Creates and manages master policies (Health, Life, Vehicle).
- Handles policy validation, renewal, and status management.

- Provides search and filtering capabilities for policies.

UserPolicyService

- Links users to their purchased policies.
- Manages policy purchases, cancellations, and history.
- Retrieves a user's active policies.

HealthPremiumService

- Calculates health insurance premiums based on user profile.
- Adjusts premiums for pre-existing conditions.
- Applies discounts and recalculates premiums on changes.

LifePremiumService

- Calculates life insurance premiums based on sum assured and term.
- Adjusts premiums based on age and risk factors.
- Manages beneficiary details.

VehiclePremiumService

- Calculates vehicle premiums based on vehicle and driver details.
- Considers driver history and coverage type.
- Adjusts for geographic and other risk factors.

HealthPreexistingConditionService

- Manages pre-existing medical conditions for users.
- Calculates additional premium costs for specific conditions.
- Validates conditions and applies exclusions where necessary.

ClaimService

- Manages the claim lifecycle from submission to settlement.
- Handles document verification and fraud detection.
- Provides claim status tracking.

SupportTicketService

- Manages customer support tickets from creation to resolution.
- Handles ticket assignment, escalation, and SLAs.
- Tracks agent workload.

MessageService

- Manages all communication within the system, primarily for support tickets.
- Supports messaging, notifications, and attachments.
- Tracks message status (read/unread).

AuthenticationService

- Validates user login credentials.
- Generates and verifies JWT tokens for session management.
- Manages password encryption.

AuthorizationService

- Enforces role-based access control and method-level security.
- Validates user permissions for specific API endpoints.
- Manages admin privileges.

FileUploadService

- Handles secure file and document uploads to Supabase storage.
- Validates file size and type.
- Manages file URLs and access.

NotificationService

- Sends system notifications via email, SMS, and in-app alerts.
- Manages notification templates and user preferences.
- Tracks delivery status.

ReportService

- Generates analytics and reports on policies, claims, and users.
- Provides financial summaries and usage statistics.
- Supports exporting reports to PDF and Excel.

ValidationService

- Provides centralized validation for input and business rules.
- Checks user eligibility for policies.
- Ensures data integrity and regulatory compliance.

AuditService

- Tracks all significant user and system activities.
- Records changes to sensitive data like policies and claims.
- Provides audit logs for monitoring and compliance.

Repositories (Data Access Layer)

UserRepository

- Provides CRUD operations for User entity.
- Supports searching by username, email, and role.
- Validates uniqueness of usernames and emails.

PolicyRepository

- Manages persistence of Policy entities.
- Fetches policies by policy number, type, or status.
- Supports date-based queries.

UserPolicyRepository

- Manages user-policy associations.
- Fetches policies purchased by a user.
- Supports filtering by status.

HealthPolicyPremiumRepository

- Stores and retrieves health premium details.
- Fetches premiums by policy ID, age group, or coverage type.

LifePolicyPremiumRepository

- Provides persistence for life policy premium details.
- Retrieves premiums by sum assured, term, or beneficiary.

VehiclePolicyPremiumRepository

- Stores vehicle-specific premium details.
- Fetches premiums by vehicle type, make, model, or registration number.

HealthPreexistingConditionRepository

- Manages pre-existing medical conditions linked to health premiums.

- Fetches conditions by premium ID, policy ID, or condition name.

SupportTicketRepository

- Manages persistence of support tickets.
- Fetches tickets by user, status, priority, or assigned agent.
- Identifies open and overdue tickets.

MessageRepository

- Handles messaging between users and agents.
- Retrieves messages by sender/receiver or support ticket.
- Tracks unread messages.

ClaimRepository

- Manages insurance claims.
- Fetches claims by user, policy, status, or date range.
- Retrieves pending and recent claims for reporting.

UserPrincipalRepository

- Handles persistence for authentication-related user data.
- Fetches security principals by username or user ID.
- Tracks active users and login activity.

Repository Layer Features

- Built on Spring Data JPA.
- Provides CRUD, pagination, and sorting.
- Supports custom JPQL and native queries.
- Integrates with Specifications API for dynamic filtering.
- Includes transaction management and batch processing.
- Optimized using indexing, lazy loading, and caching.

Frontend (Vue.js)

This document outlines the structure, components, routing, and state management for the Vue.js frontend of the Digital Insurance Management System.

Components

ClaimsManagement Module

- **ClaimList.vue:** Displays user claims with status, documents, and admin reviews.
- **SubmitClaim.vue:** A form to submit new claims with document upload and validation.
- **AdminClaims.vue:** Admin dashboard to approve/reject claims, filter, and view documents.

Policies Module

- **PolicyView.vue:** Shows the policy catalog and allows for comparisons.
- **MyPolicies.vue:** Displays a user's purchased/active policies and their claim history.
- **AdminPolicies.vue:** An admin interface to create, edit, and manage master policies.
- **PurchaseModal.vue:** A modal for the policy purchase flow, including premium & NCB discounts.

Tickets Module

- **TicketList.vue:** Displays a user's support ticket history.
- **SubmitTicket.vue:** A form for submitting new support tickets.
- **AdminTickets.vue:** An admin interface for managing and responding to tickets.

User Management

- **RegisterPage.vue:** Handles user registration and profile editing.
- **LoginPage.vue:** Manages user authentication with JWT.

Dashboard & Views

- **DashboardView.vue:** Provides admin/user dashboards with statistics and quick actions.

Routing

- **Public Routes:** /, /login, /register
- **User Routes:** /dashboard, /policies, /claims, /submit-claim, /tickets
- **Admin Routes:** /admin/dashboard, /admin/policies, /admin/claims, /admin/tickets, /admin/users
- **Fallback Route:** /:pathMatch(.*)* redirects to a NotFound page.
- **Route Guards:** Role-based navigation is enforced using the JWT stored in localStorage.

State Management (Vuex)

Modules

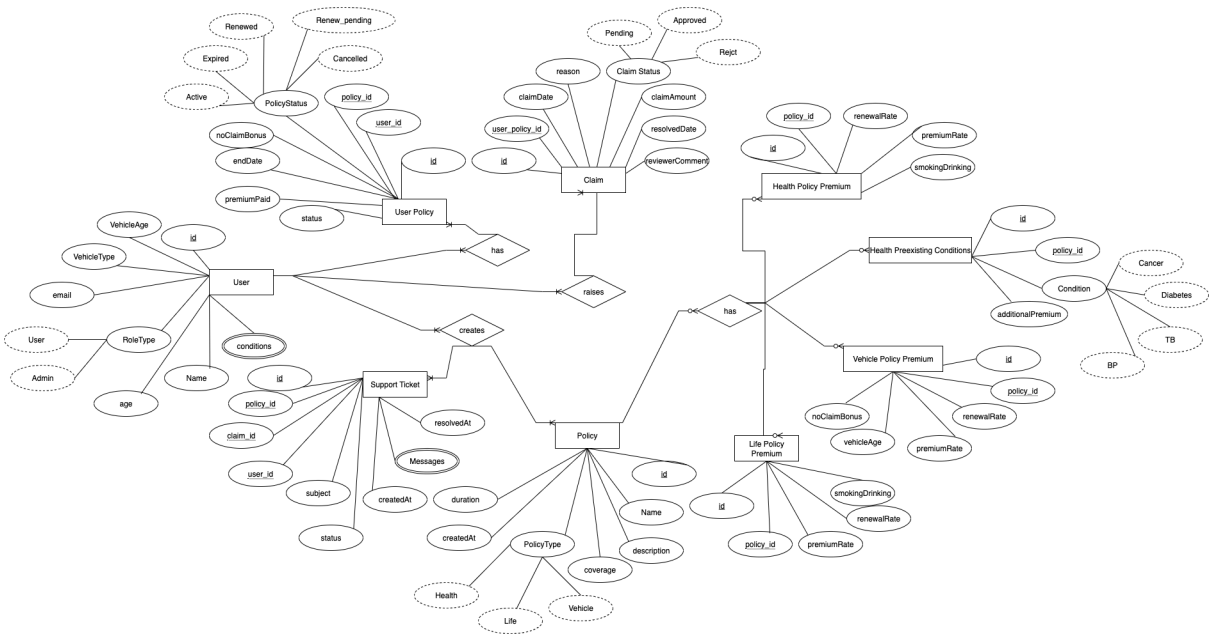
- **claims/:** Manages the claim list, submission process, and related documents.
- **policies/:** Manages the policy catalog, purchases, and user-specific policies.
- **user/:** Handles authentication, user profile data, and session state.
- **tickets/:** Manages ticket CRUD operations and admin ticket management.
- **Store Access:** State is centralized and actions are dispatched via $store.dispatch().

API Integration

- **HTTP Client:** Axios is used for all HTTP requests.
- **Utility Functions:** makeRequestWithToken and makeRequestWithoutToken are used to streamline API calls.
- **Authentication:** The JWT is stored in localStorage and automatically included in request headers.
- **File Uploads:** Supabase Storage is used for handling document uploads (5MB limit, supporting JPG/PNG/PDF/DOC).
- **Error Handling:** Errors are handled centrally in Vuex to provide user-friendly messages.

Database Schema Design

This document outlines the core entities and relationships in the database for the Digital Insurance Management System.



Core Entities

User

- Stores user details (id, name, email, age, role, phone, address, lifestyle info, vehicle details, conditions).
- **Relationships:**
    - Can purchase many policies (UserPolicy).
    - Can raise claims.
    - Can create support tickets.

Policy

- Master table for all policies (Health, Life, Vehicle).
- Fields: id, name, description, coverage, duration, type.
- **Relationships:**
    - Linked with UserPolicy (many users can purchase a policy).
    - Each has one premium record (health, life, or vehicle).

UserPolicy

- Junction table linking User and Policy.
- Fields: id, user_id, policy_id, status, premiumPaid, endDate, noClaimBonus.
- **Relationships:**
    - One user policy can have many claims.

Premium Entities

HealthPolicyPremium

- Stores health premium rates (renewalRate, premiumRate, lifestyle factors).
- Linked to HealthPreexistingConditions.

LifePolicyPremium

- Stores life insurance premiums (sum assured, premium amount, term, etc.).

VehiclePolicyPremium

- Stores vehicle premium details (renewalRate, premiumRate, vehicleAge, NCB).

HealthPreexistingCondition

- Tracks extra costs for health conditions (Cancer, Diabetes, BP, TB).

Claims

- Stores insurance claims linked to a user's policy.
- Fields: id, user_policy_id, claimDate, claimAmount, status, reviewerComment, resolvedDate.
- Status values: Pending, Approved, Rejected.

Support & Communication

SupportTicket

- Stores user support tickets (subject, status, createdAt, resolvedAt).
- Related to a user and optionally a policy/claim.

Message

- Messages under support tickets.
- Fields: id, sender, receiver, content, createdAt.

Key Relationships

- **User ↔ UserPolicy ↔ Policy** (many-to-many via UserPolicy).
- **Policy ↔ Premium tables** (one-to-one).
- **HealthPolicyPremium ↔ HealthPreexistingConditions** (one-to-many).
- **UserPolicy ↔ Claim** (one-to-many).
- **User ↔ SupportTicket ↔ Message** (one-to-many chain).

Test Plan

Unit Testing

Unit testing is the process of testing individual components or functions in isolation to ensure they work as expected. In this project, unit tests validate backend services and controllers, as well as frontend Vue components and stores.

Frameworks & Tools

- **JUnit 5** – Backend unit testing framework
- **Mockito** – Mocking framework for backend dependencies
- **Vitest** – Modern testing framework for Vue.js frontend (components, Vuex, utilities)

Test Cases

Service Layer Tests

- **UserService:**
    - Register user successfully
    - Prevent registration with duplicate email
    - Encode and store passwords securely
- **PolicyService:**
    - Create and fetch policies by type
    - Apply premium calculation logic
    - Handle invalid policy creation requests
- **ClaimService:**
    - Submit claim for active policy
    - Reject claim for inactive/expired policy
    - Approve/reject claim with reviewer comments

Controller Layer Tests

- Verify API endpoints return correct status codes (200, 201, 400, 404)
- Validate request/response JSON mapping
- Test secured endpoints with valid/invalid JWT tokens
- Check role-based access (USER vs ADMIN routes)
- Ensure error handling returns proper messages

Test Execution Strategy

- **Unit Tests (Backend):**
    - Run using JUnit and Mockito on each commit (CI/CD pipeline).
- **Frontend Tests (Vue + Vitest):**
    - Run Vitest for Vue components, Vuex stores, and utilities.
    - Component tests include rendering, props validation, event handling, and user interactions.
    - Store tests verify actions, mutations, and API integration.
    - Vitest runs in watch mode for fast feedback during development.

4. Setup and Configuration

Backend Setup

Prerequisites

- Java 17+ (ensure JAVA_HOME is set)
- Maven 3.8+
- (Optional) IntelliJ IDEA or VS Code

Steps to Run Backend

- cd backend
- ./mvnw spring-boot:run

Backend will run at **http://localhost:9090** (configurable in application.properties).

Frontend Setup

Prerequisites

- Node.js 18+
- npm

Steps to Run Frontend

- cd frontend
- npm install
- npm run serve

Frontend will be available at **http://localhost:8080**, communicating with backend via Axios.

Database Setup

Prerequisites

- PostgreSQL 13+ (local or Supabase)
- Database client (pgAdmin, DBeaver, CLI)

Steps

1. Start PostgreSQL server.

- Create a database:

  CREATE DATABASE insurecore;

- Update src/main/resources/application.properties:

  # PostgreSQL Configuration

- spring.datasource.url=jdbc:postgresql://localhost:5432/insurecore
- spring.datasource.username=your_username
- spring.datasource.password=your_password
- spring.datasource.driver-class-name=org.postgresql.Driver

  # Supabase Configuration

- supabase.url=https://exhhnhdfkmwxluwhvyvq.supabase.co
- supabase.anon.key=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6ImV4aGhuaGRma213eGx1d2h2eXZxIiwicm9sZSI6ImFub24iLCJpYXQiOjE3NTkxNDA5NTEsImV4cCI6MjA3NDcxNjk1MX0.Vf4aUCUDj1F-grXYztUrUFn4t-sgAupuFBwRybLDhBw
- supabase.bucket.name=insurance-management-system
-

- # JPA & Hibernate
- spring.jpa.hibernate.ddl-auto=update
- spring.jpa.show-sql=true
- spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

Auto-Created Tables (via Hibernate)

- users
- user_preexisting_conditions
- policies
- user_policies
- claims
- support_tickets
- ticket_messages
- health_policy_premium
- health_preexisting_conditions
- life_policy_premium
- vehicle_policy_premiums

5. API Documentation

1. Register User

This endpoint allows for the creation of a new user account in the system.

- **Endpoint:** POST /register
- **Description:** Registers a new user with their personal, health, and vehicle details. The user's password is encrypted upon storage.

Request Body

The request body must be a JSON object containing the following fields:

- **name (string, required):** The full name of the user.
- **email (string, required):** The user's unique email address. Used for login.
- **age (number, required):** The age of the user.
- **phone (string, required):** The user's phone number.
- **roleType (string, required):** The role assigned to the user (e.g., "ADMIN", "USER").
- **address (string, required):** The user's complete mailing address.
- **password (string, required):** The user's desired password. Must be strong.
- **smokingDrinking (boolean, required):** Indicates if the user smokes or drinks.
- **preexistingConditions (array of strings, required):** A list of any medical conditions the user has.
- **vehicleType (string, required):** The type of vehicle the user owns (e.g., "CAR", "BIKE").
- **vehicleAge (number, required):** The age of the user's vehicle in years.

Example Success Response (201 Created)

Upon successful registration, the API returns the created user object with a unique id and an encrypted password.


```
{
  "id": 18,
  "name": "SAM PARKER",
  "email": "sam@gmail.com",
  "age": 21,
  "phone": "9876543210",
  "roleType": "ADMIN",
  "address": "123 Main Street, New Delhi, India",
  "password":"$2a$12$KdN9qmkZr0NqIrJ6zVKLhuSo5yq3c6hXZa85yK0hD
  Pr7OAEAK.uUm",
  "smokingDrinking": false,
  "preexistingConditions": [
    "CANCER",
    "DIABETES"
  ],
```

```
  "vehicleType": "CAR",
  "vehicleAge": 4
}
```

2. Login User

This endpoint authenticates a user and provides them with a JSON Web Token (JWT) for accessing protected routes.

- **Endpoint:** POST /login
- **Description:** Authenticates a user with their email and password. On success, it returns a JWT and key user details.

Request Body

The request body must be a JSON object containing the following fields:

- email (string, required): The registered email address of the user.
- password (string, required): The user's password.

Example Request

```
{
  "email": "sam@gmail.com",
  "password": "12345678"
}
```

Example Success Response (200 OK)

Upon successful authentication, the API returns a response containing a JWT and essential user information. The response body will be a JSON object with the following fields:

```
{
  "userId": 18,
  "role": "admin",
                                                          "token":
"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzYW1AZ21haWwuY29tIiwiaWF0Ijox
NzU5MDgyMTg0LCJleHAiOjE3NjE2NzQxODR9.vR9N4iKWTqnZl8rlF8Dpd5wyh
kx3MR58KsX7K9BcfJo",
  "smokingDrinking": false,
  "vehicleType": "CAR",
  "vehicleAge": 4,
  "preexistingConditions": [
    "CANCER",
    "DIABETES"
  ],
  "name": "SAM PARKER",
  "email": "sam@gmail.com"
}
```

3. Get All Policies

- **Endpoint:** POST /policies/allPolicies
- **Description:** Fetches all policies (vehicle, life, health) based on user details.

Request Body

- smokingDrinking (boolean): The user's smoking/drinking status.
- vehicleType (string): The user's vehicle type.
- vehicleAge (number): The age of the user's vehicle.
- preexistingConditions (array of strings): The user's pre-existing medical conditions.

Example Response


```
[
  {
    "policyId": 28,
    "policyName": "VEHICLE-1",
    "policyType": "VEHICLE",
    "premiumRate": 100.0,
    "renewalRate": 1200.0,
    "duration": 1200,
    "coverage": 100.0
  },
  {
    "policyId": 26,
    "policyName": "LIFE-1",
    "policyType": "LIFE",
    "premiumRate": 100.0,
    "renewalRate": 100.0,
    "duration": 100,
    "coverage": 100.0
  }
]
```


4. Get Vehicle Policies

- **Endpoint:** POST /policies/vehiclePolicies
- **Description:** Fetches only vehicle-related policies.

Example Response


```
[
  {
    "policyId": 28,
    "policyName": "VEHICLE-1",
    "policyType": "VEHICLE",
    "premiumRate": 100.0,
    "renewalRate": 1200.0,
```

```
    "duration": 1200,
    "coverage": 100.0
  }
]
```

5. Get Life Policies

- **Endpoint:** POST /policies/lifePolicies
- **Description:** Fetches only life insurance policies.

Example Response

```
[
  {
    "policyId": 26,
    "policyName": "LIFE-1",
    "policyType": "LIFE",
    "premiumRate": 100.0,
    "renewalRate": 100.0,
    "duration": 100,
    "coverage": 100.0
  }
]
```

6. Get Health Policies

- **Endpoint:** POST /policies/healthPolicies
- **Description:** Fetches only health insurance policies.

Example Response

```
[
  {
    "policyId": 27,
    "policyName": "health-1",
    "policyType": "HEALTH",
    "premiumRate": 100.0,
    "renewalRate": 100.0,
    "duration": 100,
    "coverage": 199.0
  }
]
```

7. Purchase Policy

- **Endpoint:** POST /user/policy/purchase
- **Description:** Purchases a policy for a user and creates a policy record.

Request Body

- userId (number): The ID of the user purchasing the policy.
- policyId (number): The ID of the policy being purchased.
- startDate (string): The start date of the policy in yyyy-MM-dd format.
- endDate (string): The end date of the policy in yyyy-MM-dd format.
- status (string): The initial status of the policy (e.g., "ACTIVE").
- premiumPaid (number): The amount of premium paid by the user.

Example Response


```
{
  "id": 13,
  "userId": 1,
  "policyId": 28,
  "startDate": "2025-09-25",
  "endDate": "2026-09-25",
  "status": "ACTIVE",
  "premiumPaid": 15000.0,
  "policyName": "VEHICLE-1",
  "policyType": "VEHICLE",
  "noClaimBonus": false,
  "coverageAmount": 100.0
}
```


8. Get User Policies

- **Endpoint:** GET /user/policies/{userId}
- **Description:** Fetches all policies purchased by a specific user.

Example Response


```
[
  {
    "id": 13,
    "userId": 1,
    "policyId": 28,
    "startDate": "2025-09-25",
    "endDate": "2026-09-25",
    "status": "ACTIVE",
    "premiumPaid": 15000.0,
    "policyName": "VEHICLE-1",
    "policyType": "VEHICLE",
```

```
    "noClaimBonus": false,
    "coverageAmount": 100.0
  }
]
```

9. Apply No Claim Bonus (NCB)

- **Endpoint:** PATCH /user/policy/ncb/{policyRecordId}
- **Description:** Applies a No Claim Bonus to a specific user's policy record.

Example Response

```
{
  "id": 13,
  "userId": 1,
  "policyId": 28,
  "status": "ACTIVE",
  "noClaimBonus": true
}
```

10. Update Policy Status

- **Endpoint:** PATCH /user/policy/status/{policyRecordId}
- **Description:** Updates the status of a specific policy record (e.g., to "RENEWED", "EXPIRED").

Request Body

- policyStatus (string): The new status for the policy.

Example Response

```
{
  "id": 13,
  "userId": 1,
  "policyId": 28,
  "status": "RENEWED",
  "noClaimBonus": true
}
```

11. Get User Policy by Id

- **Endpoint:** GET /user/policy/{policyRecordId}
- **Description:** Retrieves a single policy record by its unique ID.

Example Response


```
{
  "id": 13,
  "userId": 1,
  "policyId": 28,
  "status": "RENEWED",
  "noClaimBonus": true,
  "coverageAmount": 100.0
}
```


12. Create a Claim

- **Endpoint:** POST /claim
- **Description:** Creates a new claim request for a user’s active policy.

Request Body

- userPolicyId (number): The ID of the user's policy record.
- claimDate (string): The date of the claim in yyyy-MM-dd format.
- claimAmount (number): The amount being claimed.
- reason (string): The reason for the claim.

Example Response


```
{
  "id": 8,
  "userPolicyId": 1,
  "claimDate": "2024-01-15",
  "claimAmount": 5000,
  "reason": "Medical expenses for emergency treatment",
  "status": "PENDING",
  "reviewerComment": "",
  "resolvedDate": null
}
```

13. Get All Claims

- **Endpoint:** GET /claim/claims
- **Description:** Fetches all claims in the system (admin view).

Example Response


```
[
  {
    "id": 8,
    "userPolicyId": 1,
    "userId": 1,
    "userName": "John Doe",
    "userEmail": "john.doe@example.com",
    "policyName": "Health Plus Plan",
    "claimDate": "2024-01-15",
    "claimAmount": 5000,
    "reason": "Medical expenses for emergency treatment",
    "status": "PENDING",
    "reviewerComment": "",
    "resolvedDate": null
  }
]
```


14. Get Claims by User

- **Endpoint:** GET /claim/user/{userId}
- **Description:** Fetches all claims raised by a specific user.

Example Response


```
[
  {
    "id": 8,
    "userPolicyId": 1,
    "userId": 1,
    "userName": "John Doe",
    "userEmail": "john.doe@example.com",
    "policyName": "Health Plus Plan",
    "claimDate": "2024-01-15",
    "claimAmount": 5000,
    "reason": "Medical expenses for emergency treatment",
    "status": "PENDING",
    "reviewerComment": "",
    "resolvedDate": null
  }
]
```

15. Review a Claim

- **Endpoint:** PUT /claim/{claimId}/review
- **Description:** Used by an admin to review a claim, update its status, and add comments.

Request Body

- reviewComments (string): Comments from the reviewer.
- status (string): The new status, either "APPROVED" or "REJECTED".

Example Response

A successful request will return an HTTP 200 OK status code with no response body.

16. Get Policy Claims

- **Endpoint:** GET /claim/policy/{policyId}
- **Description:** Fetches claims associated with a given policy.

Example Response


```
[
  {
    "id": 1,
    "user": {
      "id": 1,
      "name": "John Doe",
      "email": "john.doe@example.com"
    },
    "policy": {
      "id": 1,
      "name": "Health Plus Plan",
      "description": "Comprehensive health insurance policy",
      "coverageAmt": 500000,
      "durationMonths": 12,
      "type": "HEALTH"
    },
    "startDate": "2024-01-01",
    "endDate": "2025-01-01",
    "status": "ACTIVE"
  }
]
```

17. Get User Policies

- **Endpoint:** GET /user/policies/{userId}
- **Description:** Fetches all policies purchased by a user.

Example Response


```
[
  {
    "id": 13,
    "userId": 1,
    "policyId": 28,
    "startDate": "2025-09-25",
    "endDate": "2026-09-25",
    "status": "RENEWED",
    "premiumPaid": null,
    "policyName": "VEHICLE-1",
    "policyType": "VEHICLE",
    "noClaimBonus": true,
    "coverageAmount": 100.0
  }
]
```


18. Apply No Claim Bonus (NCB)

- **Endpoint:** PATCH /user/policy/ncb/{policyRecordId}
- **Description:** Marks a policy as eligible for No Claim Bonus (NCB).

Example Response


```
{
  "id": 13,
  "userId": 1,
  "policyId": 28,
  "status": "RENEWED",
  "noClaimBonus": true,
  "coverageAmount": 100.0
}
```


19. Get Remaining Claim Amount

- **Endpoint:** GET /claim/policy/remaining-amount/{policyRecordId}
- **Description:** Returns the remaining claimable amount for a given policy.

Example Response


```
{
```

```
  "policyId": 13,
  "remainingClaimAmount": 100.0
}
```

20. Update User Details

- **Endpoint:** PUT /updateUserDetails/{userId}
- **Description:** Updates an existing user's details. This endpoint requires authentication via a Bearer Token. It updates all editable fields such as name, email, age, phone, address, lifestyle habits, pre-existing conditions, and vehicle details. The user's password remains encrypted in the response.

Request Body

The request body must be a JSON object containing the fields to be updated:

- name (string): The user's full name.
- email (string): The user's email address.
- age (number): The user's age.
- phone (string): The user's phone number.
- roleType (string): The user's role (e.g., "USER", "ADMIN").
- address (string): The user's complete address.
- smokingDrinking (boolean): Indicates if the user smokes or drinks.
- preexistingConditions (array of strings): A list of any medical conditions the user has.
- vehicleType (string): The type of vehicle the user owns.
- vehicleAge (number): The age of the user's vehicle in years.

Example Request

```
{
  "name": "user101",
  "email": "user1011@example.com",
  "age": 30,
  "phone": "9999999999",
  "roleType": "USER",
  "address": "Delhi, India",
  "smokingDrinking": false,
  "preexistingConditions": ["TB"],
  "vehicleType": "CAR",
  "vehicleAge": 5
}
```

Example Success Response (200 OK)

Upon a successful update, the API returns the complete updated user object.

```
{
  "id": 3,
  "name": "user101",
  "email": "user1011@example.com",
  "age": 30,
  "phone": "9999999999",
  "roleType": "USER",
  "address": "Delhi, India",
  "password":"$2a$12$Yf.q3M2gwGgsO2O3R3f/beNZ/o93AGwFpgbqyehVw
  dehkt0cfezgC",
  "smokingDrinking": false,
  "preexistingConditions": [
    "TB"
  ],
  "vehicleType": "CAR",
  "vehicleAge": 5
}
```


21. Get All Tickets

- **Endpoint:** GET /tickets/all
- **Description:** Retrieves all support tickets in the system (admin view).

Example Response


```
[
  {
    "id": 1,
    "userId": 1,
    "policyId": 2,
    "claimId": 2,
    "subject": "need more claim.",
      "description": "i need more claim money for my health
insurance",
    "status": "OPEN",
    "createdAt": "2025-09-30T04:04:16.269+00:00",
    "resolvedAt": null,
    "messages": []
  }
]
```


22. Get Ticket by ID

- **Endpoint:** GET /tickets/{ticketId}
- **Description:** Retrieves details of a specific ticket using its ID.

Example Response

```
{
  "id": 1,
  "userId": 1,
  "policyId": 2,
  "claimId": 2,
  "subject": "need more claim.",
    "description": "i need more claim money for my health
insurance",
  "status": "OPEN",
  "createdAt": "2025-09-30T04:04:16.269+00:00",
  "resolvedAt": null,
  "messages": []
}
```

23. Get Tickets by User

- **Endpoint:** GET /tickets/user/{userId}
- **Description:** Retrieves all tickets created by a specific user.

Example Response

```
[
  {
    "id": 1,
    "userId": 1,
    "policyId": 2,
    "claimId": 2,
    "subject": "need more claim.",
    "description": "i need more claim money for my health
insurance",
    "status": "OPEN",
    "createdAt": "2025-09-30T04:04:16.269+00:00",
    "resolvedAt": null,
    "messages": []
  }
]
```

24. Create Ticket

- **Endpoint:** POST /tickets
- **Description:** Creates a new support ticket for a user. Can optionally be linked to a policy or claim.
- **Request Body**
    - userId (number): ID of the user creating the ticket.
    - policyId (number, optional): Related policy ID.

- claimId (number, optional): Related claim ID.
- subject (string): Ticket subject/title.
- description (string): Detailed description of the issue.

Example Response


```
{
  "id": 2,
  "userId": 1,
  "policyId": null,
  "claimId": null,
  "subject": "Demo Ticket",
    "description": "this is a demo description for a demo
ticket",
  "status": "OPEN",
  "createdAt": "2025-09-30T04:06:28.725+00:00",
  "resolvedAt": null,
  "messages": []
}
```


25. Add Message to Ticket

- **Endpoint:** POST /tickets/{ticketId}/messages
- **Description:** Adds a new message to an existing ticket.
- **Request Body**
  - authorId (number): ID of the message sender.
  - content (string): The message content.
  - author (string): Role of the sender (e.g., USER, ADMIN).

Example Response


```
{
  "id": 1,
  "authorId": 1,
  "content": "this is a message response for ticket",
  "timestamp": "2025-09-30T04:07:03.104+00:00"
}
```


26. Update Ticket

- **Endpoint:** PATCH /tickets/{ticketId}
- **Description:** Updates details of an existing support ticket. Can be used to update subject, description, status, or link to a policy/claim.
- **Request Body**
  - policyId (number, optional): Updated policy ID.
  - claimId (number, optional): Updated claim ID.

- subject (string): Updated subject.
- description (string): Updated description.
- status (string): Ticket status (e.g., OPEN, IN_PROGRESS, RESOLVED, CLOSED).

Example Response


```
{
  "id": 1,
  "userId": 1,
  "policyId": 1,
  "claimId": 2,
  "subject": "Demo Ticket",
    "description": "this is a demo description for a demo
ticket",
  "status": "OPEN",
  "createdAt": "2025-09-30T04:04:16.269+00:00",
  "resolvedAt": null,
  "messages": [
    {
      "id": 1,
      "authorId": 1,
      "content": "this is a message response for ticket",
      "timestamp": "2025-09-30T04:07:03.104+00:00"
    }
  ]
}
```

6. Deployment

Deployment Environments

Local Development → Backend on localhost:9090, frontend on localhost:8080, database on localhost:5432.

Staging → For QA/testing before production.

Production → Live system accessible by end-users.

Deployment Steps

1. Backend Deployment

Build JAR:

cd backend

./mvnw clean package -DskipTests

Output: target/digital-insurance-management-system-0.0.1-SNAPSHOT.jar

Run JAR locally:

java -jar target/digital-insurance-management-system-0.0.1-SNAPSHOT.jar

2. Frontend Deployment

Build Vue app:

- cd frontend
- npm install
- npm run build

Output: Static files in /dist.

Serve with Nginx:

```
server {
    listen 80;
    server_name insurance.example.com;

    root /var/www/insurance-frontend;
    index index.html;

    location / {
        try_files $uri /index.html;
    }

    location /api/ {
        proxy_pass http://localhost:9090/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
sudo systemctl reload nginx
```

3. Database Deployment

Local

- PostgreSQL running on localhost:5432
- Default database: insurecore
- Credentials in application.properties

Production

- Use managed PostgreSQL (AWS RDS, GCP Cloud SQL, Azure PostgreSQL).
- Apply migrations using Hibernate auto-DDL or Flyway/Liquibase.

4. Environment Configuration

Backend env variables:

DB_URL=jdbc:postgresql://localhost:5432/insurecore

DB_USER=your_username

DB_PASS=your_password

JWT_SECRET=your-secret-key

SERVER_PORT=9090

Frontend .env file:

VUE_APP_API_BASE_URL=http://localhost:9090

After Deployment

Frontend → http://insurance.example.com

Backend APIs → http://insurance.example.com/api

Database → Managed PostgreSQL instance (AWS RDS / GCP Cloud SQL / Azure PostgreSQL) or local PostgreSQL (localhost:5432)

7. Future Enhancements

The Digital Insurance Management System is feature-rich for core workflows (user management, policies, claims, and tickets). However, several improvements can be made to enhance usability, efficiency, and security in future releases.

1. Enhanced User Experience

- **Mobile App Development:** Build a companion mobile app for Android/iOS for quick access to policies, claims, and support.
- **Multi-language Support:** Provide localization and translation for wider accessibility.

2. Policy Management Improvements

- **AI-Driven Recommendations:** Suggest suitable policies based on user profile, health history, and risk factors.
- **Policy Renewal Automation:** Automated reminders and auto-renewal options with payment integration.
- **Bundled Products:** Ability to offer combined packages (e.g., Health + Life).

3. Claims Processing Enhancements

- **Document OCR Integration:** Automatically read uploaded claim documents for faster processing.
- **AI-based Fraud Detection:** Flag suspicious claims using ML models.
- **Digital Signature Support:** Enable secure approvals via e-signatures.

4. Advanced Support & Communication

- **Chatbot Integration:** AI chatbot for instant support and FAQs.
- **Multi-channel Support:** Support via email, SMS, WhatsApp, and in-app chat.
- **Agent Assignment System:** Automatically route tickets to available agents based on workload.
- **Feedback & Rating System:** Collect user ratings after issue resolution.

5. Security & Compliance

- **Two-Factor Authentication (2FA):** Additional security for logins.
- **Role Expansion:** More granular roles like AGENT, MANAGER, PARTNER.

8.Team Roles & Responsibilities

Soumya Behura

– User Authentication & Admin Policy Management

Backend APIs

- POST /auth/register – User Registration (with JWT authentication)
- POST /auth/login – User Login (JWT-based authentication & token generation)
- POST /policies – Create new insurance policies (Admin only)
- PUT /policies/{policyId} – Update existing policies (Admin only)

Business Rules

- JWT-based authentication for all protected routes
- Role-based access control (USER vs ADMIN)
- Only admins can create, update, or delete policies

Frontend

- **Register.vue, Login.vue** – User registration & login forms with JWT token storage
- **AdminPolicies.vue** – Admin dashboard to create, edit, and delete policies
- **PolicyCatalog.vue** – (for all users) browsing available policies

Unit Testing

- Authentication & JWT validation tests
- Policy creation and update flow tests

Deep Parekh

– User Policy Management & Dashboards

Backend APIs

- GET /policies – Retrieve list of policies (filterable by type/status)
- POST /user/policy/purchase – Purchase a policy
- GET /user/policies – View a user's purchased policies
- PUT /user/policy/{id} – Manage/renew/cancel policies

Business Rules

- Only registered users can purchase policies
- Users can own multiple policies with statuses: ACTIVE, EXPIRED, CANCELLED, RENEW_PENDING
- Admins can view and manage all user policies

Frontend

- **PolicyCatalog.vue** – Displays all available policies with filters
- **MyPolicies.vue** – User's purchased policies and actions (renew, cancel)
- **UserDashboard.vue** – Personalized user dashboard (overview of policies & claims)
- **AdminUsers.vue** – Admin dashboard to manage registered users

Unit Testing

- Policy purchase & renewal logic
- Expiry simulation tests
- Admin user management tests

SK Hussain – Claims Management & Admin Dashboard

Backend APIs

- POST /claim – Submit a claim for a policy
- GET /user/claims – Retrieve user's submitted claims
- PUT /claim/{claimId}/status – Admin action: Approve/Reject claim

Business Rules

- Claims are allowed only for ACTIVE policies
- Claims require admin approval for validation
- Claim Status = PENDING, APPROVED, REJECTED

Frontend

- **SubmitClaim.vue** – Form to raise a claim
- **ClaimList.vue** – User view of submitted claims
- **AdminClaims.vue** – Admin dashboard for claim review and actions

Unit Testing

- Claim eligibility and approval logic
- API tests for claim status transitions

Parth Verma – Support & Ticketing System

Backend APIs

- POST /support – Submit a support query (linked to policy/claim)
- GET /support/user/{userId} – Fetch all tickets raised by a user

- PUT /support/{ticketId} – Admin updates ticket response & status

Business Rules

- Users can raise tickets linked to specific policies or claims
- Admins can resolve and close tickets

Frontend

- **SupportForm.vue** – Submit queries/issues
- **TicketList.vue** – User view of submitted support tickets
- **AdminSupport.vue** – Admin dashboard for ticket management

Unit Testing

- Ticket creation & resolution flow
- Input validation tests

9. Appendix

PolicyBazaar – Online Insurance Comparison & Purchase Platform
https://www.policybazaar.com

Coverfox Insurance – Digital insurance brokerage and policy management
https://www.coverfox.com

ACKO General Insurance – Digital-first insurer with claim & policy management
https://www.acko.com

IRDAI (Insurance Regulatory and Development Authority of India) – Regulatory guidelines for insurance services
https://irdai.gov.in

Spring Boot Documentation – Backend framework used for APIs
https://spring.io/projects/spring-boot

Vue.js Documentation – Frontend framework used for building reactive UIs
https://vuejs.org/

Supabase Documentation – PostgreSQL as a managed service with authentication/storage support
https://supabase.com/docs

10. Conclusion

The InsureCore application successfully delivers a digital-first insurance management system that bridges the gap between customers and providers. By integrating user registration, authentication, policy management, claims processing, and support ticket resolution into one platform, it eliminates manual inefficiencies and ensures transparency at every stage.

With a Vue.js frontend, Spring Boot backend, and PostgreSQL database, the system provides a scalable, secure, and user-friendly solution for managing life, health, and vehicle insurance policies. Features such as JWT-based authentication, role-based access (User vs Admin), responsive dashboards, and policy lifecycle automation further enhance usability and reliability.

The project also emphasizes maintainability and quality through unit testing with Vitest, JUnit, and Mockito, ensuring robustness across both frontend and backend modules.

In conclusion, InsureCore achieves its goal of modernizing insurance operations by offering a seamless, transparent, and efficient experience for both customers and administrators, setting a strong foundation for future enhancements like analytics, premium prediction, and cloud-native scaling.