**High-Level Design Document for Insurance-backend**

**1. Introduction**

The **Insurance-backend** project is a backend system designed to support insurance-related services, offering features like user management, insurance policy tracking, claims processing, and automated notifications. Built with Spring Boot, the system leverages an H2 database for data storage, JWT for secure authentication, and Jakarta Mail for email notifications.

**2. System Architecture**

The Insurance-backend architecture follows a **multi-layered microservices-inspired design**. Each layer is separated by its responsibilities, ensuring modularity, ease of maintenance, and scalability.

**2.1 Architecture Layers**

* **Presentation Layer (Controller)**: Manages HTTP requests, routing them to the appropriate services.
* **Business Logic Layer (Service)**: Contains business rules, logic, and orchestrates data manipulation.
* **Persistence Layer (Repository)**: Directly interfaces with the H2 database to store and retrieve data.
* **Utility Layer**: Provides utilities for functions like token generation, encryption, and email configuration.

**2.2 Technology Stack**

* **Backend Framework**: Spring Boot
* **Database**: H2 (in-memory, development), JDBC
* **Authentication**: JWT (JSON Web Token)
* **Email**: Jakarta Mail
* **IDE**: IntelliJ or Eclipse for development

**3. Component Diagram**

The application’s major components and their interactions are as follows:

1. **Authentication Component**: Manages user authentication using JWT.
2. **User Management Component**: Manages user profiles and access.
3. **Policy Management Component**: Handles operations on insurance policies.
4. **Claim Management Component**: Manages claims associated with policies.
5. **Email Notification Component**: Sends emails for various actions like registration and claim updates.
6. **Utility Component**: Supports JWT processing, password encryption, and file uploads.

**4. Module-Level Design and Interactions**

**4.1 Authentication and Authorization Module**

**Purpose**: Secures user access by managing JWT-based authentication and role-based authorization.

**Components**

* **AuthController**: Exposes endpoints for login and registration.
* **JwtTokenUtil**: Creates, parses, and validates JWT tokens.
* **JwtRequestFilter**: Intercepts requests to verify the presence and validity of the JWT.
* **UserDetailsServiceImpl**: Implements Spring Security’s UserDetailsService to load user details from the database.

**Workflow**

1. **Login**: Users send credentials to AuthController.
   * Credentials are verified, and a JWT is generated for the user.
2. **JWT Validation**: Each incoming request passes through JwtRequestFilter.
   * The filter extracts and validates the JWT, and loads user details if the token is valid.

**Security**

* **JWT Token Structure**: Header.Payload.Signature, with the payload containing user roles and expiration data.
* **Role-Based Access Control (RBAC)**: Routes and services are restricted based on roles (USER, ADMIN).

**4.2 User Management Module**

**Purpose**: Manages user registration, profile details, and access roles.

**Components**

* **UserController**: Exposes APIs for user registration, profile retrieval, and updates.
* **UserService**: Contains business logic for managing user data and enforcing business rules.
* **UserRepository**: Provides database operations for CRUD operations on user entities.
* **User Entity**: Maps to the User table in the database with fields like userId, username, password, email, role, and createdDate.

**Workflow**

1. **User Registration**:
   * New users register via UserController, which interacts with UserService to create user entries in the database.
   * Passwords are encrypted using PasswordEncoderUtil.
2. **Profile Management**:
   * Users can fetch and update their profile details through UserController, using JWT for access control.

**4.3 Policy Management Module**

**Purpose**: Enables users to create, retrieve, update, and delete insurance policies.

**Components**

* **PolicyController**: Exposes REST APIs for policy operations.
* **PolicyService**: Contains business logic related to policy management.
* **PolicyRepository**: Interfaces with the H2 database for CRUD operations.
* **Policy Entity**: Represents an insurance policy, mapped to the Policy table.

**Workflow**

1. **Policy Creation**:
   * Users create policies through PolicyController, providing details like policy type, premium amount, and coverage period.
   * PolicyService performs validations and invokes PolicyRepository to save the policy.
2. **Policy Management**: Users can retrieve, update, or delete policies through PolicyController, with role-based permissions applied.

**4.4 Claim Management Module**

**Purpose**: Allows users to file claims against insurance policies.

**Components**

* **ClaimController**: Manages API endpoints for claim operations.
* **ClaimService**: Contains business rules and validation for claim processing.
* **ClaimRepository**: Interfaces with the database for CRUD operations.
* **Claim Entity**: Represents an insurance claim, mapped to the Claim table.

**Workflow**

1. **Filing a Claim**:
   * Users submit claims via ClaimController, linking them to existing policies.
   * Claims are validated and saved to the database, using ClaimService and ClaimRepository.
2. **Claim Management**: Users retrieve and update claim statuses and remarks, with role-based restrictions enforced.

**4.5 Email Notification Module**

**Purpose**: Sends notifications to users regarding registration, policy updates, and claim statuses.

**Components**

* **EmailService**: Manages email sending, SMTP configuration, and message preparation.
* **EmailTemplates**: Provides standardized templates for different email types.

**Workflow**

1. **Event-Based Trigger**: Certain events (e.g., registration, claim status updates) trigger email notifications.
2. **Email Generation**:
   * EmailService retrieves the relevant template from EmailTemplates, customizes it, and sends the email via SMTP.

**4.6 Utility Module**

**Purpose**: Provides helper functions and utilities for JWT, encryption, date formatting, and file handling.

**Components**

* **JwtTokenUtil**: Creates and validates JWT tokens.
* **PasswordEncoderUtil**: Encrypts passwords before saving to the database.
* **DateUtil**: Formats and parses dates.
* **FileUploadUtil**: Manages the file upload structure in the project’s main directory.

**5. Data Flow and Detailed Interactions**

**5.1 User Login and Authentication Flow**

1. **Login Request**: User submits credentials to AuthController.
2. **Token Generation**:
   * AuthController calls JwtTokenUtil to generate a JWT upon successful login.
   * JWT includes user role and expiration date.
3. **JWT Validation**: Incoming requests pass through JwtRequestFilter.
   * The token is validated, and user details are loaded for authorized requests.

**5.2 Policy Management Flow**

1. **Create Policy**:
   * PolicyController accepts policy details and passes them to PolicyService.
   * After validation, PolicyRepository saves the policy, associating it with the user’s ID.
2. **Retrieve Policy**: Users fetch policies through PolicyController, which calls PolicyRepository to retrieve policy details.

**5.3 Claim Management Flow**

1. **File Claim**:
   * ClaimController receives the claim details, including policy ID, and forwards it to ClaimService.
   * ClaimService verifies policy validity and other constraints, then ClaimRepository saves the claim in the database.
2. **Claim Status Update**: Claims can be updated or retrieved through ClaimController, with restricted access to prevent unauthorized updates.

**6. Database Design**

**6.1 Entity-Relationship Diagram**

The system includes three core tables: **User**, **Policy**, and **Claim** with the following relationships:

* **User (1)** ↔ **Policy (M)**: One user can have multiple policies.
* **Policy (1)** ↔ **Claim (M)**: Each policy can have multiple claims.

**6.2 Database Tables**

1. **User Table**:
   * Fields: userId (PK), username, password, email, role, createdDate
2. **Policy Table**:
   * Fields: policyId (PK), userId (FK), policyType, premiumAmount, startDate, endDate, status
3. **Claim Table**:
   * Fields: claimId (PK), policyId (FK), claimAmount, claimDate, status, remarks

**7. Security Design**

**7.1 Authentication**

* **JWT-Based Authentication**: Secures APIs with token-based authentication, preventing unauthorized access.
* **Token Expiry**: Tokens are time-bound, enhancing security by reducing the risk of token misuse.
* **RBAC**: Role-based access ensures that only authorized users can perform certain actions.

**7.2 Data Protection**

* **Password Encryption**: All passwords are encrypted using PasswordEncoderUtil.
* **HTTPS Recommendations**: For production, HTTPS is recommended to protect data in transit.

**8. Non-Functional Requirements**

**8.1 Performance**

* **Stateless Architecture**: Stateless design allows the system to scale horizontally.
* **Database Optimization**: Indexed columns for frequent queries, and caching is recommended for production.

**8.2 Reliability**

* **Retry Mechanism**: Email and database connections include retry logic to handle transient issues.
* **Transaction Management**: Ensures atomic operations across dependent modules.

**8.3 Scalability**

* **Horizontal Scaling**: Modular design and REST API enable easy scaling with load balancers.

**8.4 Security**

* **JWT with Short Expiry**: Enhances security by limiting token lifespan.
* **Encrypted Password Storage**: Prevents exposure of plain-text credentials.