# Insurance Management System (StateFarm Replica) - Full Stack Microservices Project

## 1. Project Overview
The **Insurance Management System** is a full-stack application designed to manage policies, claims, and payments for customers. The system follows a **Microservices Architecture** and is built using **Spring Boot (backend) and React.js (frontend)**. It includes **JWT-based authentication & authorization (Spring Security)**, **RESTful APIs**, and is deployed on **AWS (EC2, S3, Lambda) with Docker & Kubernetes**.

============
BACK-END
============

## 2. Functional Requirements

### **User Roles**
1. **Admin:**
   - Manage insurance policies
   - Approve/reject claims
   - View reports & analytics
2. **Agent:**
   - Register new users
   - Process claims & manage customer policies
3. **Customer:**
   - Purchase & renew policies
   - File insurance claims
   - Track claim status

### **Core Functionalities**
✅ **User Authentication & Authorization (JWT-based, later with Spring Security)**
✅ **Insurance Policy Management** (CRUD operations)
✅ **Claims Processing** (Filing, Approval, Rejection)
✅ **Payments & Premium Management**
✅ **Notifications & Alerts** (Policy renewal reminders, Claim updates)
✅ **Reports & Analytics** (Admin Dashboard, Sales Reports)
✅ **Document Management** (Upload claim-related documents)

## 3. Microservices Architecture

1️⃣ User Service (Auth & Profile Management)
- Handles **User Registration, Login (JWT), Role Management**

2️⃣ Policy Service
- Manages **Insurance Policy CRUD, Pricing, Validity, & Policy Details**

3️⃣ Claims Service
- Manages **Filing, Approval, Rejection, Status Tracking**

4️⃣ Payment Service
- Handles **Premium Payment, Invoices, Payment Gateway Integration**

5️⃣ Notification Service
- Sends **Email, SMS alerts** using AWS SNS

6️⃣ API Gateway
- **Single entry point** for frontend to access all microservices

---

## 4. Tech Stack

### **Backend (Spring Boot - Java)**
✅ Spring Boot, Spring Data JPA, Hibernate
✅ MySQL for Database
✅ RESTful API & JWT Security

### **Frontend (React.js)**
✅ React.js with Redux for State Management
✅ Material UI for Styling
✅ Axios for API Calls

### **Cloud & DevOps**
✅ **AWS:** EC2, S3, Lambda
✅ **Docker & Kubernetes** (Containerization & Orchestration)
✅ **Jenkins & GitLab CI/CD**
✅ **Monitoring:** Splunk, Dynatrace

I've expanded your database design with **more fields** and **additional sample data**, while ensuring proper **relationships** between tables. Here's the enhanced structure:

## **5. Database Design (MySQL)**

### **User Table**
Stores user details and roles (Admin, Agent, Customer).

ID	Username	Email	Password	Role	Full Name	Phone	Address	Date of Birth	Created At	Status (Active/Inactive)
1	john_doe	john@example.com	encrypted_pass1	Customer	John Doe	9876543210	123 Main St, NY	1990-05-15	2024-01-10 12:30:00	Active
2	jane_smith	jane@example.com	encrypted_pass2	Agent	Jane Smith	8765432109	456 Elm St, CA	1985-08-22	2024-02-05 14:00:00	Active
3	admin_1	admin@example.com	encrypted_pass3	Admin	Admin One	9988776655	789 Oak St, TX	1980-12-01	2024-01-01 10:00:00	Active

---

### **Policy Table** 2
Stores available policies with coverage details.

ID	Policy Name	Policy Type	Premium	Duration (Years)	Coverage Amount	Description	Created At
101	Health Plus	Health Insurance	5000	5	200000	Covers hospitalization expenses	2024-01-01 10:00:00
102	Car Secure	Auto Insurance	7000	3	500000	Covers vehicle damage	2024-02-15 11:30:00
103	Home Shield	Home Insurance	10000	10	1000000	Covers home damages & theft	2024-03-05 15:20:00

### **User_Policy Table (Mapping Users to Policies)**
Represents the **many-to-many** relationship between **Users and Policies**.

ID	User ID	Policy ID	Start Date	End Date	Status (Active/Expired)
1	1	101	2024-03-01	2029-03-01	Active
2	1	102	2024-04-01	2027-04-01	Active
3	2	103	2024-05-01	2034-05-01	Active

### **Claim Table**
Stores claim requests made by users for their policies.

ID	User ID	Policy ID	Claim Date	Claim Status (Pending/Approved/Rejected)	Claim Amount	Reason	Processed By (Agent ID)	Processed Date
201	1	101	2024-06-01	Pending	50000	Medical emergency	NULL	NULL
202	2	103	2024-06-05	Approved	300000	Fire damage	2	2024-06-10
203	1	102	2024-07-10	Rejected	7000	Minor vehicle dent	2	2024-07-12

### **Payment Table**
Tracks payments made by users.

ID	User ID	Policy ID	Amount	Payment Date	Status (Success/Failed)	Payment Mode (Card/UPI/NetBanking)	Transaction ID
301	1	101	5000	2024-03-15	Success	Card	TXN123456
302	2	103	10000	2024-05-10	Success	UPI	TXN789012
303	1	102	7000	2024-04-12	Failed	NetBanking	TXN345678

### **Database Relationships & Foreign Keys**
- **User ↔ User_Policy (One-to-Many)** → A user can have multiple policies.
- **Policy ↔ User_Policy (One-to-Many)** → A policy can be taken by multiple users.
- **User_Policy ↔ Claim (One-to-Many)** → A user can claim a policy multiple times.
- **User ↔ Payment (One-to-Many)** → A user can make multiple payments.
- **Policy ↔ Payment (One-to-Many)** → A policy can have multiple payments associated.
- **User (Agent) ↔ Claim (One-to-Many)** → An agent processes multiple claims.

### **Next Steps**
- ✅ Database schema with sample data.
- 🔜 Define **Spring Boot entity classes** with `@OneToMany`, `@ManyToOne`, `@ManyToMany` annotations.
- 🔜 Create **repository interfaces** using Spring Data JPA.
- 🔜 Develop **service and controller layers** for CRUD operations.
- 🔜 Implement **React components** for frontend integration.

Is this **database model sufficient**, or do you need any **more fields/relationships**? 🚀

### **High-Level Design for Insurance Management System (Microservices Architecture)**

## **Microservices in the System**
1. **User Service** (Authentication, roles, JWT security)
2. **Policy Service** (Insurance policies CRUD)
3. **Claim Service** (Handle policy claims)
4. **Payment Service** (Payments tracking)
5. **Notification Service** (Emails, SMS alerts)
6. **API Gateway** (Spring Cloud Gateway for routing requests)
7. **Config Server** (Centralized configuration management)

## **1. User Service**
Responsibilities:
- User authentication (JWT)
- Role-based authorization
- CRUD operations on users

### **Model Classes**
#### `User`
- `id: Long`
- `username: String`
- `email: String`
- `password: String`
- `role: Enum (ADMIN, AGENT, CUSTOMER)`
- `policies: List<Policy> (OneToMany)`

#### `Role`
- `id: Long`
- `name: Enum (ADMIN, AGENT, CUSTOMER)`

### **Repository Interfaces**
#### `UserRepository`
- `findByEmail(String email): Optional<User>`
- `findByUsername(String username): Optional<User>`

### **Service Interface**
#### `UserService`
- `registerUser(UserDTO userDto): User`
- `authenticateUser(String username, String password): String (JWT Token)`
- `getUserById(Long userId): User`
- `assignRoleToUser(Long userId, Role role): User`

## **2. Policy Service**
Responsibilities:
- Manage insurance policies
- Associate policies with users
- CRUD operations on policies

### **Model Classes**

#### `Policy`
- `id: Long`
- `name: String`
- `premium: Double`
- `duration: Integer`
- `coverageAmount: Double`
- `user: User (ManyToOne)`

### **Repository Interfaces**
#### `PolicyRepository`
- `findByUserId(Long userId): List<Policy>`

### **Service Interface**
#### `PolicyService`
- `createPolicy(PolicyDTO policyDto): Policy`
- `getPolicyById(Long policyId): Policy`
- `getPoliciesByUser(Long userId): List<Policy>`
- `updatePolicy(Long policyId, PolicyDTO policyDto): Policy`
- `deletePolicy(Long policyId)`

## **3. Claim Service**
Responsibilities:
- Manage policy claims
- Approval/rejection of claims

### **Model Classes**
#### `Claim`
- `id: Long`
- `policy: Policy (ManyToOne)`
- `user: User (ManyToOne)`
- `claimAmount: Double`
- `status: Enum (PENDING, APPROVED, REJECTED)`

### **Repository Interfaces**
#### `ClaimRepository`
- `findByUserId(Long userId): List<Claim>`
- `findByPolicyId(Long policyId): List<Claim>`

### **Service Interface**
#### `ClaimService`
- `createClaim(ClaimDTO claimDto): Claim`
- `approveClaim(Long claimId): Claim`
- `rejectClaim(Long claimId, String reason): Claim`
- `getClaimsByUser(Long userId): List<Claim>`
- `getClaimsByPolicy(Long policyId): List<Claim>`

## **4. Payment Service**
Responsibilities:
- Track payments
- Handle payment success/failure

### **Model Classes**
#### `Payment`
- `id: Long`
- `user: User (ManyToOne)`
- `policy: Policy (ManyToOne)`
- `amount: Double`
- `paymentDate: LocalDateTime`
- `status: Enum (SUCCESS, FAILED)`

### **Repository Interfaces**
#### `PaymentRepository`
- `findByUserId(Long userId): List<Payment>`
- `findByPolicyId(Long policyId): List<Payment>`

### **Service Interface**
#### `PaymentService`
- `processPayment(PaymentDTO paymentDto): Payment`
- `getPaymentsByUser(Long userId): List<Payment>`
- `getPaymentsByPolicy(Long policyId): List<Payment>`

## **5. Notification Service**
Responsibilities:
- Send notifications (email, SMS)
- Notify users about policy renewals, claim updates

### **Model Classes**
#### `Notification`
- `id: Long`
- `user: User (ManyToOne)`
- `message: String`
- `status: Enum (SENT, FAILED)`

### **Repository Interfaces**
#### `NotificationRepository`
- `findByUserId(Long userId): List<Notification>`

### **Service Interface**
#### `NotificationService`
- `sendNotification(Long userId, String message): Notification`
- `getNotificationsByUser(Long userId): List<Notification>`

## **6. API Gateway (Spring Cloud Gateway)**
Responsibilities:
- Route API requests to appropriate microservices
- Implement security filters (JWT authentication)

### **Filters & Routes**
- `POST /api/users/login → User Service`
- `GET /api/users/{id} → User Service`
- `GET /api/policies/{id} → Policy Service`
- `POST /api/claims → Claim Service`
- `POST /api/payments → Payment Service`
- `POST /api/notifications → Notification Service`

## **7. Configuration Server (Spring Cloud Config)**
Responsibilities:
- Centralized configuration management
- Store service configurations in Git

## **Database Relationships**
- `User (OneToMany) → Policies`
- `User (OneToMany) → Claims`
- `User (OneToMany) → Payments`
- `Policy (OneToMany) → Claims`
- `Policy (OneToMany) → Payments`

## **Next Steps**
✅ **Define Entity Classes with Annotations**
🔜 **Create Repository Interfaces using Spring Data JPA**
🔜 **Develop Service and Controller Layers**
🔜 **Implement React Components for Frontend**
🔜 **Secure APIs using Spring Security & JWT**
🔜 **Deploy Microservices using Docker & Kubernetes on AWS**

===========
FRONT - END

===========
### **Frontend Design for Insurance Management System (React.js)**
The frontend will be developed using **React.js**, following a modular **component-based architecture**. It will communicate with the backend microservices via **REST APIs** exposed by the Spring Cloud API Gateway.

## **📁 Folder Structure**
```
insurance-frontend/
│── public/                     # Static assets
│── src/
│   ├── api/                    # API service calls
│   ├── components/             # Reusable UI components (Navbar, Sidebar, etc.)
│   ├── pages/                  # Main pages (Login, Dashboard, Policies, etc.)
│   ├── store/                   # State management (Redux/Context API)
│   ├── App.js                   # Main React Component
│   ├── index.js                 # Entry point
│   ├── routes.js                # Route definitions
│   ├── styles/                  # Global styles
│── .env                         # Environment variables
│── package.json                 # Dependencies
```

## **📌 Key Pages & Functionalities**
### **1. Authentication Pages**
- **Login Page**
  - Takes **email & password**.
  - Sends request to `/api/users/login` (User Service via API Gateway).
  - Stores **JWT token** after successful authentication.

- **Register Page**
  - Takes **user details (name, email, password, role)**.
  - Calls `/api/users/register`.

- **Protected Routes**
  - Routes restricted to **authenticated users only**.
  - Uses **JWT stored in LocalStorage** for authentication.

### **2. Dashboard**
- Displays:
  - **User details** (fetched from User Service).
  - **Active policies** (from Policy Service).

  - **Pending claims** (from Claim Service).
  - **Recent payments** (from Payment Service).
  - **Notifications** (from Notification Service).

### **3. Policies Page**
- Displays **all policies owned by the user**.
- Allows **searching & filtering policies**.
- Provides an option to **purchase new policies** by making payments.

### **4. Claims Page**
- Lists **all user claims** with status **(Pending, Approved, Rejected)**.
- Allows users to **submit a new claim**.
- Sends request to `/api/claims`.

### **5. Payments Page**
- Displays **user’s payment history**.
- Provides an option to **make new payments**.
- Calls `/api/payments`.

### **6. Notifications Page**
- Fetches **system notifications** from the Notification Service.
- Displays **policy renewals, claim status updates, payment confirmations**.

### **7. Profile Page**
- Allows users to **update profile details**.
- Provides an option to **change passwords**.
- Integrates with `/api/users/update`.

## **🔌 API Integration**
Each microservice (User, Policy, Claim, Payment) has an **API client** inside `/src/api/`.
- Calls will be made to API Gateway (`/api/` prefix).
- **JWT tokens** will be attached to requests.

## **🔐 Security: JWT Authentication**
- **React will restrict access** to protected pages using JWT authentication.
- **Users must be logged in** to access Dashboard, Policies, Claims, and Payments.
- **JWT is stored in Redux/LocalStorage** and attached to API requests.

## **📌 Next Steps**
✅ **Define React folder structure & components**
✅ **Define API integrations for microservices**
🔜 **Develop UI using Material UI / Tailwind CSS**
🔜 **Implement JWT authentication logic**
🔜 **Connect React with Dockerized backend services**
🔜 **Deploy frontend to AWS S3**

Let me know if you need more details! 🚀

## 7. Deployment (Docker + Kubernetes + AWS)
- **Backend:**
  - Containerized microservices using **Docker**
  - Deployed using **Kubernetes**
  - Hosted on **AWS EC2**
- **Frontend:**
  - Built with React.js
  - Hosted on **AWS S3 + CloudFront**

## 8. Test Cases (JUnit & Mockito for Spring Boot)
✅ User Authentication Test
✅ Policy Creation Test
✅ Claim Processing Test
✅ API Response Validation

## 9. Future Enhancements (Spring Security & More)
🔒 **Spring Security for Role-Based Access Control**
📈 **AI-based Fraud Detection for Claims**
📲 **Mobile App Integration**

### **Conclusion**
This is the **complete blueprint** for your **Insurance Management System** 🚀.
Let me know if you need additional details! 😊

Got it! You want a **multi-module Spring Boot project** structured inside a **`smart-insurance` folder**, containing all microservices, with **MySQL, Spring Cloud API Gateway, Eureka Server**, and necessary configurations.

I'll explain:
✅ **Project Structure**
✅ **Configurations for Each Microservice**
✅ **Required Dependencies & Their Roles**
✅ **How Everything Works Together**

## **1. Project Structure**
```
smart-insurance/
├── pom.xml  # Parent POM
├── api-gateway/  # Gateway Service
├── claim-service/
├── config-server/  # Centralized Configuration
├── discovery-service/  # Eureka Server
├── notification-service/
├── payment-service/
├── policy-service/
├── user-service/
```

Each microservice will have its **own `pom.xml`**, and dependencies will be managed centrally.

## **2. Parent POM (`smart-insurance/pom.xml`)**
This is the **root `pom.xml`** that manages dependencies for all microservices.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.4.4</version>
        <relativePath/>
    </parent>

    <groupId>com.insurance</groupId>
    <artifactId>smart-insurance</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>smart-insurance</name>
    <description>Smart Insurance Project</description>

    <modules>
        <module>api-gateway</module>
        <module>claim-service</module>
        <module>config-server</module>
        <module>discovery-service</module>
        <module>notification-service</module>
        <module>payment-service</module>
        <module>policy-service</module>
        <module>user-service</module>
    </modules>

    <properties>
        <java.version>17</java.version>
        <spring-cloud.version>2023.0.0</spring-cloud.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

</project>
```
### **Why These Configurations?**
✅ `packaging` as `pom`: Makes it a **multi-module project**
✅ **Spring Boot Parent**: Manages versions of Spring dependencies
✅ `modules`: Lists all microservices

## **3. Config Server (`config-server/`)**
This **centralizes all configurations** (database, API keys, etc.).

### **Dependencies (`config-server/pom.xml`)**
```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

### **Application Properties (`application.yml`)**
```yaml
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-repo/configs
```
### **How It Works?**
1. Clients (microservices) **fetch configs** from the Git repo.
2. Running on port **8888**.

## **4. Eureka Discovery Service (`discovery-service/`)**
Registers all microservices so they can find each other dynamically.

### **Dependencies (`discovery-service/pom.xml`)**
```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>
</dependencies>
```

### **Configuration (`application.yml`)**
```yaml
server:
  port: 8761

eureka:
  instance:
```

~~~~
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
```

### **How It Works?**
✅ **Runs on port 8761**
✅ **Microservices register** here
✅ **Load balancing** for microservices

---

## **5. API Gateway (`api-gateway/`)**
Handles **routing & authentication**.

### **Dependencies (`api-gateway/pom.xml`)**
```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
```

### **Configuration (`application.yml`)**
```yaml
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://USER-SERVICE
          predicates:
            - Path=/users/**
        - id: policy-service
          uri: lb://POLICY-SERVICE
          predicates:
            - Path=/policies/**

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

### **How It Works?**
✅ **Acts as an entry point**
✅ **Routes requests** to microservices
✅ **Uses Eureka** for load balancing

---
~~~~

## **6. Microservices (`user-service/`, `policy-service/`, etc.)**
Each microservice has:
1. **Spring Boot Starter Dependencies**
2. **Spring Data JPA for MySQL**
3. **Eureka Client for Service Discovery**

### **Dependencies (`user-service/pom.xml`)**
```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
```

### **Configuration (`application.yml`)**
```yaml
server:
  port: 8081

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/smart_insurance
    username: root
    password: root
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

### **How It Works?**
✅ **Connects to MySQL**
✅ **Registers with Eureka**
✅ **CRUD operations via Spring Data JPA**

## **Next Steps**
🔹 **Create Repository Interfaces for Microservices**
🔹 **Implement Service and Controller Layers**
🔹 **Secure APIs with JWT Authentication**
🔹 **Deploy on Docker & Kubernetes**

Would you like me to create **entity classes, repository interfaces, or service layers** next? 🚀

### **Securing APIs with JWT Authentication & Deployment Using Docker, Kubernetes, and AWS**

## **1. Securing APIs with JWT Authentication**
Why JWT?
JWT (JSON Web Token) provides **stateless authentication**, meaning there is no need to store session data on the server.

### **Steps to Implement JWT Authentication**
1. **User Logs In** → API validates credentials and generates a **JWT token**.
2. **User Makes Requests** → JWT token is sent in the **Authorization header**.
3. **Backend Verifies JWT** → If valid, the request is processed; otherwise, access is denied.

### **Key Components:**
- **JWT Filter** (Intercepts requests, validates tokens).
- **Authentication Controller** (Handles login, token generation).
- **UserDetailsService** (Fetches user details).
- **Security Configuration** (Defines access control rules).

#### **Dependencies (`pom.xml`)**
```xml
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

#### **Generate JWT Token (JWTUtil.java)**
```java
public String generateToken(String username) {
    return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60)) // 1 hour validity
            .signWith(SignatureAlgorithm.HS256, "secret_key")
            .compact();
}
```

#### **Spring Security Configuration**
```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
```

        http.csrf().disable()
            .authorizeHttpRequests()
            .requestMatchers("/auth/login", "/auth/register").permitAll()
            .anyRequest().authenticated()
            .and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        return http.build();
    }
}

🔹 **Outcome**: APIs are protected, and only authenticated users with a valid JWT can access them.

## **2. Deploying with Docker, Kubernetes, and AWS**

### **Step 1: Create a Docker Image**
1. **Add a `Dockerfile`** in each microservice:
```dockerfile
FROM openjdk:17-jdk-slim
COPY target/user-service.jar user-service.jar
ENTRYPOINT ["java", "-jar", "user-service.jar"]
```
2. **Build & Run Docker Container**
```sh
docker build -t user-service .
docker run -p 8081:8081 user-service
```

### **Step 2: Deploy with Kubernetes**
1. **Create a Deployment YAML (`user-deployment.yaml`)**
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: user-service:latest
        ports:
        - containerPort: 8081
```
2. **Apply Kubernetes Deployment**
```sh
kubectl apply -f user-deployment.yaml
```
🔹 **Outcome**: Service is **scalable** and **highly available**.

### **Step 3: Deploy to AWS (EC2, S3, Lambda, RDS)**
1. **Push Docker Images to AWS Elastic Container Registry (ECR)**
```sh
aws ecr create-repository --repository-name smart-insurance
docker tag user-service:latest
<aws_account_id>.dkr.ecr.<region>.amazonaws.com/user-service
docker push <aws_account_id>.dkr.ecr.<region>.amazonaws.com/user-service
```
2. **Deploy to AWS Elastic Kubernetes Service (EKS)**
```sh
eksctl create cluster --name smart-insurance-cluster --region us-east-1
kubectl apply -f user-deployment.yaml
```

### **Final Outcome**
✅ **Secure APIs** using JWT
✅ **Deploy with Docker**
✅ **Scale with Kubernetes**
✅ **Host on AWS**

### **Next Steps**
🔹 Implement **Monitoring (Splunk, Dynatrace)**
🔹 Configure **CI/CD with Jenkins, GitLab**

What do you want to do next? 🚀

## **CI/CD Pipeline for Smart Insurance using GitLab**

### **Why CI/CD?**
🔹 Automates build, test, and deployment.
🔹 Ensures **faster & reliable** software delivery.
🔹 Deploys microservices to **AWS (EKS, EC2, or S3)**.

## **1. GitLab CI/CD Pipeline Overview**
✅ **Code Push** → **GitLab Runner** triggers pipeline
✅ **Build & Test** → Runs unit tests, creates a Docker image
✅ **Push to AWS ECR** → Uploads Docker images
✅ **Deploy to Kubernetes (EKS)** → Runs microservices

## **2. Create `.gitlab-ci.yml` File**
This file defines **stages**:
🔹 `build` → Compile & create Docker images
🔹 `test` → Run unit tests
🔹 `deploy` → Push images & deploy to AWS

```yaml
```

```yaml
stages:
  - build
  - test
  - deploy

variables:
  AWS_REGION: "us-east-1"
  ECR_REPO: "<aws_account_id>.dkr.ecr.${AWS_REGION}.amazonaws.com/smart-insurance"
  KUBE_CONFIG: "/root/.kube/config"

before_script:
  - echo "Logging into AWS ECR"
  - aws ecr get-login-password --region $AWS_REGION | docker login --username AWS --password-stdin $ECR_REPO

build:
  stage: build
  script:
    - echo "Building Docker image"
    - docker build -t user-service .
    - docker tag user-service:latest $ECR_REPO/user-service:latest
    - docker push $ECR_REPO/user-service:latest

test:
  stage: test
  script:
    - echo "Running Unit Tests"
    - mvn test

deploy:
  stage: deploy
  script:
    - echo "Deploying to Kubernetes"
    - kubectl apply -f k8s/user-deployment.yaml
```

## **3. Set Up GitLab Runner**

1. Install GitLab Runner:

```sh
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash
sudo apt install gitlab-runner
```

2. Register the runner:

```sh
gitlab-runner register
```

## **4. Deploying to AWS (EKS)**

1️⃣ Create AWS EKS Cluster

```sh
eksctl create cluster --name smart-insurance-cluster --region us-east-1
```

2️⃣ Deploy Microservices

```sh
kubectl apply -f k8s/user-deployment.yaml
kubectl get pods
```

## **Final Outcome**
✅ **Automated Build, Test & Deployment**
✅ **Microservices deployed to AWS EKS**
✅ **Faster and scalable CI/CD pipeline**

## **Next Steps**
🔹 **Monitoring & Logging (Splunk, Dynatrace)**
🔹 **Set Up API Gateway Rules**

What’s your next priority? 🚀

EXPLANATION :

## **CI/CD Pipeline for Smart Insurance using GitLab, Docker, Kubernetes, and AWS**

### **Why CI/CD?**
🔹 **Continuous Integration (CI):** Ensures that code is always tested and built automatically.
🔹 **Continuous Deployment (CD):** Deploys services automatically when code changes are pushed.
🔹 **Automation Benefits:** Reduces manual work, improves reliability, and speeds up the release cycle.

## **1. GitLab CI/CD Pipeline Overview**
- **Developers push code to GitLab** → **GitLab Runner triggers CI/CD pipeline**
- **Pipeline runs stages:**
  ✅ `build` → Compiles Java code & builds Docker images
  ✅ `test` → Runs unit tests using JUnit & Spring Boot
  ✅ `deploy` → Pushes Docker images to AWS ECR & deploys to Kubernetes (AWS EKS)

## **2. `.gitlab-ci.yml` Configuration**
The **`.gitlab-ci.yml`** file defines all the CI/CD stages and steps.

```yaml
stages:
  - build
  - test
  - deploy

variables:
  AWS_REGION: "us-east-1"
  ECR_REPO: "<aws_account_id>.dkr.ecr.${AWS_REGION}.amazonaws.com/smart-insurance"
  KUBE_CONFIG: "/root/.kube/config"

before_script:
  - echo "Logging into AWS ECR"
  - aws ecr get-login-password --region $AWS_REGION | docker login --username AWS --password-stdin $ECR_REPO
```

### **Explanation:**
- **Stages:** Defines the order in which tasks execute (build → test → deploy).
- **Variables:** Stores AWS region, repository details, and Kubernetes config path.
- **Before Script:** Logs into **AWS Elastic Container Registry (ECR)** to store Docker images.

### **3. Build Stage**
```yaml
build:
  stage: build
  script:
    - echo "Building Docker image"
    - docker build -t user-service .
    - docker tag user-service:latest $ECR_REPO/user-service:latest
    - docker push $ECR_REPO/user-service:latest
```

### **Explanation:**
✅ **Builds a Docker image** for the microservice.
✅ **Tags the image** with AWS ECR repository URL.
✅ **Pushes the image** to AWS ECR for later use in Kubernetes.

### **4. Test Stage**
```yaml
test:
  stage: test
  script:
    - echo "Running Unit Tests"
    - mvn test
```

### **Explanation:**
✅ Runs **unit tests** to verify that the service is functioning correctly.
✅ Uses **Maven (`mvn test`)** to execute JUnit tests.

### **5. Deploy Stage**
```yaml
deploy:
  stage: deploy
  script:
    - echo "Deploying to Kubernetes"
    - kubectl apply -f k8s/user-deployment.yaml
```

### **Explanation:**
✅ **Uses `kubectl` to deploy the microservice to AWS EKS**.
✅ Reads the **deployment YAML file** (`k8s/user-deployment.yaml`).

## **3. Setting Up GitLab Runner**
GitLab Runner is required to **execute CI/CD jobs** on your machine or cloud.

### **Install GitLab Runner**
Run the following commands on the server:
```sh
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/
```

~~~
script.deb.sh | sudo bash
sudo apt install gitlab-runner
```

### **Register GitLab Runner**
```sh
gitlab-runner register
```
This links the runner with your GitLab project.

---

## **4. Deploying Microservices to AWS (EKS)**

### **Step 1: Create AWS EKS Cluster**
```sh
eksctl create cluster --name smart-insurance-cluster --region us-east-1
```
✅ **Creates an AWS Kubernetes Cluster (EKS).**
✅ **Manages service scaling & auto-healing.**

---

### **Step 2: Deploy Microservices to EKS**
Each microservice has a **Kubernetes deployment YAML file** (`k8s/user-deployment.yaml`):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/user-service:latest
        ports:
        - containerPort: 8081
```

Run the deployment command:
```sh
kubectl apply -f k8s/user-deployment.yaml
kubectl get pods
```

✅ **Deploys the user-service to AWS EKS.**
✅ **Scales to multiple replicas automatically.**

---

## **Final Outcome**
✅ **Automated Build, Test & Deployment**
✅ **Microservices deployed to AWS EKS**
~~~

✅ **CI/CD pipeline ensures continuous integration & delivery**

## **Next Steps**
🔹 **Monitoring & Logging (Splunk, Dynatrace)**
🔹 **Configure API Gateway & Load Balancer**

What’s your next priority? 🚀

## **Configuring API Gateway & Load Balancer in Smart Insurance**

### **Why Use API Gateway?**
✅ **Single Entry Point:** Routes requests to appropriate microservices.
✅ **Security:** Enforces authentication (JWT), rate limiting, and CORS policies.
✅ **Load Balancing:** Distributes traffic efficiently.
✅ **Service Discovery:** Works with **Eureka** for dynamic service registration.

## **1. Setting Up Spring Cloud API Gateway**
We will use **Spring Cloud Gateway** to handle all incoming requests.

### **Add Dependencies (`api-gateway/pom.xml`)**
```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```
✅ **Spring Cloud Gateway** for API routing
✅ **Eureka Client** for service discovery
✅ **Spring Security** for authentication

### **2. Configuring `application.yml` (API Gateway)**
```yaml
server:
  port: 8080

spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        - id: user-service
```

```
          uri: lb://USER-SERVICE
          predicates:
            - Path=/users/**
        - id: policy-service
          uri: lb://POLICY-SERVICE
          predicates:
            - Path=/policies/**

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```
### **Explanation:**
✅ **Routes Requests** based on URL patterns (`/users/** → User Service`).
✅ **Load Balancer (`lb://`)** dynamically resolves microservice instances via Eureka.
✅ **Eureka Client** connects to Discovery Service (`http://localhost:8761`).

## **3. Enabling JWT Authentication in API Gateway**
Add a security filter to validate JWT tokens.

### **Create `JwtAuthFilter.java`**
```java
@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {
        String authHeader = request.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String token = authHeader.substring(7);
            if (!jwtUtil.validateToken(token)) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Invalid
JWT Token");
                return;
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

✅ **Extracts JWT token from the request header.**
✅ **Validates token before forwarding the request.**

## **4. Register API Gateway in Eureka**
Add this in **`api-gateway/src/main/resources/application.yml`**:
```yaml
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
```

    hostname: localhost
~~~
✅ **API Gateway will now be discoverable in Eureka.**

---

## **5. Setting Up AWS Load Balancer (ALB)**
We use **AWS Application Load Balancer (ALB)** to distribute traffic to the API Gateway.

### **Step 1: Create Load Balancer on AWS**
1. Go to **AWS Console → EC2 → Load Balancers**.
2. Click **Create Load Balancer → Application Load Balancer**.
3. **Add Listeners** → Port `80` → Forward to API Gateway.
4. Register **API Gateway** instance as a **Target Group**.

### **Step 2: Update Security Groups**
- Allow inbound traffic on port **80, 443 (HTTPS)**.
- Restrict access to only trusted IPs.

---

## **Final Outcome**
✅ **Spring Cloud Gateway** routes requests dynamically.
✅ **JWT Authentication** ensures security.
✅ **AWS Load Balancer** distributes traffic efficiently.

---

## **Next Steps**
🔹 **Monitoring & Logging (Splunk, Dynatrace)**
🔹 **Scaling & Performance Optimization**

What do you want to focus on next? 🚀

## **Monitoring & Logging with Splunk and Dynatrace in Smart Insurance**

### **Why Monitoring & Logging?**
✅ **Detect Issues Quickly** → Monitor service performance and failures in real-time.
✅ **Centralized Logging** → Store logs in one place for easy debugging.
✅ **Automated Alerts** → Receive notifications for errors and performance bottlenecks.

---

## **1. Setting Up Centralized Logging with Splunk**
🔹 **Splunk** collects, indexes, and analyzes logs from all microservices.

### **Step 1: Add Splunk Logging Dependencies (`pom.xml`)**
```xml
<dependency>
    <groupId>com.splunk.logging</groupId>
    <artifactId>splunk-library-javalogging</artifactId>
    <version>1.8.0</version>
</dependency>
~~~
✅ Enables **Splunk HTTP Event Collector (HEC)** integration.

### **Step 2: Configure Splunk Logging (`logback.xml`)**
Create `logback.xml` inside `src/main/resources`:
```xml
<configuration>
    <appender name="SPLUNK"
class="com.splunk.logging.HttpEventCollectorLogbackAppender">
        <url>http://splunk-server:8088</url>
        <token>YOUR_SPLUNK_HEC_TOKEN</token>
        <source>smart-insurance-logs</source>
        <sourcetype>_json</sourcetype>
        <index>main</index>
    </appender>

    <root level="INFO">
        <appender-ref ref="SPLUNK"/>
    </root>
</configuration>
```
### **Explanation:**
✅ **`url`** → Splunk server endpoint (replace with actual IP).
✅ **`token`** → Authentication token from Splunk HEC.
✅ **`index`** → Stores logs in **Splunk's main index**.

### **Step 3: Test Logging in Microservices**
Modify `UserService.java`:
```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
@RequestMapping("/users")
public class UserController {
    private static final Logger LOGGER =
LoggerFactory.getLogger(UserController.class);

    @GetMapping("/{id}")
    public ResponseEntity<String> getUser(@PathVariable String id) {
        LOGGER.info("Fetching user with ID: {}", id);
        return ResponseEntity.ok("User details...");
    }
}
```
✅ Logs will be **sent to Splunk** whenever a request is made.

## **2. Setting Up Dynatrace for Real-Time Monitoring**
🔹 **Dynatrace** provides AI-powered observability for microservices.

### **Step 1: Install Dynatrace OneAgent on the Server**
```sh
wget -O Dynatrace-OneAgent.sh
"https://YOUR_DYNATRACE_URL/api/v1/deployment/installer?token=YOUR_ACCESS_TOKEN"
sudo sh Dynatrace-OneAgent.sh --install
```
✅ Installs **Dynatrace Agent** for automatic monitoring.

### **Step 2: Configure Dynatrace in `application.yml`**
```yaml
management:
```

  metrics:
    export:
      dynatrace:
        uri: https://YOUR_DYNATRACE_URL/api/v2/metrics
        api-token: YOUR_DYNATRACE_TOKEN
~~~
✅ **Dynatrace collects performance metrics** for each microservice.

---

### **Step 3: Enable Actuator Endpoints**
Modify `pom.xml`:
```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
~~~
Modify `application.yml`:
```yaml
management:
  endpoints:
    web:
      exposure:
        include: "health,metrics"
  tracing:
    sampling:
      probability: 1.0
```
✅ **`/actuator/metrics`** endpoint provides service-level monitoring.

## **3. Set Up Alerts in Splunk & Dynatrace**
🔹 **Splunk Alerts:**
1. **Create Alert Query:**
   ```splunk
   index="main" source="smart-insurance-logs" error
   ```
2. **Set Trigger Condition:** If logs contain `"error"`, send an alert.
3. **Notification:** Configure **Slack, Email, or PagerDuty** alerts.

🔹 **Dynatrace Alerts:**
1. **Go to Dynatrace → Problem Detection & Alerts**.
2. **Define Thresholds:** Set CPU, Memory, and Response Time limits.
3. **Enable Notifications:** Integrate with **Slack, PagerDuty, or Email**.

## **Final Outcome**
✅ **Splunk captures real-time logs from all microservices**.
✅ **Dynatrace monitors CPU, Memory, and Response Time**.
✅ **Automated Alerts notify developers of system issues**.

## **Next Steps**
🔹 **Scaling & Performance Optimization**
🔹 **Error Handling & Retry Mechanisms**

What do you want to focus on next? 🚀