**✅ Learning Roadmap**

**1. Fundamentals of Microservices**

* What are microservices?
* Monolith vs Microservices
* Pros & Cons

**2. Spring Boot Basics (if not already covered)**

* Spring Boot project setup
* REST API creation
* Dependency Injection
* Spring Data JPA & Hibernate

**3. Microservice Architecture Essentials**

* Service Registry (Eureka)
* API Gateway (Spring Cloud Gateway)
* Configuration Server (Spring Cloud Config)
* Communication (RestTemplate, WebClient, Feign Client)
* Load Balancing (Ribbon/Resilience4j)
* Circuit Breaker (Resilience4j)

**4. Security in Microservices**

* JWT-based Authentication & Authorization
* Role-based access with Spring Security

**5. Production Readiness**

* Centralized Logging (ELK Stack / Sleuth + Zipkin)
* Monitoring (Micrometer + Prometheus + Grafana)
* Docker & Kubernetes (basic containerization & deployment)

**✅ Microservices + Project Roadmap (with Key Concepts)**

| **🔢** | **Step** | **Description** |
| --- | --- | --- |
| 1️ | ✅ **Eureka Server** | Central service registry (you’ve already done this ✅) |
| 2️ | **Config Server** | Centralized configuration for all microservices |
| 3️ | **User Service** | First real microservice: registers with Eureka & uses config server |
| 4️ | **API Gateway** | Entry point for all microservices. Routes requests + filters + security |
| 5️ | **Policy Service** | Another microservice: communicates with User Service (via Feign) |
| 6️ | **Claim Service** | Third microservice: files/view claims for a user |
| 7️ | **Service Communication** | Use **Feign Client**, **WebClient** for inter-service calls |
| 8️ | **JWT + Spring Security** | Secure services; authenticate users |
| 9️ | **Resilience4j** | Fault tolerance: **Circuit Breaker**, **Retry**, **Rate Limiter** |
| 10 | **Logging & Tracing** | Sleuth + Zipkin for distributed tracing |
| 1️1 | **Dockerize** | Containerize services |
| 1️2 | **Deploy with Docker Compose / Kubernetes** | Run as a complete system |

**Step 1: Introduction to Microservices Architecture**

**🔹 What are Microservices?**

Microservices is an architectural style that structures an application as a **collection of small autonomous services**, each modeled around a business domain.

🧠 Example from our project: We'll have separate services like:

* User Service for user data
* Policy Service for insurance policies
* Claim Service for insurance claims

Each service:

* Has its own database
* Can be developed & deployed independently
* Communicates with other services via **REST APIs**

**🔹 Benefits of Microservices:**

* Independent deployment
* Technology diversity
* Better scalability
* Fault isolation

**🔹 Challenges:**

* Complex inter-service communication
* Distributed data management
* Deployment complexity
* Monitoring and logging overhead

**🛠️ Real-Time Project Setup: Online Insurance Platform**

We’ll build this using:

* Spring Boot for microservices
* Spring Cloud (Eureka, Gateway, Config Server)
* Spring Security + JWT
* MySQL

**✅ Let's Start with Eureka Server (Service Registry)**

Why Eureka?

Eureka is a service registry where each microservice registers itself, making it discoverable by others

Think of Eureka like a phone book for microservices.

**Why Do We Need Eureka Server?**

**🔹 1. Service Discovery in Microservices**

In a microservices architecture:

* You don’t have just one application.
* You have many small services like user-service, policy-service, claim-service, etc.
* These services **need to talk to each other** over the network.

But here's the problem:

| **🧨 Challenge** | **🤔 Why It’s a Problem** |
| --- | --- |
| **Hardcoding URLs/IPs** | If one service changes its IP/port (e.g., due to scaling or crash), others will fail to connect. |
| **Load balancing** | You can't easily balance traffic between multiple instances of a service manually. |

**🔹 2. Eureka Solves This with Dynamic Service Discovery**

With Eureka:

* Each microservice registers itself with Eureka Server when it starts.
* Eureka keeps a list (registry) of all available services and their current IP/port.
* When policy-service wants to talk to user-service, it doesn't need to know its IP or port.
* It simply asks Eureka:  
  👉 *"Hey, where is user-service?"*

✅ Eureka replies with:

"user-service is running on IP X.X.X.X at port XXXX"

🎯 Eureka also **load balances** between multiple instances of the same service.

**🔄 How Services Use Eureka**

**🧾 Microservices Do Two Things:**

1. **Register with Eureka** (like saying “I’m alive!”)
2. **Fetch other services** from Eureka (service discovery)

Spring Boot + Spring Cloud makes this super easy:

* Just annotate with @EnableDiscoveryClient and add a dependency.
* Eureka takes care of heartbeats, self-preservation, and instance registry.

**🔧 Real World Example in Our Project**

Say you have:

* user-service running on localhost:8081
* policy-service running on localhost:8082

When both register with Eureka:

* user-service says: “Eureka, I’m here!” ✅
* policy-service says: “Eureka, I’m also here!” ✅

Now, when:

* policy-service wants to get user info,
* It doesn’t need to call http://localhost:8081/users
* It calls http://user-service/users
* Eureka resolves that to the correct host/port.

**⚠️ Without Eureka?**

If we didn’t use Eureka:

* We'd hardcode host:port for each service.
* Wouldn’t scale to real-world environments (Kubernetes, Docker, etc.).
* Resilience and dynamic scaling become very difficult

**What is Spring Cloud Config Server?:**

The **Config Server** provides a **centralized way** to manage the **external configuration** of all your microservices.

In a microservices architecture, each service needs some config:

* Port numbers
* Database URLs
* Feature flags
* API keys
* Logging settings
* … and more

Instead of putting these in **each service’s application.yml**, we can keep **all configs in one place** — the **Config Server**.

**💡 Why Do We Need a Config Server?**

| **Problem Without Config Server** | **How Config Server Helps** |
| --- | --- |
| Each service has its own config file | One centralized place (usually Git) for all configs |
| Updating config means re-deploying services | Change config in Git → Auto-refresh configs in services |
| Difficult to manage configs in 10+ microservices | Manage configs in one repo, organized by service name |
| No version control for config | Use Git as a versioned backend |

**🧾 Real-World Example:**

In our Insurance Project, we’ll have:

| **Microservice** | **Config File in Git/Folder** |
| --- | --- |
| user-service | user-service.yml |
| policy-service | policy-service.yml |
| claim-service | claim-service.yml |
| Common config | application.yml (shared settings) |

We’ll store these in a **Git repo**, and the **Config Server** will serve them to each microservice at runtime.

**⚙️ How It Works**

**📤 Config Server**

* Acts as a **pull server**
* Fetches configuration from:
  + GitHub repo **(most common)**
  + Local folder (during development)

**📥 Client Microservice**

* Adds spring.config.import=configserver: in its bootstrap config
* At startup, it fetches its config by name (e.g., user-service.yml)

**🔄 Flow Diagram**

+--------------------+ +----------------------+

| user-service | | policy-service |

| (spring client) | | (spring client) |

+--------+-----------+ +----------+-----------+

| |

| Requests config from Config |

| Server using app name |

| |

v v

+-------------------------------+ |

| Spring Cloud Config Server |<----+

| Fetches from Git repo |

+-------------------------------+

**🔧 Example Git Config Repo**

In your Git repo (e.g., https://github.com/yourname/microservice-configs):

📂 microservice-configs

│

├── application.yml

├── user-service.yml

├── policy-service.yml

└── claim-service.yml

**Example: user-service.yml**

server:

port: 8081

spring:

datasource:

url: jdbc:mysql://localhost:3306/userdb

username: root

password: root

**🛡️ Benefits**

| **Feature** | **Benefit** |
| --- | --- |
| Centralized Config | One location for all configs |
| Dynamic Refresh | Auto-refresh values at runtime (with @RefreshScope) |
| Environment Support | Different config for dev, test, prod |
| Git-backed Versioning | Track who changed what and when |

**✅ Summary**

| **Term** | **Meaning** |
| --- | --- |
| **Config Server** | Central server to manage external configs |
| **Git Repo** | Backend store for all config files |
| **Client Microservice** | A service that fetches its config from the server |
| **application.yml** | Common shared config for all services |
| **{service}.yml** | Config specific to a service (e.g., user-service.yml) |
| **✅ What is user-service?**  **user-service is responsible for managing user-related data and actions in our insurance microservices system. It handles:**   * **User creation and retrieval** * **Storing user details (name, email, etc.)** * **Serving as the foundation for login/auth modules later** * **Interacting with other services (policy, claim, etc.)**   **🏗️ Layered Architecture We’ll Build**  **pgsql**  **CopyEdit**  **user-service**  **├── controller --> REST APIs**  **├── service --> Business logic**  **├── repository --> Data access layer**  **├── dto --> Data Transfer Objects**  **├── exception --> Global error handling**  **├── entity --> JPA entities**  **└── config --> Future use (for config/jwt/etc.)**  **🧱 Let's Build It Step-by-Step**  **We'll do this in steps:**   1. **📁 Project structure + base dependencies** 2. **🧾 Entity and DTO** 3. **⚙️ Repository** 4. **🧠 Service Layer** 5. **🌐 Controller** 6. **❗ Exception Handling (Global)** 7. **🔁 Mapper (DTO <-> Entity)** |  |

**Next Microservice: policy-service :**

**💡 Why build policy-service next?**

Because after users register via user-service, they should be able to **browse, view, and buy policies**. So we need a service to manage:

* List of insurance policies
* Policy details
* Assigning policies to users (later connected to user-service)

**🎯 Features of policy-service**

| **Feature** | **Description** |
| --- | --- |
| Get all policies | Return all available policies |
| Get policy by ID | Fetch details of a single policy |
| Add/Edit policy | Admin-only: Create or update policies |
| Assign policy to user | (Later: interaction with user-service) |

**📦 policy-service Structure**

We'll build policy-service just like user-service:

* ✅ Registered with Eureka
* ✅ Reads config from Config Server
* ✅ Uses DB (e.g., policydb)
* ✅ Proper structure (Controller, Service, DTO, Exception, ModelMapper)
* ✅ Input validation