**https://chatgpt.com/share/683ea4c3-0d30-800d-9eee-1633106532fe**

**✅ 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

**Next Microservice: Claim Service**

**🎯 Purpose:**

The claim-service handles insurance claims raised by users for specific policies.

**🧩 Responsibilities of claim-service:**

* Allow users to **raise a claim** on a policy.
* Allow fetching all claims for a user.
* Claim approval/denial logic (basic for now, extendable later).
* Communicate with:
  + **user-service** (to get user info)
  + **policy-service** (to get policy info)
* Register with **Eureka**
* Fetch config from **Config Server**
* Use ModelMapper, DTOs, validations, exception handling.

**📦 Key Features:**

| **Feature** | **Included** |
| --- | --- |
| Service Layer | ✅ |
| DTO with Validation | ✅ |
| Global Exception Handling | ✅ |
| ModelMapper | ✅ |
| JPA + MySQL | ✅ |
| Config from Config Server | ✅ |
| Eureka Client | ✅ |
| Feign Clients (later) | ⏳ (after claim-service basic setup) |

**🔧 Technologies Used:**

* Spring Boot
* Spring Data JPA
* MySQL
* Spring Cloud Config
* Eureka Discovery
* Lombok
* Bean Validation
* ModelMapper

**✅ Next Steps:**

1. **Create the claim-service project** with required dependencies
2. **Create config in Config Server** for claim-service
3. **Implement entity, DTO, repo, service, controller**
4. **Integrate Feign Clients** to talk to user-service and policy-service (if time allows)
5. **Test end-to-end**

**What is Feign Client?**

**Feign** is a **declarative web service client** provided by Spring Cloud. It allows you to **call other microservices** over HTTP **just like calling a Java method**.

✅ No need to write RestTemplate code  
✅ Automatically handles service discovery with **Eureka**  
✅ Simplifies inter-service communication

**🔧 What Feign Client is Doing in Your Project**

Let’s look at this example:

java

@FeignClient(name = "user-service")

public interface UserClient {

@GetMapping("/users/{userId}")

ResponseEntity<UserDto> getUserById(@PathVariable("userId") Long userId);

}

**🧠 What’s happening here?**

1. **@FeignClient(name = "user-service")**
   * Tells Spring: “I want to talk to a microservice called user-service”
   * user-service is the name it’s registered with in **Eureka Server**
2. **Method definition (getUserById)**
   * This tells Feign:  
     → When someone calls this method,  
     → Send an HTTP GET request to:  
     http://user-service/users/{userId}  
     (where user-service is resolved via Eureka)
3. **Return type**
   * Feign expects a response in the form of a UserDto JSON object
   * Automatically converts (deserializes) the response

**⚙️ How Feign Communicates Internally**

Here’s what happens step-by-step during communication:

1. 🚀 **You call userClient.getUserById(1)** in claim-service
2. 📡 **Feign Client** sends an HTTP GET request to user-service
3. 🧭 **Eureka Server** resolves user-service to its running instance (http://localhost:8081, for example)
4. 🌐 The request is routed to:  
   http://localhost:8081/users/1
5. 📥 Response JSON is received (e.g., { id: 1, name: "Sunil", email: "test@example.com" })
6. 🧱 Feign converts it into a UserDto object
7. ✅ You can now use it in your claim-service logic

**📦 Benefits of Feign**

| **Feature** | **Benefit** |
| --- | --- |
| Declarative | Write less code — no manual HTTP handling |
| Integrated with Eureka | Automatically finds service instances |
| Spring Boot compatible | Uses familiar annotations (@GetMapping) |
| Easy to test & maintain | Just mock the interface in unit tests |

**🔁 Example Communication Flow**

Client (claim-service)

|

| calls userClient.getUserById(1)

|

Feign client generates HTTP GET request

|

| Eureka resolves user-service → http://localhost:8081

|

|-----> http://localhost:8081/users/1

|

user-service returns JSON

|

Feign maps JSON → UserDto

|

Claim-service continues logic

**Objective**

We want to **secure our microservices** by enforcing **authentication and authorization at the API Gateway**, using **JWT (JSON Web Token)**.

**🔐 Why API Gateway Security?**

In a microservices app, the API Gateway is the **entry point** for all client requests. By adding security here:

| **Benefits** | **Explanation** |
| --- | --- |
| ✅ **Centralized Auth** | Token validation is done in one place. |
| ✅ **Simpler Services** | Microservices don’t need to worry about security. |
| ✅ **Scalability** | Easier to manage changes or upgrades to auth logic. |
| ✅ **Cleaner Logs & Audits** | You can log and monitor access at a single point. |

**🔄 Basic Flow**

Here’s the **step-by-step flow** for how this works:

Client

|

| 1. Sends login request to Auth Service --> receives JWT token

|

| 2. Sends request with Authorization header (Bearer <token>)

|

API Gateway

|

| 3. Validates token

| - If valid → forward to requested microservice

| - If invalid → reject with 401 Unauthorized

|

↓

Microservice

**🧱 Key Components in API Gateway Security**

| **Component** | **Role** |
| --- | --- |
| 🔐 spring-security | To intercept and secure incoming requests. |
| 🧪 JWT Validator | To check token signature and expiration. |
| 🚧 Filter | To intercept and authenticate every incoming request. |
| 🚪 SecurityConfig | To configure which paths are public/private. |

**✅ Implementation Plan**

Here’s how we will implement this in your **API Gateway**:

**Step 1: Add JWT and Spring Security Dependencies**

We’ll need:

* spring-boot-starter-security
* jjwt (or similar) for token parsing

**Step 2: Create JwtUtil to validate tokens**

This utility will:

* Decode the token
* Verify signature
* Check expiry

**Step 3: Add a JwtAuthenticationFilter**

This is a custom filter that:

* Runs before each request
* Extracts and validates the token
* If valid → adds user info to SecurityContext

**Step 4: Create a SecurityConfig class**

This config will:

* Allow open access to /auth/\*\* or login/register
* Protect all other routes
* Register the JWT filter

**🧪 Test Flow**

After implementation:

1. You get a token from the Auth Service.
2. You call any other API by attaching that token in the Authorization header.
3. API Gateway checks it.
4. Only if the token is valid → it lets the request pass.

**🎯 What You Secure at Gateway:**

| **Endpoint** | **Access** |
| --- | --- |
| /auth/\*\* | Public (login/register) |
| /policy/\*\* | Authenticated |
| /claim/\*\* | Authenticated |
| /user/\*\* | Authenticated |
| etc. | Based on role/permission (optional) |

**Add an auth-service — the dedicated Authentication & Authorization service**

**🔐 Here's What We'll Do:**

**1. Create auth-service**

* Expose /login and /register APIs.
* Authenticate using username/password.
* On success, generate **JWT token**.
* Use Spring Security, JWT, BCrypt for password encoding.

**2. Secure api-gateway**

* Intercept all incoming requests (except /auth/login, /auth/register).
* Validate the JWT token in the **Authorization header**.
* Reject if invalid or expired.
* Forward to microservices if valid.

**3. Secure all microservices**

* No direct access to user-service, claim-service, policy-service.
* Requests must come **through API Gateway** with valid JWT.

**🔧 Microservices Security Architecture (with JWT)**

Client

|

|---> POST /auth/login (get JWT)

|

|---> Request to any service via API Gateway

|

|---> API Gateway checks JWT

| |

| |---> If valid, forward to:

| - user-service

| - claim-service

| - policy-service

|

|---> If invalid, reject with 401 Unauthorized

**🧱 Services Overview**

| **Service** | **Responsibility** |
| --- | --- |
| **auth-service** | Login, Register, JWT token generation |
| **api-gateway** | Central gatekeeper, token validation |
| **user-service** | Business logic for users (secured) |
| **claim-service** | Business logic for claims (secured) |
| **policy-service** | Business logic for policies (secured) |

**🔐 Auth-Service Features**

**1. User Registration**

* API to register new users
* Save user with encrypted password
* Role assignment (e.g., USER, ADMIN)

**2. User Login**

* Validate credentials
* Generate JWT token
* Return token in response

**3. JWT Generation & Validation**

* Generate JWT with user details and roles
* Validate JWT in API Gateway or filters
* Token expiration handling

**4. Secure Password Storage**

* Use BCryptPasswordEncoder

**5. User Roles and Authorities**

* Store roles in DB
* Include roles in JWT claims

**6. Token Refresh (optional, advanced)**

**🧭 What is an API Gateway?**

An **API Gateway** is a single entry point for all client requests in a **microservices** architecture. Instead of calling each microservice directly, clients send requests to the **API Gateway**, and it routes them to the appropriate backend service.

**✅ API Gateway helps by:**

| **Feature** | **Description** |
| --- | --- |
| **Single Entry Point** | All client requests go through the gateway. |
| **Routing** | Forwards requests to appropriate microservice. |
| **Security** | Handles authentication & authorization (like validating JWT). |
| **Load Balancing** | Works with Eureka to route to available service instances. |
| **Rate Limiting** | Prevents abuse by limiting number of requests. |
| **Logging/Monitoring** | Easy to monitor all traffic through one place. |
| **CORS / Headers** | Manages headers, CORS, and request transformations. |

**⚙️ How It Works (Spring Cloud Gateway Example)**

1. **Client → API Gateway:**
   * Sends a request to http://gateway.com/policies/123
2. **Gateway → Routes:**
   * Matches route /policies/\*\* and forwards to policy-service
3. **Security Filter:**
   * Gateway has a JWT filter to validate the token.
   * If valid, request goes forward.
   * If invalid → 401 Unauthorized.
4. **Response Flow:**
   * Response from microservice comes back **through the gateway** to the client.

**Approach 1: Separate auth-service + API Gateway**

* **Auth-service** handles:
  + User registration
  + User login
  + JWT generation, validation logic
  + Password encoding, roles, authorities
  + Token refresh
* **API Gateway** handles:
  + Routing to microservices
  + Validating JWT (just validation, no generation)
  + Forwarding only valid requests

✅ **This is the recommended and scalable approach**