Spring Boot

Microservices

Configuration Management with Spring Cloud Config

Rohan Thapa
thaparohan2019@gmail.com

What is Configuration Management?

Configuration management is the process of handling **configuration files** in a way that allows:

- **Centralization:** Store and manage application configurations centrally.
- **Consistency:** Ensure that all microservices get the correct configuration consistently.
- **Versioning:** Version control configurations, allowing rollbacks and changes to be tracked easily.
- **Dynamic Updates:** Refresh configuration changes without restarting the service.
- **Security:** Secure sensitive data such as API keys and database credentials through encryption.

What is Spring Cloud Config?

Spring Cloud Config is a tool designed to provide **centralized external configuration** across distributed services in **microservices architecture**.

It uses a **Spring Cloud Config Server** that serves configuration properties to different microservices, while the microservices act as Config Clients, fetching their properties from the Config Server.

Key Components:

- **Config Server:** Acts as the centralized configuration management service. It fetches configuration data from various sources like **Git**, **local file system**, or **database** and serves them to the clients.
- **Config Client:** The microservices or applications that fetch their configuration properties from the Config Server.

Why Use Spring Cloud Config?

- **Centralized Management:** Manage all configuration for multiple environments (**development, testing, production**) in a single place.
- **Separation of Concerns:** Application code is separated from configuration, enabling easy management of different environments.
- **Version Control:** Store configurations in a Git repository (or similar) allowing version control and easy rollback.
- **Dynamic Refresh:** Ability to refresh the **configuration properties** at runtime **without restarting** the microservices.

How Spring Cloud Config Works:

1. The **Spring Cloud Config** Server **stores** configuration properties for microservices in **external sources** such as **Git**, **Vault**, or **databases**.
2. The **Config Clients** (**microservices**) request their configurations from the Config Server at **startup**, specifying their application name and profile (e.g., **dev, test, prod**).
3. Optionally, with **Spring Cloud Bus** or **Actuator**, configuration can be refreshed dynamically across multiple microservices without needing to restart them.

Setting Up Spring Cloud Config

1. Create a Spring Cloud Config Server

In this example, we will create a **Spring Cloud Config** Server that fetches configurations from a Git repository.

1. Add Dependencies to **pom.xml**:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

Setting Up Spring Cloud Config

2.**Enable Config Server** by adding **@EnableConfigServer** in your main application class:

```
import ...

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

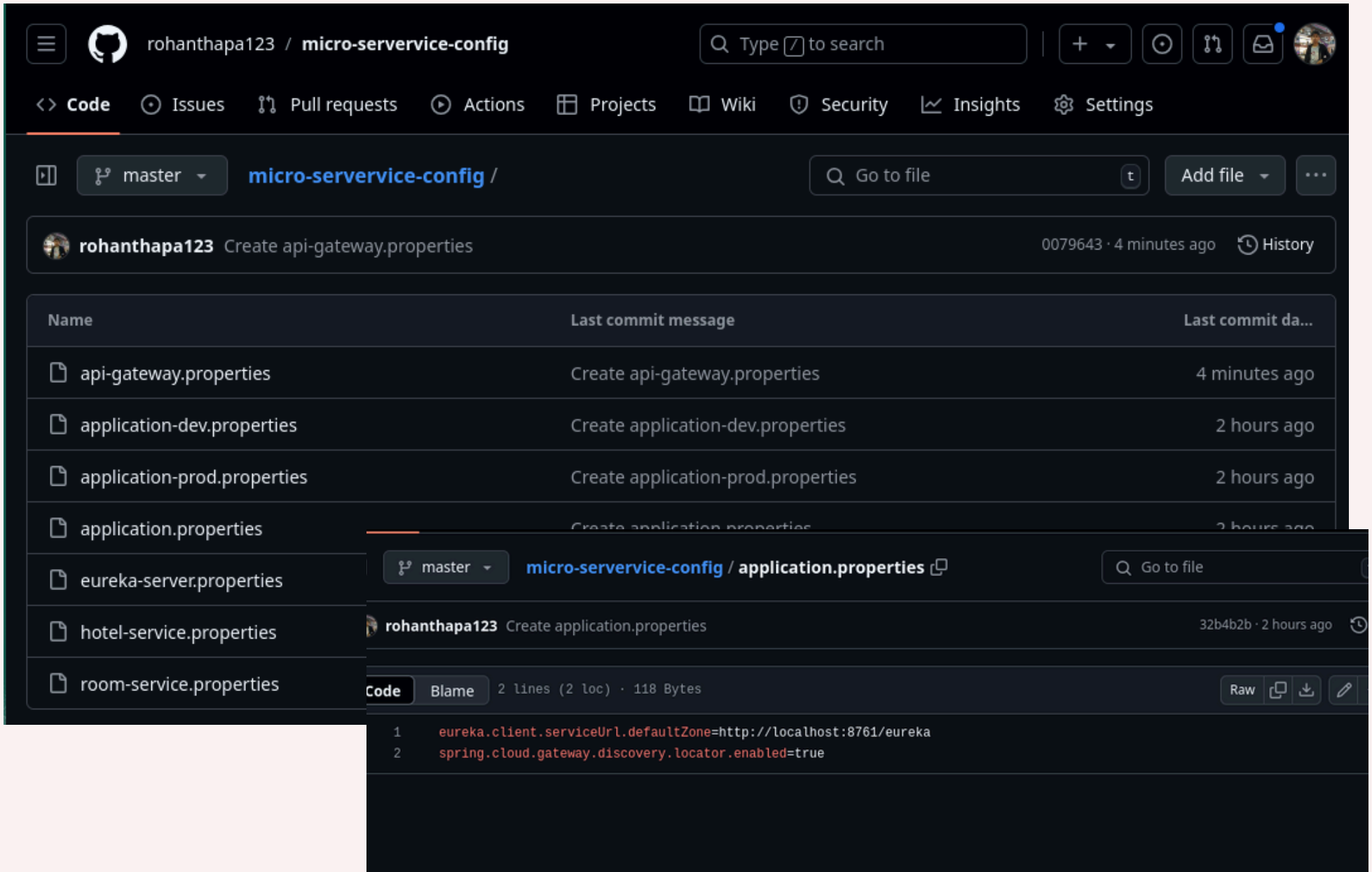
Setting Up Spring Cloud Config

3. Configure the Config Server in **application.properties** or **application.yml**:

```
spring.application.name=config-server

server.port=8188

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.config.server.git.uri=https://github.com/rohanthapa123/micro-servervice-config
spring.cloud.config.server.git.clone-on-start=true

management.endpoints.web.exposure.include=*
```

Create Configurations in Git Repository

In the **Git repository**, create configuration files for each microservice in the format **application-{profile}.properties** (e.g., **application-dev.properties**, **microservice1-prod.properties**).



Set Up Spring Cloud Config Client

Add dependencies to the client's **pom.xml**:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

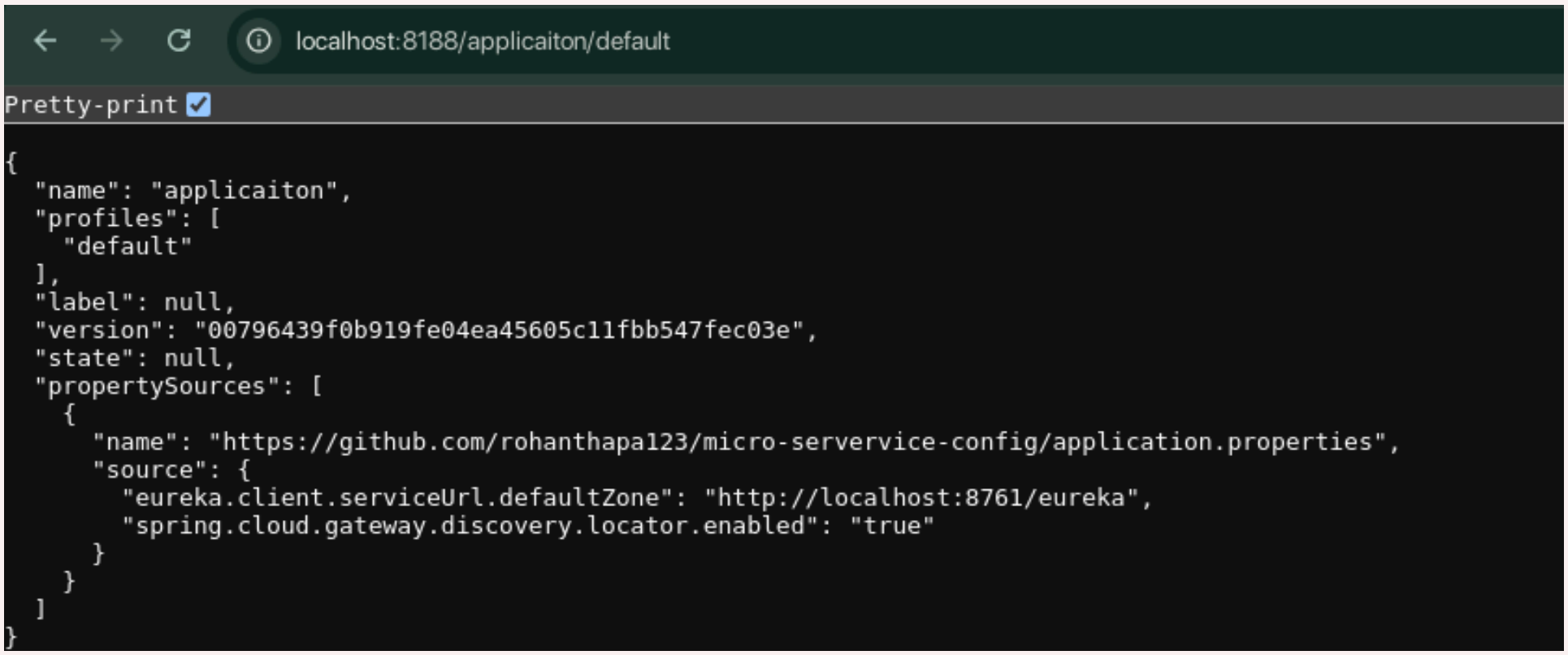
Set Up Spring Cloud Config Client

Configure the client to fetch configurations from the server by adding **application.properties** or **application.yml** to the client application:

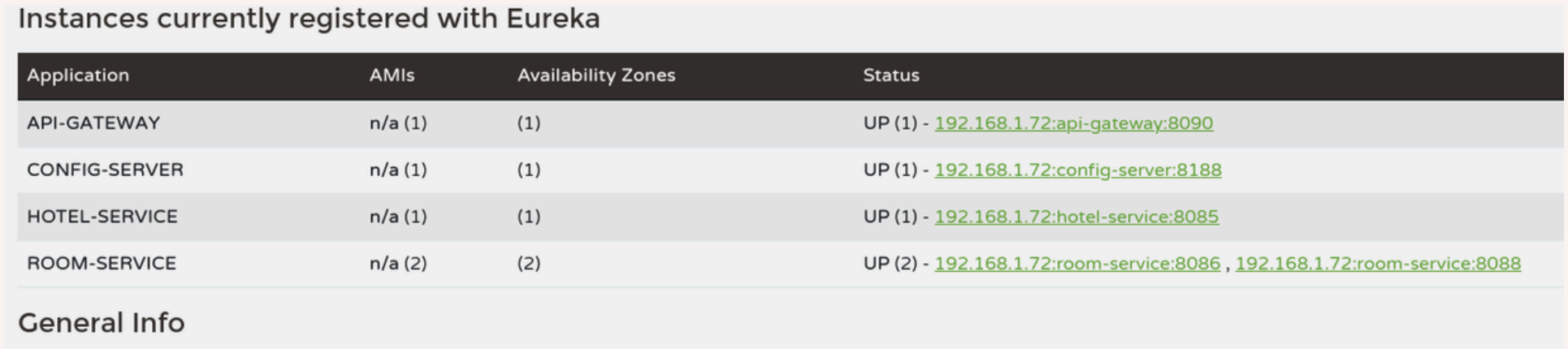
```
spring.application.name=api-gateway

server.port=8090

#fromconfigserver
spring.config.import = configserver:http://localhost:8188
```

Output

Default **application.properties**

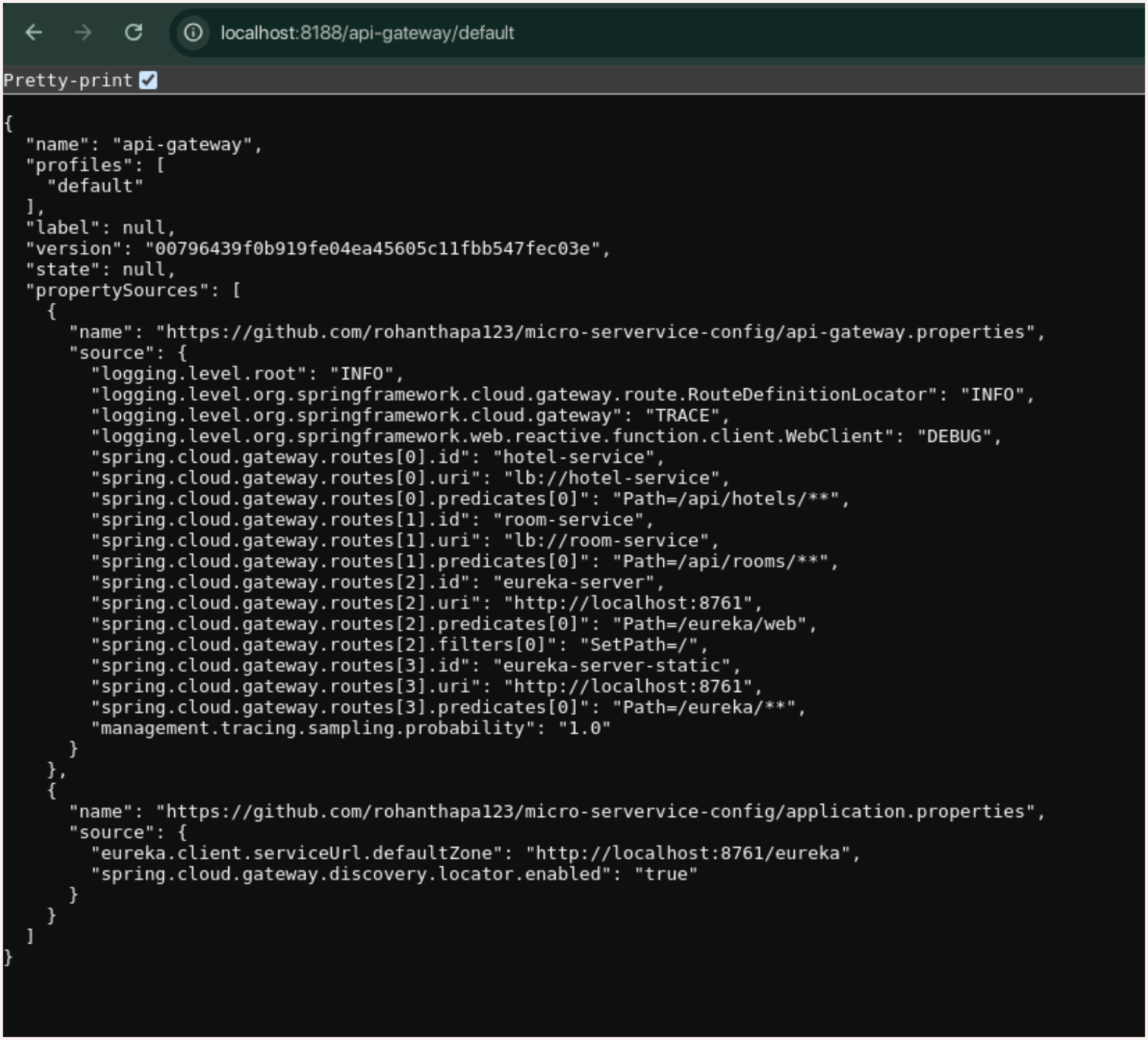


And all microservices shows in eureka



Output

Default **api-gateway.properties**



Enable Auto Refresh with Spring Cloud Bus

To automatically refresh configuration in microservices, **Spring Cloud Bus** uses a message broker (e.g., **RabbitMQ, Kafka**). This ensures that once a configuration change is made, it propagates to all services without requiring manual refreshes.

Configuration Properties

spring cloud config server

Property	Description
`spring.cloud.config.server.git.uri`	URI of the Git repository that stores configuration files (e.g., `https://github.com/repo/config.git`).
`spring.cloud.config.server.git.username`	Username for accessing the Git repository (if authentication is required).
`spring.cloud.config.server.git.password`	Password for accessing the Git repository (if authentication is required).
`spring.cloud.config.server.git.clone-on-start`	Clones the repository when the config server starts. Defaults to `false`.
`spring.cloud.config.server.git.searchPaths`	Directory inside the Git repository where configuration files are stored.
`spring.cloud.config.server.native.search-locations`	Specifies a native (file-based) source location for configuration files.
`spring.cloud.config.server.git.default-label`	Default branch or label to use when no specific label is requested. Defaults to `master` or `main`.
`spring.cloud.config.server.composite[0].type`	Use `git`, `native`, or `vault` for composite backends.
`spring.cloud.config.server.encrypt.enabled`	Enable encryption and decryption for sensitive properties. Defaults to `true`.
`spring.cloud.config.server.health.enabled`	Enables the `/health` endpoint to check the status of the config server. Defaults to `true`.
`spring.cloud.config.server.bootstrap`	Enables bootstrap configuration for the config server itself. Defaults to `false`.

Configuration Properties

spring cloud config client

Property	Description
`spring.config.import`	Specifies the URL of the Config Server (e.g., `configserver:http://localhost:8888`).
`spring.cloud.config.name`	The name of the application to search for in the configuration repository (e.g., `api-gateway`).
`spring.cloud.config.profile`	Specifies the profile to use for configuration (e.g., `dev`, `prod`).
`spring.cloud.config.label`	The Git branch or label to use when pulling configuration (e.g., `main`, `development`).
`spring.cloud.config.fail-fast`	Causes the application to fail startup if it cannot connect to the Config Server. Defaults to `false`.
`spring.cloud.config.token`	Authentication token to access private configuration repositories.
`spring.cloud.config.username`	Username for accessing the Config Server (if secured).
`spring.cloud.config.password`	Password for accessing the Config Server (if secured).
`spring.cloud.config.retry.enabled`	Enables retry logic for connecting to the Config Server (default is `false`).
`spring.cloud.config.discovery.enabled`	Enables the Config Client to discover the Config Server via Eureka (defaults to `false`).

Important Concepts in Spring Cloud Config

- **Profiles:** Separate configurations based on environments (dev, test, prod). Spring uses the application-{profile}.properties format.

- **Encryption/Decryption:** Spring Cloud Config supports encryption and decryption of sensitive properties using the spring-cloud-starter-config and spring-security-rsa dependencies.

- **Fallbacks:** If the Config Server is down or unreachable, Config Clients can use default configurations defined locally.

Conclusion

Spring Cloud Config simplifies configuration management for microservices by centralizing and externalizing configuration, versioning it with **Git**, and providing **dynamic refresh features**.

It enables distributed systems to scale while maintaining consistency across all services. By integrating with Spring Cloud Bus, you can trigger real-time configuration changes across the entire system without redeploying services.

Thank You

Rohan Thapa
thaparohan2019@gmail.com