Spring Cloud Netflix Eureka: Step-by-Step Tutorial

author: Ramesh Fadatare

SPRING BOOT SPRING CLOUD

In a microservices architecture, dynamic service discovery is crucial to enable services to find and communicate with each other. Spring Cloud Netflix Eureka provides a robust service registry to manage this discovery. This tutorial demonstrates how to set up a Eureka server and register two microservices (`ProductService` and `OrderService`). We will also show how one service can call another service using service discovery.

Prerequisites

- JDK 17 or higher
- Maven or Gradle
- IDE (e.g., IntelliJ IDEA, Eclipse)
- Basic knowledge of Spring Boot

What is Spring Cloud Netflix Eureka?

Spring Cloud Netflix Eureka is a service registry that allows microservices to register themselves at runtime and discover other services. It acts as a lookup service where microservices can find each other to communicate without hardcoding the hostnames and ports.

Key Components of Eureka

Eureka Server: Acts as a service registry where microservices register themselves and discover other services.

Eureka Client: Microservices that register themselves with the Eureka Server and fetch the registry information to locate other services.

How Eureka Works

1. Service Registration

When a microservice starts, it registers itself with the Eureka Server. The registration process involves sending the service metadata, such as the hostname, IP address, port, and health check URL, to the Eureka Server.

2. Service Discovery

Eureka Clients periodically fetch the registry information from the Eureka Server to get the details of other registered services. This allows microservices to locate and communicate with each other dynamically.

3. Health Checks

Eureka performs periodic health checks on the registered services to ensure they are available. If a service fails the health check, it is marked as unavailable, and Eureka stops routing requests to it.

4. Heartbeats

Eureka Clients send periodic heartbeats to the Eureka Server to indicate that they are still alive. If a client fails to send heartbeats within a certain period, it is considered down and removed from the registry.

Step-by-Step Tutorial

Step 1: Create a Eureka Server

1.1: Using Spring Initializr to Create the Project

1. Go to **Spring Initializr**.
2. Set the project metadata:
   - **Group**: `com.example`
   - **Artifact**: `eureka-server`
   - **Name**: `Eureka Server`
   - **Description**: `Eureka Server for Service Discovery`
   - **Package Name**: `com.example.eurekaserver`
   - **Packaging**: `Jar`
   - **Java**: `17` (or higher)
3. Add dependencies:
   - **Eureka Server**
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

1.2: Configure the Eureka Server

Add the following configuration to `src/main/resources/application.yml`:

```yaml
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
  server:
    enable-self-preservation: false
```

1.3: Enable Eureka Server

Add the `@EnableEurekaServer` annotation to the main application class:

```java
package com.example.eurekaserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

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

1.4: Run the Eureka Server

Run the application and verify that the Eureka Server is running by accessing `http://localhost:8761` in your browser. You should see the Eureka dashboard.

Step 2: Create `ProductService` Microservice

2.1: Using Spring Initializr to Create the Project

1. Go to **Spring Initializr**.
2. Set the project metadata:
    - **Group**: `com.example`
    - **Artifact**: `product-service`
    - **Name**: `Product Service`
    - **Description**: `E-commerce Product Service`
    - **Package Name**: `com.example.productservice`
    - **Packaging**: `Jar`
    - **Java**: `17` (or higher)
3. Add dependencies:
    - **Spring Web**
    - **Eureka Discovery Client**
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

2.2: Configure `ProductService`

Add the following configuration to `src/main/resources/application.yml`:

```yaml
server:
  port: 8081

spring:
  application:
    name: product-service
```

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

2.3: Enable Eureka Client

Add the `@EnableEurekaClient` annotation to the main application class:

```java
package com.example.productservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

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

2.4: Define the Product Model

Create a new class `Product` in `src/main/java/com/example/productservice`:

```java
package com.example.productservice;

public class Product {
    private Long id;
    private String name;
    private double price;

    // Constructors, getters, and setters
    public Product() {}

    public Product(Long id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
}
```

2.5: Create a ProductController

Create a new class `ProductController` in `src/main/java/com/example/productservice`:

```
package com.example.productservice;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ProductController {

    @GetMapping("/products/{id}")
    public Product getProductById(@PathVariable Long id) {
        // For simplicity, returning a hardcoded product. In a real application, you'd query the database.
        return new Product(id, "Sample Product", 99.99);
    }
}
```

2.6: Run `ProductService`

Run the application. Verify that it registers with Eureka Server by accessing the Eureka dashboard at `http://localhost:8761`. You should see `product-service` listed under "Instances currently registered with Eureka".

Step 3: Create `OrderService` Microservice

3.1: Using Spring Initializr to Create the Project

1. Go to **Spring Initializr**.
2. Set the project metadata:
    - **Group**: `com.example`

- **Artifact**: `order-service`
- **Name**: `Order Service`
- **Description**: `E-commerce Order Service`
- **Package Name**: `com.example.orderservice`
- **Packaging**: `Jar`
- **Java**: `17` (or higher)

3. Add dependencies:
   - **Spring Web**
   - **Eureka Discovery Client**
   - **OpenFeign**
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

3.2: Configure `OrderService`

Add the following configuration to `src/main/resources/application.yml`:

```yaml
server:
  port: 8082

spring:
  application:
    name: order-service

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

3.3: Enable Eureka Client

Add the `@EnableEurekaClient` and `@EnableFeignClients` annotations to the main application class:

```java
package com.example.orderservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.openfeign.EnableFeignClients;

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

3.4: Define the Order Model

Create a new class `Order` in `src/main/java/com/example/orderservice`:

```
package com.example.orderservice;

public class Order {
    private Long id;
    private String product;
    private int quantity;

    // Constructors, getters, and setters
    public Order() {}

    public Order(Long id, String product, int quantity) {
        this.id = id;
        this.product = product;
        this.quantity = quantity;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getProduct() {
        return product;
    }

    public void setProduct(String product) {
        this.product = product;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

3.5: Create a Feign Client to Communicate with `ProductService`

Create a new interface `ProductClient` in `src/main/java/com/example/orderservice`:

```
package com.example.orderservice;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.PathVariable;

@FeignClient(name = "product-service")
public interface ProductClient {

    @GetMapping("/products/{id}")
    Product getProductById(@PathVariable("id") Long id

);
}
```

3.6: Create an OrderController

Create a new class `OrderController` in `src/main/java/com/example/orderservice`:

```
package com.example.orderservice;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    @Autowired
    private ProductClient productClient;

    @GetMapping("/orders/{id}")
    public Order getOrderById(@PathVariable Long id) {
        // For simplicity, returning a hardcoded order with a product fetched from ProductService.
        Product product = productClient.getProductById(id);
        return new Order(id, product.getName(), 2);
    }
}
```

3.7: Run `OrderService`

Run the application. Verify that it registers with Eureka Server by accessing the Eureka dashboard at `http://localhost:8761`. You should see `order-service` listed under "Instances currently registered with Eureka".

Testing the Setup

Test with Postman

1. **Start All Services**: Ensure that `EurekaServer`, `ProductService`, and `OrderService` are running.

Test Product Service

- **URL**: `http://localhost:8081/products/1`

- **Method**: GET
- **Expected Response**:

```
{
  "id": 1,
  "name": "Sample Product",
  "price": 99.99
}
```

Test Order Service

- **URL**: `http://localhost:8082/orders/1`
- **Method**: GET
- **Expected Response**:

```
{
  "id": 1,
  "product": "Sample Product",
  "quantity": 2
}
```

Conclusion

In this tutorial, we've created a complete end-to-end setup using Spring Cloud Netflix Eureka. We've set up a Eureka server and two microservices (`ProductService` and `OrderService`) that register with the Eureka server. We also demonstrated how one service (`OrderService`) can call another service (`ProductService`) using Spring Cloud OpenFeign for service discovery.

This setup allows for dynamic service discovery, enabling microservices to locate and communicate with each other without hardcoding hostnames and ports. Spring Cloud Netflix Eureka simplifies the management of microservices by providing a robust and reliable service discovery mechanism. By following this tutorial, you've gained hands-on experience in setting up and using Eureka for service discovery in a microservices architecture.