Service Registry/ Service Discovery/ Load Balancing

Challenges:

1. Endpoint keep on changing as per requirement[services discovery]
2. Multiple instances are available as per load [Load balancing]
3. How new services enter to micro service environment[service registration]

What is a Registry Service in Microservices?

In a microservices architecture, a **registry service** (or service registry) is responsible for keeping track of all services and their instances running in the system. The registry allows microservices to discover other services without needing hard-coded URLs. This is particularly useful in dynamic environments where service instances can be scaled up or down or might change their network locations (such as in a containerized environment).

In Spring Boot, the most commonly used service registry is **Eureka**, which is part of the Netflix OSS ecosystem.

Key Concepts in a Registry Service:

1. **Service Registration**: When a service starts up, it registers itself with the registry.
2. **Service Discovery**: When a service needs to communicate with another service, it queries the registry to get the network location (IP and port) of the required service.

Client Side Service Discovery

In client-side service discovery, the client is responsible for determining the location of a service instance. It queries a service registry to get a list of available service instances and then chooses one to send the request to, usually using a load-balancing algorithm.

Components:

1. **Service Registry**: A database that stores the available service instances and their locations (e.g., Eureka, Consul, etc.).
2. **Client**: The service making a request to another service. It is responsible for querying the service registry to get the service instance details.
3. **Load Balancer**: Runs on the client-side and determines which service instance to use from the list provided by the service registry.

Workflow:

1. The client sends a request to the service registry.
2. The service registry returns a list of available instances of the requested service.
3. The client chooses one instance using a load-balancing algorithm (e.g., round-robin, least connections, etc.).
4. The client sends the request directly to the chosen service instance.

Server Side Service Discovery

In server-side service discovery, the client does not need to be aware of the service registry or load balancing. Instead, it sends a request to a "discovery server" (often a load balancer or API Gateway) that queries the service registry, selects an available instance, and forwards the request to that instance.

Components:

1. **Service Registry**: A database that stores the available service instances (same as in client-side discovery).
2. **Client**: Sends a request to the discovery server without needing to know the location of the service instances.
3. **Discovery Server**: A load balancer or API Gateway that queries the service registry, selects a service instance, and forwards the request to the service.

Workflow:

1. The client sends a request to the discovery server (e.g., API Gateway).
2. The discovery server queries the service registry for available instances of the requested service.

3. The discovery server selects one service instance using a load-balancing algorithm.
4. The discovery server forwards the request to the selected service instance.

Implementing a Registry Service Using Eureka in Spring Boot

Step 1: Add Dependencies

First, you need to include the Eureka Server and Eureka Client dependencies in your `pom.xml`.

For **Eureka Server**:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

For **Eureka Client** (for other microservices that need to register with Eureka):

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

You also need to add the Spring Cloud BOM for compatibility:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>2021.0.3</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Step 2: Setting Up Eureka Server

1. Create a Spring Boot application and add the `@EnableEurekaServer` annotation in your main class to make it a Eureka server.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServe
r;

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

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

1. Configure the `application.yml` or `application.properties` file for the Eureka server.

`application.yml`:

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

```
spring:
  application:
    name: eureka-server
```

- `port: 8761`: By default, Eureka runs on port `8761`.
- `register-with-eureka: false` and `fetch-registry: false`: These settings tell Eureka not to register itself as a client or fetch the registry (since it's the registry itself).

Now, run the application, and you can access the Eureka dashboard at `http://localhost:8761`.

Step 3: Setting Up Eureka Clients (Other Microservices)

Each microservice that you want to register with Eureka needs to include the Eureka client dependency and configure the Eureka client settings.

1. Create a Spring Boot microservice and annotate the main application class with `@EnableEurekaClient`.

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

@SpringBootApplication
@EnableEurekaClient
public class MyMicroserviceApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyMicroserviceApplication.class, args);
    }
}
```

1. Configure the `application.yml` or `application.properties` file for the client.

`application.yml`:

```
spring:
  application:
    name: my-microservice

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/  # Eureka Server URL

server:
  port: 8081

info:
    app:
        name: 'service'
        description: 'service description',
        version: '1.0'

management:
    endpoints:
        web:
            exposure:
                include: "*"
    endpoint:
        shutdown:
            enabled: true
    info:
        env:
            enabled: true
```

- `spring.application.name`: This is the name of your microservice. Eureka will register the service with this name.
- `eureka.client.service-url.defaultZone`: This is the URL of the Eureka server where the service will register itself.

1. Run the client microservice. You should see it registered in the Eureka dashboard at `http://localhost:8761`.

Step 4: Service Discovery

Once the services are registered in Eureka, you can use **service discovery** to communicate between microservices without hardcoding URLs. Here's how you can use the Spring Boot `RestTemplate` with Eureka for service discovery:

1. Enable service discovery by adding the `@LoadBalanced` annotation to the `RestTemplate` bean.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;

@Configuration
public class AppConfig {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

1. Use the service name instead of the actual URL when making requests:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class MyController {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/consume")
    public String consumeService() {
        String response = restTemplate.getForObject("http://my-microservice/
endpoint", String.class);
        return response;
    }
}
```

In the example above, `http://my-microservice/endpoint` will be resolved to the actual host and port of the service registered with the name `my-microservice`.

Conclusion

- **Eureka Server** is used to maintain a registry of services.
- **Eureka Clients** register themselves with the Eureka server so they can be discovered.
- Other services use service discovery to locate the instances of other services, enabling loose coupling and scalability.

You can further enhance your setup by adding **load balancing**, **circuit breaking**, and other Spring Cloud Netflix features.

In a Spring Boot microservices architecture, you can use **Feign Client** and **WebClient** with **client-side load balancing** to call other services. Spring Cloud provides built-in support for load balancing using **Spring Cloud LoadBalancer** (which replaced Netflix Ribbon).