Spring Cloud LoadBalancer Example - Client-Side Load Balancing

author: Ramesh Fadatare

SPRING BOOT SPRING CLOUD

In a microservices architecture, services often need to communicate with each other. While having multiple instances of a service running improves scalability and reliability, effectively distributing the traffic among these instances can be challenging. Client-side load balancing addresses this challenge by balancing the load across service instances directly from the client side.

Spring Cloud LoadBalancer is a modern and lightweight alternative to Netflix Ribbon for client-side load balancing in Spring Boot applications. It integrates seamlessly with Spring Cloud and Spring Boot applications, making it easy to implement and configure.

What is Client-Side Load Balancing?

Client-side load balancing involves distributing network traffic among multiple servers from the client's side. Unlike server-side load balancing, where the load balancer sits between clients and servers, client-side load balancing places the responsibility of distributing requests across available servers on the client.

Key Benefits

1. **Improved Performance**: Reduces latency by distributing the load directly from the client side.
2. **Fault Tolerance**: Enhances reliability by routing requests away from failed or slow service instances.
3. **Scalability**: Easily scales by adding more service instances without changing the client configuration.

We will create two microservices: **ProductService** and **OrderService**. **OrderService** will call **ProductService** using **Spring Cloud OpenFeign** and `Spring Cloud LoadBalancer` for client-side load balancing.

Create ProductService

Step 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**: `11` (or higher)
3. Add dependencies:
   - **Spring Web**
4. Click on **Generate** to download the project.

5. Unzip the downloaded project and open it in your IDE.

Step 2: Define the Product Model

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

```
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;
    }
}
```

Step 3: 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;
```

```java
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);
    }
}
```

Step 4: Run `ProductService`

Ensure the main application class is set up correctly:

```java
package com.example.productservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Create OrderService

Step 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**: `11` (or higher)
3. Add dependencies:
    - **Spring Web**
    - **Spring Cloud OpenFeign**
    - **Spring Cloud LoadBalancer**
4. Click on **Generate** to download the project.
5. Unzip the downloaded project and open it in your IDE.

Step 2: Enable Feign Clients

Open the `OrderService` project and add the `@EnableFeignClients` annotation to the main application class. This enables Feign client support in the Spring Boot application.

```
package com.example.orderservice;

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

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

Step 3: Define a Feign Client Interface

Create a Feign client interface to communicate with `ProductService`.

Let's create a new package `com.example.orderservice.clients` and a new interface `ProductClient` in `src/main/java/com/example/orderservice/clients`:

```
package com.example.orderservice.clients;

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);
}

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;
    }
}
```

Step 4: Use the Feign Client

Create a service that uses the Feign client to fetch product information from `ProductService`.

Let's create a new class `OrderService` in `src/main/java/com/example/orderservice`:

```
package com.example.orderservice;

import com.example.orderservice.clients.Product;
import com.example.orderservice.clients.ProductClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class OrderService {

    @Autowired
    private ProductClient productClient;

    public Product getProductById(Long id) {
        return productClient.getProductById(id);
    }
}
```

Step 5: Create a REST Controller

Create a REST controller that exposes an endpoint to get product information using the `OrderService`.

Let's create a new class `OrderController` in `src/main/java/com/example/orderservice`:

```
package com.example.orderservice;

import com.example.orderservice.clients.Product;
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 OrderService orderService;

    @GetMapping("/orders/{id}/product")
    public Product getProductById(@PathVariable Long id) {
        return orderService.getProductById(id);
    }
}
```

Configure Client-Side Load Balancing

To enable client-side load balancing, we need to configure Spring Cloud LoadBalancer.

Add Load Balancer Configuration:

Add the `spring-cloud-starter-loadbalancer` dependency to `OrderService`'s `pom.xml`:

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

Ensure that `OrderService` is aware of multiple instances of `ProductService`. For simplicity, we can define multiple instances in the `application.yml` file of `OrderService`:

```
spring:
  cloud:
    loadbalancer:
      instances:
        product-service:
          - uri: http://localhost:8081
          - uri: http://localhost:8082
```

Run the Applications

1. **Run Multiple Instances of `ProductService`:**
    - Ensure `ProductService` runs on different ports (e.g., `8081` and `8082`).

- You can achieve this by copying the `ProductService` project and running the instances on different ports or configuring the application to run on different ports.

2. **Run `OrderService`:**

   - Ensure `OrderService` runs on a different port (default port `8080`).
   - Verify it is running by accessing `http://localhost:8080/orders/1/product` in your browser. It should return the product details fetched from `ProductService`.

Conclusion

Spring Cloud LoadBalancer simplifies client-side load balancing in a microservices architecture. By leveraging declarative REST client definitions with Spring Cloud OpenFeign and configuring Spring Cloud LoadBalancer, you can easily distribute traffic among multiple service instances.

By following the steps in this blog post, you've set up two simple e-commerce microservices and demonstrated how to use Spring Cloud OpenFeign with Spring Cloud LoadBalancer for client-side load balancing. Embrace the power of Spring Cloud LoadBalancer and simplify your microservices architecture today!