

# Spring Tips: Spring Cloud Loadbalancer

speaker: [Josh Long \(@starbuxman\)](#)

Spring Tips: Spring Cloud Loadbalancer #SpringFramework...



Hi, Spring fans! Welcome to another installment of Spring Tips! In this installment, we're going to look at a new feature in Spring Cloud, Spring Cloud Loadbalancer. Spring Cloud Loadbalancer is a generic abstraction that can do the work that we used to do with Netflix's Ribbon project. Spring Cloud still supports Netflix Ribbon, but Netflix Ribbons days are numbered, like so much else of the Netflix microservices stack, so we've provided an abstraction to support an alternative.

## The Service Registry

For us to use the Spring Cloud Load Balancer, we need to have a service registry up and running. A service registry makes it trivial to programmatically query for the location of a given service in a system. There are several popular implementations, including Apache Zookeeper, Netflix's Eureka, Hashicorp Consul, and others. You can even use Kubernetes and Cloud Foundry as service registries. Spring Cloud provides an abstraction,

`DiscoveryClient`, that you can use to talk to these service registries generically. There are several patterns that a service registry enables that just aren't possible with *good 'ol* DNS. One thing I love to do is client-side load-balancing. Client-side load-balancing requires the client code to decide which node receives the request. There is any number of instances of the service out there, and their suitability to handle a particular request is something each client can decide. It's even better if it can make the decision *before* launching a request that might otherwise be doomed to failure. It saves time, unburdens the services with tedious flow control requirements, and makes our system more dynamic since we can *query* its topology.

You can run any service registry you like. I like to use Netflix Eureka for these sorts of things because it is simpler to setup. Let's set up a new instance. You could download and run a stock-standard image if you want, but I want to use the pre-configured instance provided as part of Spring Cloud.

Go to the Spring Initializer, choose `Eureka Server` and `Lombok`. I named mine `eureka-service`. Hit `Generate`.

Most of the work of using the built-in Eureka Service is in the configuration, which I've reprinted here.

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

COPY

Then you'll need to customize the Java class. Add the `@EnableEurekaServer` annotation to your class.

```
package com.example.eurekaservice;

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

@SpringBootApplication
@EnableEurekaServer
public class EurekaServiceApplication {

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

}
```

COPY

You can run that now. It'll be available on port `8761` and other clients will connect to that port by default.

## A Simple API

Let's now turn to the API. Our API is as trivial as these things come. We just want an endpoint to which our client can issue requests.

Go to the Spring Initializr, generate a new project with `Reactive Web` and `Lombok` and the `Eureka Discovery Client`. That last bit is the critical part! You're not going to see it used in the following Java code. It's [all autoconfiguration, which we also covered way back in 2016](#), that runs at application startup. The autoconfiguration will automatically register the application with the specified registry (in this case, we're using the `DiscoveryClient` implementation for Netflix's Eureka) using the `spring.application.name` property.

Specify the following properties.

```
spring.application.name=api
server.port=9000
```

COPY

Our HTTP endpoint is a "Hello, world!" handler that uses the functional reactive HTTP style that we [introduced in another Spring Tips video way, way back in 2017](#).

```
package com.example.api;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.function.server.HandlerFunction;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;

import java.util.Map;

import static org.springframework.web.reactive.function.server.RouterFunctions.route;
import static org.springframework.web.reactive.function.server.ServerResponse.*;

@SpringBootApplication
public class ApiApplication {

    @Bean
    RouterFunction<ServerResponse> routes() {
        return route()
```

COPY

```

        .GET("/greetings", r -> ok().bodyValue(Map.of("greetings",
"Hello, world!")))
        .build();
    }

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

```

Run the application, and you'll see it reflected in the Netflix Eureka instance. You can change the `server.port` value to `0` in `application.properties`. If you run multiple instances, you'll see them reflected in the console.

## The Load-Balancing Client

All right, now we're ready to demonstrate load balancing in action. We'll need a new spring boot application. Go to the Spring Initializr and generate a new project using the `Eureka Discovery Client`, `Lombok`, `Cloud Loadbalancer`, and `Reactive Web`. Click `Generate` and open the project in your favorite IDE.

Add the Caffeine Cache to the classpath. It's not on the Spring Initializr, so I added it manually. It's Maven coordinates are

`com.github.ben-manes.caffeine:caffeine:${caffeine.version}`. If this dependency is present, then the load balancer will use it to cache resolved instances.

Let's review what we want to happen. We want to make a call to our service, `api`. We know that there could be more than one instance of the service in the load balancer. We *could* put the API behind a load balancer and just call it done. But what we want to do is to use the information available to us about the state of each application to make smarter load balancing decisions. There are a lot of reasons we might use the client-side load balancer instead of DNS. First, Java DNS clients tend to cache the resolved IP information, which means that subsequent calls to the same resolved IP would end up subsequently dogpiling on top of one service. You can disable that, but you're working against the grain of DNS, a caching-centric system. DNS only tells you *where* something is, not *if* it is. Put another way; you don't know if there is going to be anything waiting for your request on the other side of that DNS based load balancer. Wouldn't you like to be able to know before making the call, sparing your client the tedious timeout period before the call fails? Additionally, some patterns like service hedging - [also the topic of another Spring Tips video](#) - is only possible with a service registry.

Let's look at the usual configuration properties for the `client`. The properties specify the `spring.application.name`, nothing novel about that. The second property is important. It disables the default Netflix Ribbon-backed load balancing strategy that's been in place since Spring Cloud debuted in 2015. We want to use the new Spring Cloud Load balancer, after all.

```
spring.application.name=client
spring.cloud.loadbalancer.ribbon.enabled=false
```

COPY

So, let's look at the use of our service registry. First thing's first, our client needs to establish a connection to the service registry with the Eureka `DiscoveryClient` implementation. The Spring Cloud `DiscoveryClient` abstraction is on the classpath, so it'll automatically start-up and register the `client` with the service registry.

Here are the beginnings of our application, an entry point class.

```
package com.example.client;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.extern.log4j.Log4j2;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.client.loadbalancer.reactive.ReactiveLoadBalancer;
import org.springframework.cloud.client.loadbalancer.reactive.ReactorLoadBalancerExchangeFilterFunction;
import org.springframework.cloud.client.loadbalancer.reactive.Response;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Flux;

import static com.example.client.ClientApplication.call;

@SpringBootApplication
public class ClientApplication {

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

COPY

We'll add to this a DTO class to convey the JSON structure returned from the service to the clients. This class uses some of Lombok's convenient annotations.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
class Greeting {
    private String greetings;
}
```

COPY

Now, let's look at three different approaches to load balancing, each progressively more sophisticated.

## Using the Loadbalancer Abstraction Directly

This first approach is the simplest, albeit most verbose, of the three. In this approach, we'll work with the load balancing abstraction directly. The component injects a pointer to the `ReactiveLoadBalancer.Factory<ServiceInstance>`, which we can then use to vend a `ReactiveLoadBalancer<ServiceInstance>`. This `ReactiveLoadBalancer` is the interface with which we load-balance calls to the `api` service by invoking `api.choose()`. I then use that `ServiceInstance` to build up a URL to the particular host and port of that specific `ServiceInstance` and then make an HTTP request with `WebClient`, our reactive HTTP client.

```
@Log4j2
@Component
class ReactiveLoadBalancerFactoryRunner {

    ReactiveLoadBalancerFactoryRunner(ReactiveLoadBalancer.Factory<ServiceInstance> serviceInstanceFactory) {
        var http = WebClient.builder().build();
        ReactiveLoadBalancer<ServiceInstance> api =
            serviceInstanceFactory.getInstance("api");
        Flux<Response<ServiceInstance>> chosen = Flux.from(api.choose());
        chosen
            .map(responseServiceInstance -> {
                ServiceInstance server = responseServiceInstance.getServer();
                var url = "http://" + server.getHost() + ':' +
                    server.getPort() + "/greetings";
                log.info(url);
                return url;
            })
            .flatMap(url -> call(http, url))
            .subscribe(greeting -> log.info("manual: " +
                greeting.toString()));
    }
}
```

COPY

```
}  
}
```

The actual work of making the HTTP request is done by a static method, `call`, that I have stashed in the application class. It expects a valid `WebClient` reference and an HTTP URL.

```
```java
```

```
static Flux<Greeting> call(WebClient http, String url) {  
    return http.get().uri(url).retrieve().bodyToFlux(Greeting.class);  
}
```

COPY

```
```
```

This approach works, but it's a *lot* of code to make one HTTP call.

## Using the `ReactorLoadBalancerExchangeFilterFunction`

This next approach hides a lot of that boilerplate load-balancing logic in a `WebClient` filter, of the type `ExchangeFilterFunction`, called `ReactorLoadBalancerExchangeFilterFunction`. We plug in that filter before making the request, and a *lot* of the previous code disappears.

```
@Component  
@Log4j2  
class WebClientRunner {  
  
    WebClientRunner(ReactiveLoadBalancer.Factory<ServiceInstance>  
serviceInstanceFactory) {  
  
        var filter = new  
ReactorLoadBalancerExchangeFilterFunction(serviceInstanceFactory);  
  
        var http = WebClient.builder()  
            .filter(filter)  
            .build();  
  
        call(http, "http://api/greetings").subscribe(greeting ->  
log.info("filter: " + greeting.toString()));  
    }  
}
```

COPY

Ahhhhhhh. Much better! But we can do better.

## The `@LoadBalanced` Annotation

In this final example, we'll have Spring Cloud configure the `WebClient` instance for us. This approach is excellent if *all* requests that pass through that shared `WebClient` instance

require load balancing. Just define a provider method for the `WebClient.Builder` and annotate it with `@LoadBalanced`. You can then use that `WebClient.Builder` to define a `WebClient` that'll load balance automatically for us.

```
@Bean
@LoadBalanced
WebClient.Builder builder() {
    return WebClient.builder();
}

@Bean
WebClient webClient(WebClient.Builder builder) {
    return builder.build();
}
```

COPY

With that done, our code shrinks to virtually nothing.

```
@Log4j2
@Component
class ConfiguredWebClientRunner {

    ConfiguredWebClientRunner(WebClient http) {
        call(http, "http://api/greetings").subscribe(greeting ->
log.info("configured: " + greeting.toString()));
    }
}
```

COPY

Now, *that* is convenient.


The load balancer uses round-robin load balancing, where it randomly distributes the load across any of a number of configured instances, using the

`org.springframework.cloud.loadbalancer.core.RoundRobinLoadBalancer` strategy. The nice thing about this is that this is pluggable. You can plugin in other heuristics if you wanted as well.

## Next Steps

In this Spring Tip installment, we've only begun to scratch the surface of the load balancing abstraction, but we have already achieved immense flexibility and conciseness. If you're further interested in customizing the load balancer, you might look into the

`@LoadBalancedClient` annotation.

14 Comments   spring.io   

 1 Login

 Recommend 2    Tweet    Share

Sort by Best



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name



**Angelho Suarez** • a year ago

does someone has an example with reactive feign + spring cloud loadbalancer ?

1 ^ | v • Reply • Share ›



**Shahed Satter** ➔ Angelho Suarez • a month ago

did you get to the bottom of this?

^ | v • Reply • Share ›



**Spandan** • 4 months ago • edited

Hello Josh, We need to distribute the load between two different micro services but call of the two microservices are not based on the endpoint(URL) but it is calling via it's SDK-Client.

Can we customize either of Netflix ribbon or Spring Cloud Gateway to achieve it or is there any other solution for this?

In short, we need to distribute the load over two two different methods in place of the two endpoint because our SDK is taking care of rest call & URL & PORT of SDK is the black box for us.

Any feedback are greatly appreciated, Thanks a lot in advance!!!

^ | v • Reply • Share ›



**Babu** • 6 months ago

while reading article just got a doubt so that posting, if i use spring client load balancer , cloud it woks but without changing code, can i deploy in IBM WAS or weblogic?.

^ | v • Reply • Share ›



**Arjun Patil** • 9 months ago

Can you point me to reference documentation for cloud LoadBalancer?

^ | v • Reply • Share ›



**Nishada Liyanage** • a year ago

**@Spencer Gibb**

How to configure Spring Cloud Loadbalancer to work without a discovery server. (by giving a list of ip addresses directly)

^ | v • Reply • Share ›



**Nishada Liyanage** → Nishada Liyanage • a year ago

check this link <https://stackoverflow.com/q...>

^ | v • Reply • Share ›



**TheFhdude** • a year ago • edited

Compared to Netflix Ribbon this looks way complicated tbh. In Ribbon I could just configure two hosts in application.properties and that was it. No need for service registry if I don't need it. I need a client load balancing feature but without any service registry. I want to just list the services in the application.properties. How can I do that with the Spring load balancer?

^ | v • Reply • Share ›



**Nishada Liyanage** → TheFhdude • a year ago

<https://stackoverflow.com/q...>

^ | v • Reply • Share ›



**barnwaldo** • a year ago

Is it possible to run parallel Spring Cloud Gateways and perform client load balancing across these for very high traffic applications. I would prefer that one Gateway is not a bottleneck for high traffic even if it is asynchronous as is the case here. (One Gateway could also be a problem if out-of-service for any reason.)

It appears do-able, but I have not found any examples of load balancing across gateways and would appreciate feedback here. Is there anything specific to load balancing across gateways that would differ from the Hello World app? For example, can the gateway URL just be added to the WebClientRunner and everything will work as it does in the Hello World App?

^ | v • Reply • Share ›

