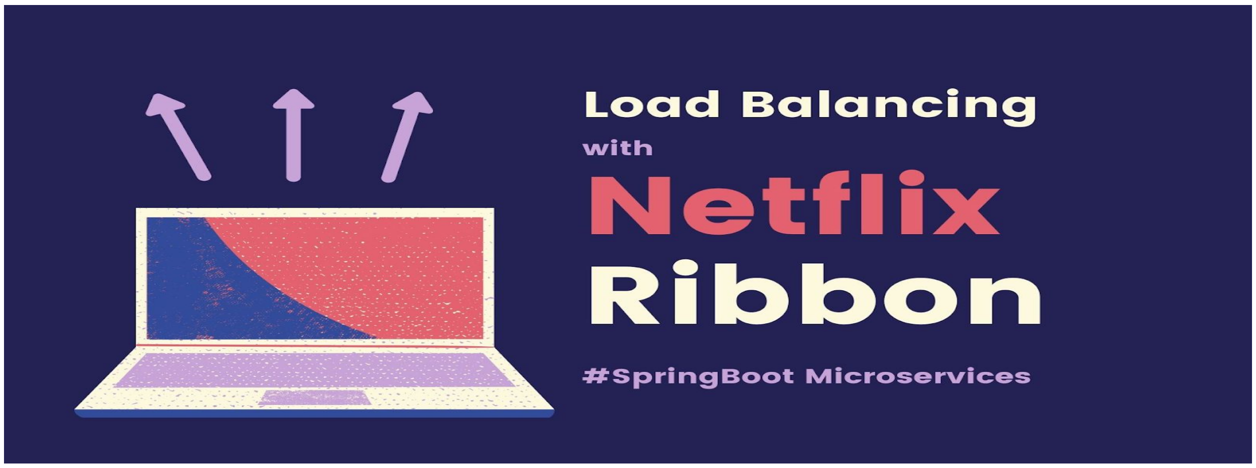
Microservices Load Balancing Notes 1



One of the most prominent reasons for evolution from monolithic to microservices architecture is horizontal scaling. It is required in modern day applications to improve user experience in the case of higher traffic for a particular service. We create multiple instances of the service in order to handle the large traffic of requests. But if the requests are not distributed among the instances effectively, then horizontal scaling is of no use.

Load balancing refers to efficiently distributing the incoming network traffic across a group of backend servers (multiple instances of the service).

In this article we will try to explain load balancing and how Netflix's Ribbon can be used for load balancing.

Types of load-balancing

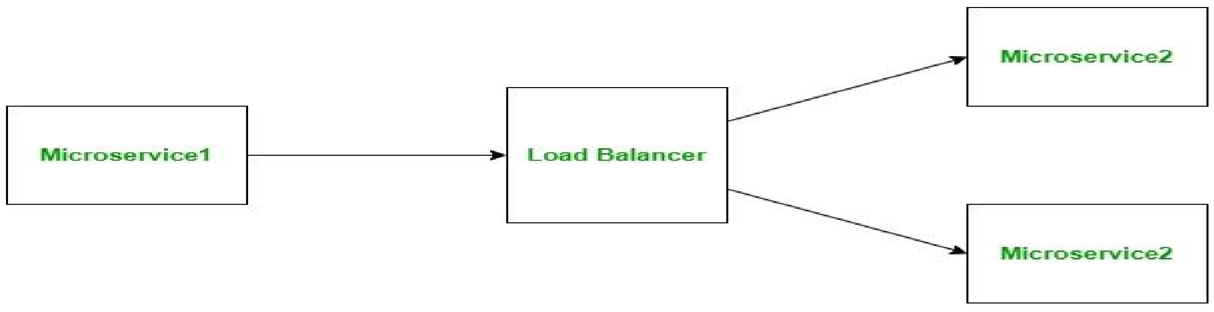
Load balancing can be of two types:

Server-side Load Balancing

Client-side Load Balancing

1. Server-side Load Balancing:
In Server-side load balancing, the instances of the service are deployed on multiple servers and then a load balancer is put in front of them. It is generally a hardware load balancer. All the incoming requests traffic firstly comes to this load balancer acting as a

middle component. It then decides to which server a particular request must be directed to based on some algorithm.



Server side load balancing

Disadvantages of Server-side load balancing

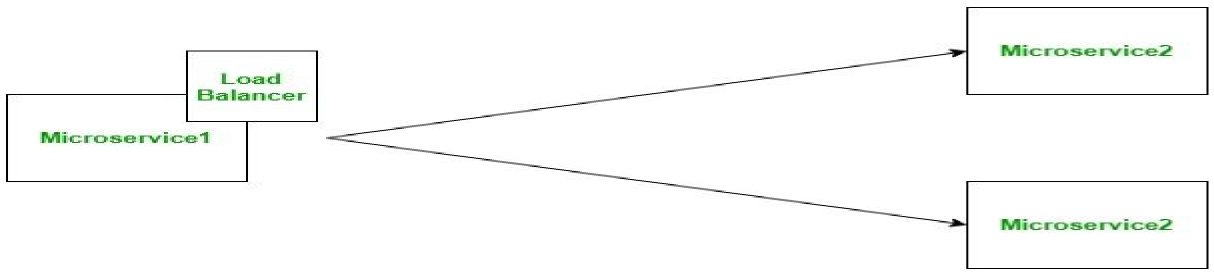
1. Server side load balancer acts as a single point of failure as if it fails, all the instances of the microservice become inaccessible as only the load balancer has the list of servers.

2. Since each microservice will have a separate load balancer, the overall complexity of the system increases and it becomes hard to manage.

3. The network latency increases as the number of hops for the request increases from one to two with the load balancer, one to the load balancer and then another from load balancer to the microservice.

2. Client-side Load Balancing

The instances of the service are deployed on multiple servers. Load balancer's logic is part of the client itself, it holds the list of servers and decides to which server a particular request must be directed to based on some algorithm. These client side load balancers are also known as software load balancers.



Client side load balancing

Microservices Load Balancing Notes 3

Disadvantages of Client-side load balancing

1. The load balancer's logic is mixed up with the microservice code.

What is Netflix's Ribbon
As per official website, Netflix's Ribbon is an Inter Process Communication (remote procedure calls) library with built in client side(software) load balancer and is a part of Netflix Open Source Software (Netflix OSS).

Following are the features of Ribbon:

1. Load balancing: It provides client side load balancing functionality.

2. Fault tolerance: It can be used to determine whether the servers are up or not and can also detect those servers that are down and hence, ignore them for sending the further requests.

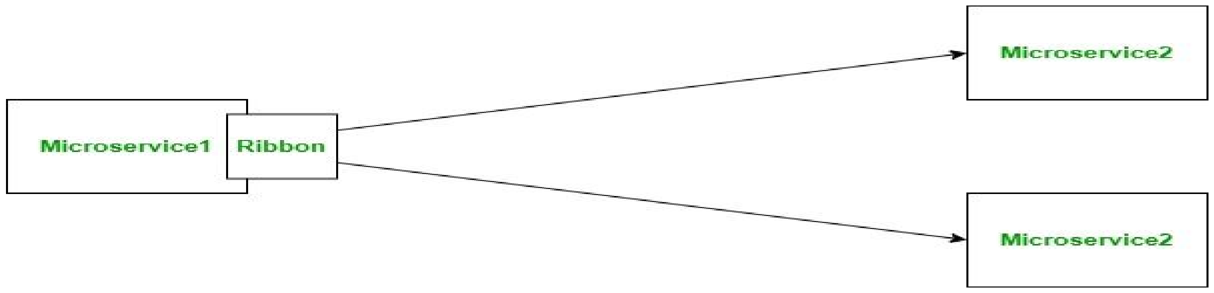
3. Configurable load balancing rules: By default ribbon uses RoundRobinRule for distributing requests among servers. In addition to it, it also provides AvailabilityFilteringRule and WeightedResponseTimeRule. We can also define our custom rules as per our needs.

4. It supports multiple protocols like HTTP, TCP, UDP etc.

Using Ribbon with Java (Spring) Microservice
Let's understand Ribbon using an example application.

We will create two microservices: Microservice1 and Microservice2. The Microservice2 will be having its two instances running and Microservice1 will call Microservice2.

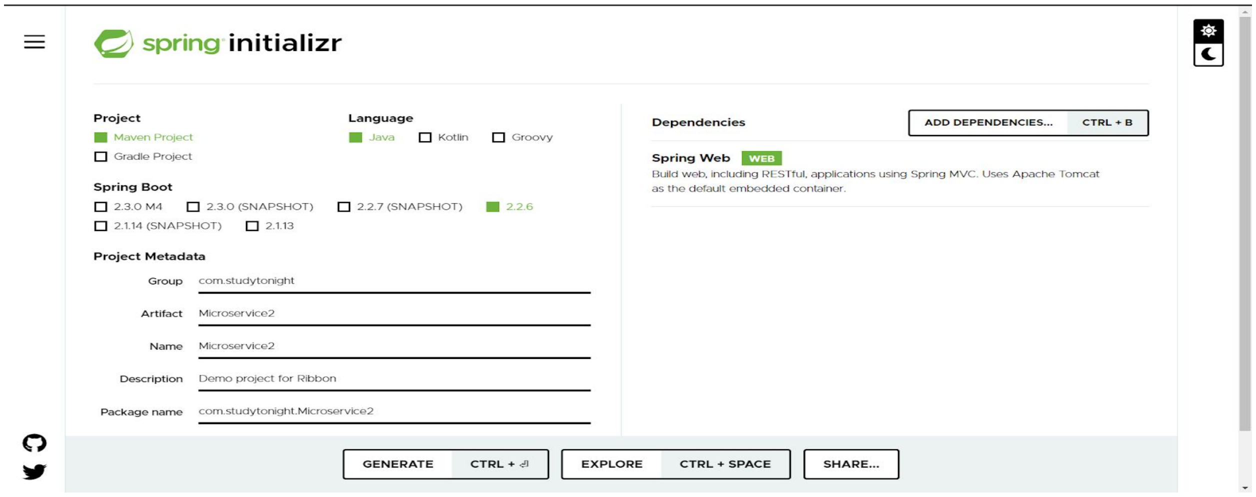
Ribbon logic will be included in Microservice1 in order to load balance the requests among the two instances of Microservice2. We will be using spring tool suite for this example:



Microservices Load Balancing Notes 4

Step 0: Create Microservice2 and run its two instances on ports: 8200 and 8201.

Let's work on creating Microservice2. Go to Spring Initializr and create a new spring boot application as described in the below snapshot. Add Spring Web dependency and click on Generate. A zip file will be downloaded, extract it. Then open your spring tool suite (STS) and click on File > Import > Maven > Existing Maven Project > Browse and open the project you extracted.



Project
[✓ Maven Project]
Gradle Project

Language
[✓ Java] Kotlin Groovy

Spring Boot
2.3.0 M4 · 2.3.0 (SNAPSHOT) · 2.2.7 (SNAPSHOT) · [✓ 2.2.6]
2.1.14 (SNAPSHOT) · 2.1.13

Project Metadata

Group	com.studytonight
Artifact	Microservice2
Name	Microservice2
Description	Demo project for Ribbon
Package name	com.studytonight.Microservice2

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Create a controller class as described below:

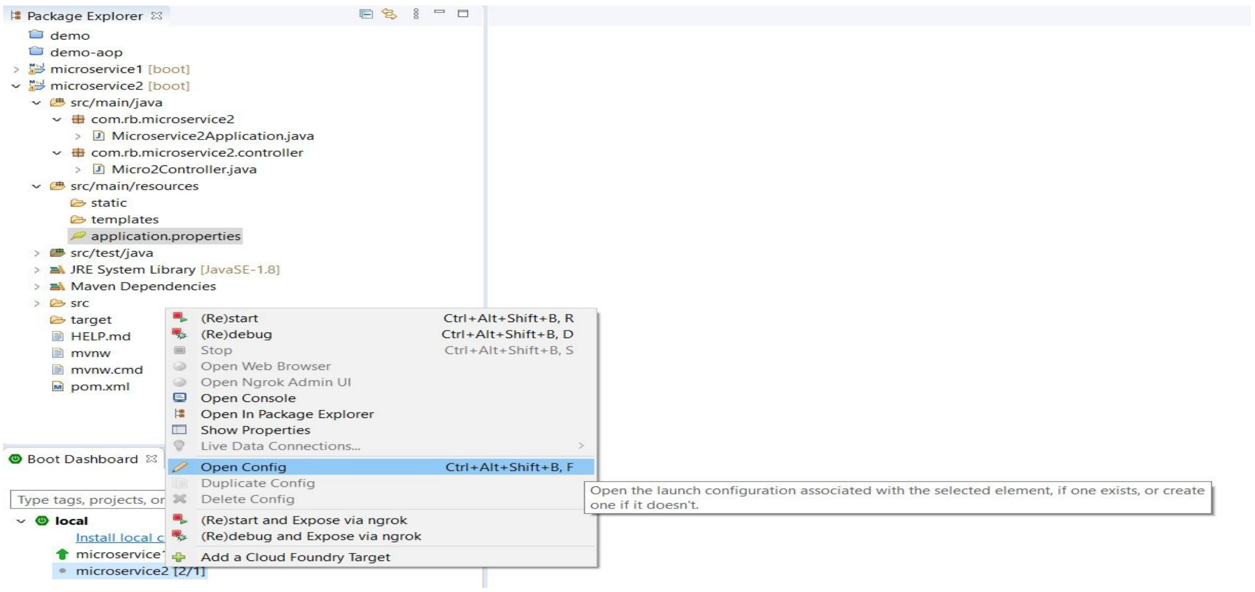
Microservice2 Controller:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/microservice2")
public class Micro2Controller {

  @Value("${server.port}")
  private String port;

  @GetMapping("/port")
```

Microservices Load Balancing Notes 5

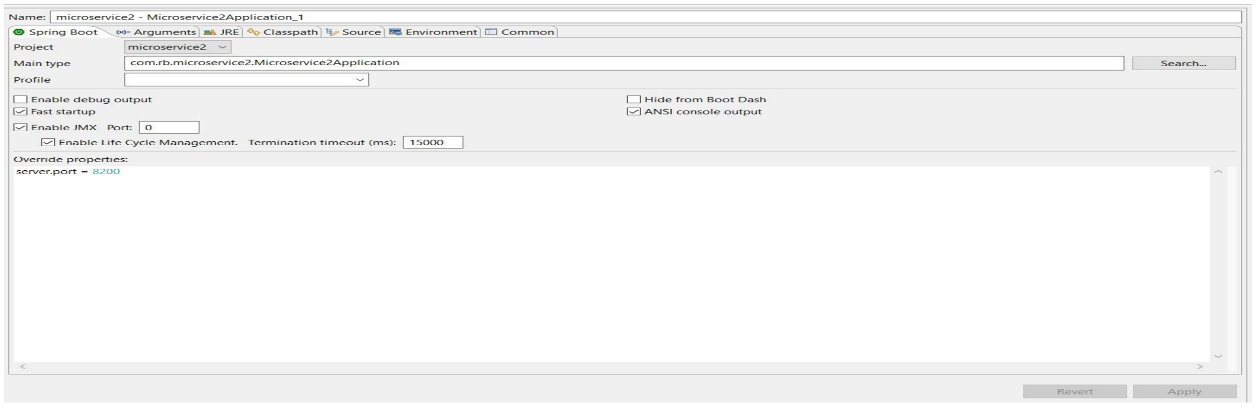
```
public String getPort()
{
      return port;
}
}
```

Right-click on the Microservice2 under BootDashboard. Select "Open Config" and set the name as "microservice2?—?Microservice2Application_2" and override property "server.port = 8201".



Open the launch configuration associated with the selected element, if one exists, or create one if it doesn't.

Now right click on the Microservice2 under BootDashboard. Select "Duplicate Config" to create another instance. Set the name of this instance by right-clicking on it and selecting "Open Config" as "microservice2?—?Microservice2Application_1" and override property "server.port = 8200".



Name: microservice2 - Microservice2Application_1

Project: microservice2

Main type: com.rb.microservice2.Microservice2Application

Profile:

☐ Enable debug output
☑ Fast startup
☑ Enable JMX Port: 0
☑ Enable Life Cycle Management. Termination timeout (ms): 15000
☐ Hide from Boot Dash
☑ ANSI console output

Override properties:

```
server.port = 8200
```

Microservices Load Balancing Notes 6

Step 1: Create Microservice1 and add Ribbon dependency in its "pom.xml" file.
As described above, similarly create Microservice1 from Spring Initializer(just like we did for Microservice2). Here, in addition to Spring Web, add spring-cloud-starter-netflix-eureka-client dependency too. Then click on Generate and follow the same steps as before.

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

Step 2: Create a Configuration file with @Configuration annotation.
Create a bean of RestTemplate with @Bean and @LoadBalanced annotations. The @LoadBalanced annotation on the RestTemplate indicates that we want it to be load balanced and it will use RibbonClient to get the list of server addresses.

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

@Configuration
public class Micro1Config {

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

Step 3: Create a Controller class annotated with @RestController and @RibbonClient(name="micro1")
Autowire the RestTemplate field and use it to call Microservice2 from the controller method. The @RibbonClient annotation is used to customize the Ribbon settings or if we are not using any service discovery then its used to specify the list of servers.

The name we define in the name field of the @RibbonClient annotation is used as a prefix of the ribbon configurations in "application.properties" and as "logical identifier" in the URL we pass to the RestTemplate. The configuration field is used to specify a configuration class that holds all our customizations as @Bean.

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

@RestController
@RibbonClient(name = "micro1")
public class Micro1Controller {

  @Autowired
  RestTemplate restTemplate;

  @GetMapping("/getmicro2")
  public String getMicro2Instance()
  {
       String url = "http://micro1/microservice2/port";
       String port = "Currently hitting instance running on port: " +
                    restTemplate.getForObject(url, String.class);
       return port;
  }
}
```

Step 4: Disable Eureka service discovery component

Now we will disable the Eureka service discovery component by setting eureka.enabled = false and add the list of servers in the "application.properties" using named client "micro1".

```
micro1.ribbon.eureka.enabled = false
micro1.ribbon.listOfServers = localhost:8200, localhost:8201
```

Step 5: Test the Application

Now let's test our application. Open Postman and hit the URL "http://localhost:8080/getmicro2" 4 times:

You should get the following response:

```
Currently hitting instance running on port: 8200
Currently hitting instance running on port: 8201
Currently hitting instance running on port: 8200
Currently hitting instance running on port: 8201
```