# Client Side Load Balancing

**Table of Contents**

---

Estimated Time: 25 minutes

---

## Requirements

- Provided by e-mail last week
- Completion of Spring Cloud Config Lab and Service Discovery Lab

---

## What You Will Learn

---

- How to use Ribbon as a client side load balancer
- How to use a Ribbon enabled `RestTemplate`

---

# Exercises

---

## Start the `config-server`, `service-registry`, and `fortune-service`

1) Make sure config-server, service-registry, and fortune-service are running, as per the previous labs.

## Set up `greeting-ribbon`

#### ***No additions to the pom.xml***

In this case, we don't need to explicitly include Ribbon support in the `pom.xml`. Ribbon support is pulled in through transitive dependencies (dependencies of the dependencies we have already defined).

1) Review the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-ribbon/src/main/java/io/pivotal/greeting/GreetingController.java`. Notice the `loadBalancerClient`. It is a client side load balancer (Ribbon). Review the `fetchFortuneServiceUrl()` method. Ribbon is integrated with Eureka so that it can discover services as well. Notice how the `loadBalancerClient` chooses a service instance by name.

```
@Controller
public class GreetingController {
  Logger logger = LoggerFactory.getLogger(GreetingController.class);
  @Autowired
  private LoadBalancerClient loadBalancerClient;
  @RequestMapping("/")
  String getGreeting(Model model) {
    logger.debug("Adding greeting");
    model.addAttribute("msg", "Greetings!!!");
```

```
      RestTemplate restTemplate = new RestTemplate();

      String fortune = restTemplate.getForObject(fetchFortuneServiceUrl(), Str
ing.class);

      logger.debug("Adding fortune");

      model.addAttribute("fortune", fortune);

      //resolves to the greeting.vm velocity template

      return "greeting";

   }

   private String fetchFortuneServiceUrl() {

     ServiceInstance instance = loadBalancerClient.choose("fortune-service");

     logger.debug("uri: {}", instance.getUri().toString());

     logger.debug("serviceId: {}", instance.getServiceId());

     return instance.getUri().toString();

   }

}
```

2) Package and push the `greeting-ribbon` application.

```
$ mvn clean package

$ cf push greeting-ribbon -p target/greeting-ribbon-0.0.1-SNAPSHOT.jar -m 512M
 --random-route --no-start
```

3) Bind services for the `greeting-ribbon` application.

```
$ cf bind-service greeting-ribbon config-server

$ cf bind-service greeting-ribbon service-registry
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. We don't need to restage at this time.

4) Set the `TRUST_CERTS` environment variable for the `greeting-ribbon` application (our PCF instance is using self-signed SSL certificates).

```
$ cf set-env greeting-ribbon TRUST_CERTS <your api endpoint>
```

You can safely ignore the *TIP: Use ‘cf restage’ to ensure your env variable changes take effect* message from the CLI. We don’t need to restage at this time.

5) Start the `greeting-ribbon` app.

```
$ cf start greeting-ribbon
```

6) After the a few moments, check the `service-registry`. Confirm the `greeting-ribbon` app is registered.

7) Refresh the `greeting-ribbon` `/` endpoint. Confirm you are seeing fortunes. Refresh as desired. Also review the log output for the `greeting-ribbon` app. See the `uri` and `serviceId` being logged.

5) Stop the `greeting-ribbon` application.

```
$ cf stop greeting-ribbon
```

## Set up `greeting-ribbon-rest`

***No additions to the pom.xml***

In this case, we don’t need to explicitly include Ribbon support in the `pom.xml`. Ribbon support is pulled in through transitive dependencies (dependencies of the dependencies we have already defined).

1) Review the the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-ribbon-rest/src/main/java/io/pivotal/GreetingRibbonRestApplication.java`. In addition to the standard `@EnableDiscoveryClient`annotation, we're also configuring a `RestTemplate` bean. It is not the usual `RestTemplate`, it is load balanced by Ribbon. The `@LoadBalanced` annotation is a qualifier to ensure we get the load balanced `RestTemplate` injected. This further simplifies application code.

```java
@SpringBootApplication
@EnableDiscoveryClient
public class GreetingRibbonRestApplication {
    public static void main(String[] args) {
     SpringApplication.run(GreetingRibbonRestApplication.class, args);
   }
   @LoadBalanced
   @Bean
   RestTemplate restTemplate() {
     return new RestTemplate();
   }
}
```

2) Review the the following file: `$SPRING_CLOUD_SERVICES_LABS_HOME/greeting-ribbon-rest/src/main/java/io/pivotal/greeting/GreetingController.java`. Here we autowire the restTemplate we configured in the previous step. Note also that the spring cloud API is smart enough to dynamically substitute the name of the service `fortune-service` in the url parameter for `getForObject` with its load-balanced, discovered url.

```java
@Controller
public class GreetingController {
    Logger logger = LoggerFactory.getLogger(GreetingController.class);
    @Autowired
    private RestTemplate restTemplate;
    @RequestMapping("/")
    String getGreeting(Model model) {
      logger.debug("Adding greeting");
      model.addAttribute("msg", "Greetings!!!");
      String fortune = restTemplate.getForObject("http://fortune-service", Str
ing.class);
      logger.debug("Adding fortune");
      model.addAttribute("fortune", fortune);
      //resolves to the greeting.vm velocity template
      return "greeting";
   }
}
```

3) Package and push the `greeting-ribbon-rest` application.

```
$ mvn clean package
$ cf push greeting-ribbon-rest -p target/greeting-ribbon-rest-0.0.1-SNAPSHOT.j
ar -m 512M --random-route --no-start
```

4) Bind services for the `greeting-ribbon-rest` application.

```
$ cf bind-service greeting-ribbon-rest config-server
$ cf bind-service greeting-ribbon-rest service-registry
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. We don't need to restage at this time.

5) Set the `TRUST_CERTS` environment variable for the `greeting-ribbon-rest` application (our PCF instance is using self-signed SSL certificates).

```
$ cf set-env greeting-ribbon-rest TRUST_CERTS <your api endpoint>
```

You can safely ignore the *TIP: Use 'cf restage' to ensure your env variable changes take effect* message from the CLI. We don't need to restage at this time.

6) Start the `greeting-ribbon-rest` app.

```
$ cf start greeting-ribbon-rest
```

7) After the a few moments, check the `service-registry`. Confirm the `greeting-ribbon-rest` app is registered.

8) Refresh the `greeting-ribbon-rest` `/` endpoint.

9) Stop the greeting-ribbon-rest app.

```
$ cf stop greeting-ribbon-rest
```

**Note About This Lab**

If services (e.g. `fortune-service`) are registering using the first Cloud Foundry URI (using the `route` registration method) this means that requests to them are being routed through the `router` and subsequently load balanced at that layer. Therefore, client side load balancing doesn't occur.

Pivotal Cloud Foundry has recently added support for allowing cross container communication. This will allow applications to communicate with each other without passing through the `router`. As applied to client-side load balancing, services such as `fortune-service` would register with Eureka using their container IP addresses. Allowing clients to reach them without going through the `router`. This is known as using the `direct` registration method.

For more details, please read the following.

---

Back to TOP