Spring Cloud Netflix: Client Side Load Balancing

Table of Contents

Estimated Time: 25 minutes

Requirements

Lab Requirements (/spring-cloud-services/requirements)

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) Start the `config-server` in a terminal window. You may have terminal windows still open from previous labs. They may be reused for this lab.

```
$ cd $SPRING_CLOUD_SERVICES_LABS_HOME/config-server
$ mvn clean spring-boot:run
```

2) Start the `service-registry`

```
$ cd $SPRING_CLOUD_SERVICES_LABS_HOME/service-registry
$ mvn clean spring-boot:run
```

3) Start the `fortune-service`

```
$ cd $SPRING_CLOUD_SERVICES_LABS_HOME/fortune-service
$ mvn clean spring-boot:run
```

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 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(), String.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) Open a new terminal window. Start the `greeting-ribbon` app.

```
$ cd $SPRING_CLOUD_SERVICES_LABS_HOME/greeting-ribbon
$ mvn clean spring-boot:run
```

3) After the a few moments, check the `service-registry` dashboard http://localhost:8761 (http://localhost:8761). Confirm the `greeting-ribbon` app is registered.

4) Browse to http://localhost:8080/ (http://localhost:8080/) to the `greeting-ribbon` application. Confirm you are seeing fortunes. Refresh as desired. Also review the terminal output for the `greeting-ribbon` app. See the `uri` and `serviceId` being logged.

5) Stop the `greeting-ribbon` application.

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/greeting/GreetingController.java`. Notice the `RestTemplate`. 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.

```
@Controller
public class GreetingController {

        Logger logger = LoggerFactory
                        .getLogger(GreetingController.class);




        @Autowired
        @LoadBalanced
        private RestTemplate restTemplate;

        @RequestMapping("/")
        String getGreeting(Model model){

                logger.debug("Adding greeting");
                model.addAttribute("msg", "Greetings!!!");


        String fortune = restTemplate.getForObject("http://fortune-service", String.class);

                logger.debug("Adding fortune");
                model.addAttribute("fortune", fortune);

                //resolves to the greeting.vm velocity template
                return "greeting";
        }
```

```
}
```

2) Open a new terminal window. Start the `greeting-ribbon-rest` app.

```
$ cd $SPRING_CLOUD_SERVICES_LABS_HOME/greeting-ribbon-rest
$ mvn clean spring-boot:run
```

3) After the a few moments, check the `service-registry` dashboard at http://localhost:8761 (http://localhost:8761). Confirm the `greeting-ribbon-rest` app is registered.

4) Browse to http://localhost:8080/ (http://localhost:8080/) to the `greeting-ribbon-rest` application. Confirm you are seeing fortunes. Refresh as desired. Also review the terminal output for the `greeting-ribbon-rest` app.

5) When done stop the `config-server`, `service-registry`, `fortune-service` and `greeting-ribbon-rest` applications.

Deploy the `greeting-ribbon-rest` to PCF

1) If using the `route` registration method (http://docs.pivotal.io/spring-cloud-services/service-registry/registering-a-service.html), in your fork of the `app-config` repo add an additional section (`ribbon.IsSecure`) to the `$APP_CONFIG_REPO_HOME/application.yml` file as seen below and push back to GitHub. If using the `direct` method then skip this step.

```
security:
  basic:
    enabled: false

management:
  security:
    enabled: false

logging:
  level:
    io:
      pivotal: DEBUG

ribbon: # <---NEW SECTION
  IsSecure: false
```

Pivotal Cloud Foundry with HTTPS Only

If your Pivotal Cloud Foundry environment has been configured to *only* accept HTTPS traffic and is using the `route` registration method then skip this step, however you will need to change all Ribbon code examples moving forward in the labs that use `http` to `https` before deploying to your Pivotal Cloud Foundry environment. For instance, the `GreetingController` from above would have the following change:

```
String fortune = restTemplate.getForObject("https://fortune-service", String.class);
```

2) 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.jar -m 512M --r
andom-route --no-start
```

3) 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.

4) If using self signed certificates, set the `CF_TARGET` environment variable for the `greeting-ribbon-rest` application.

```
$ cf set-env greeting-ribbon-rest CF_TARGET <your api endpoint - make sure it starts with "h
ttps://">
```

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-rest` app.

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

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

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

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 (http://docs.pivotal.io/spring-cloud-services/service-registry/registering-a-service.html).

Back to TOP
