

# Curso Alura SpringCloud (Microservices)

## Aula 1

O curso usa **spring boot versão 2.15** com **spring cloud** e **mariaDB 10.3**

-> vou usar ultima versao de tudo com mongoDB

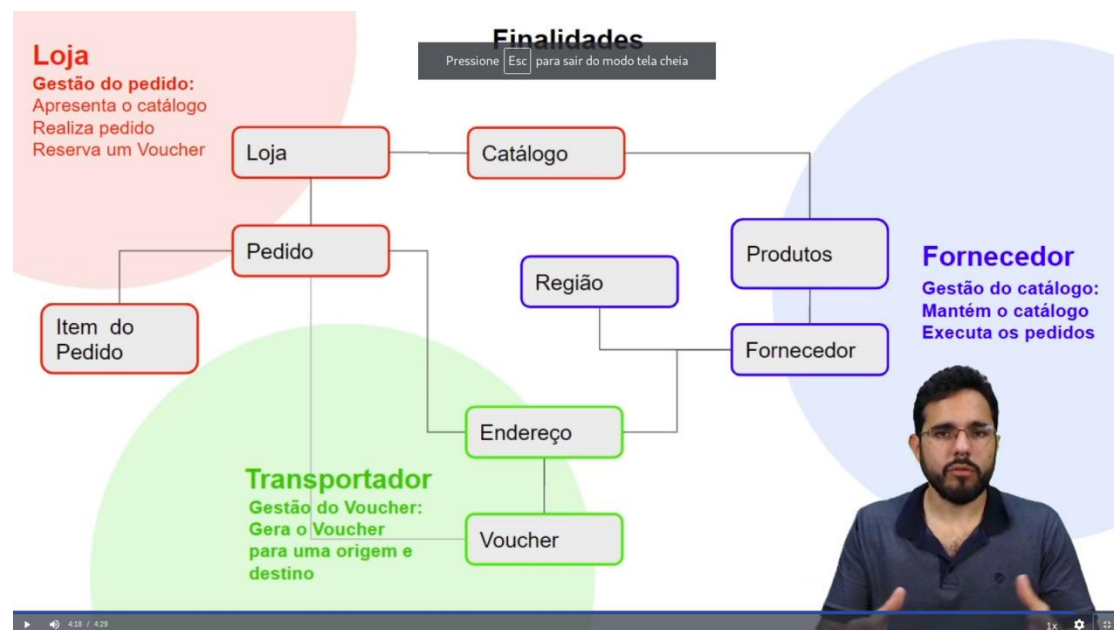
Neste caso, nesse microserviço nao precisamos do banco, já que ele não persistirá nada.

Slides:

[https://docs.google.com/presentation/d/e/2PACX-1vQTuAjVqa3-k7wLLFXB2sHs9ZSeUJW8Zr11VoAUGJoF\\_gclEUK\\_VgBfeW7-Soifk9abCWDfiLIxHmj/pub?start=true&loop=false&delayms=1000&slide=id.p](https://docs.google.com/presentation/d/e/2PACX-1vQTuAjVqa3-k7wLLFXB2sHs9ZSeUJW8Zr11VoAUGJoF_gclEUK_VgBfeW7-Soifk9abCWDfiLIxHmj/pub?start=true&loop=false&delayms=1000&slide=id.p)

\* O microserviço é a implementação de um contexto (separado).

O foco principal dos microserviços é a separação da modelagem da aplicação em contextos coesos e **independentes** uns dos outros.



Cada uma dessas cores é um contexto diferente.

\* Ele gerou o projeto a partir do STS (spring tools suite)

-> Usei o spring initializr e já passei tanto o spring web quanto o mongoDB do spring data

## AULA 02:

\* Usa-se o **RestTemplate** para comunicação entre microserviços (passando o endpoint que queremos acessar).

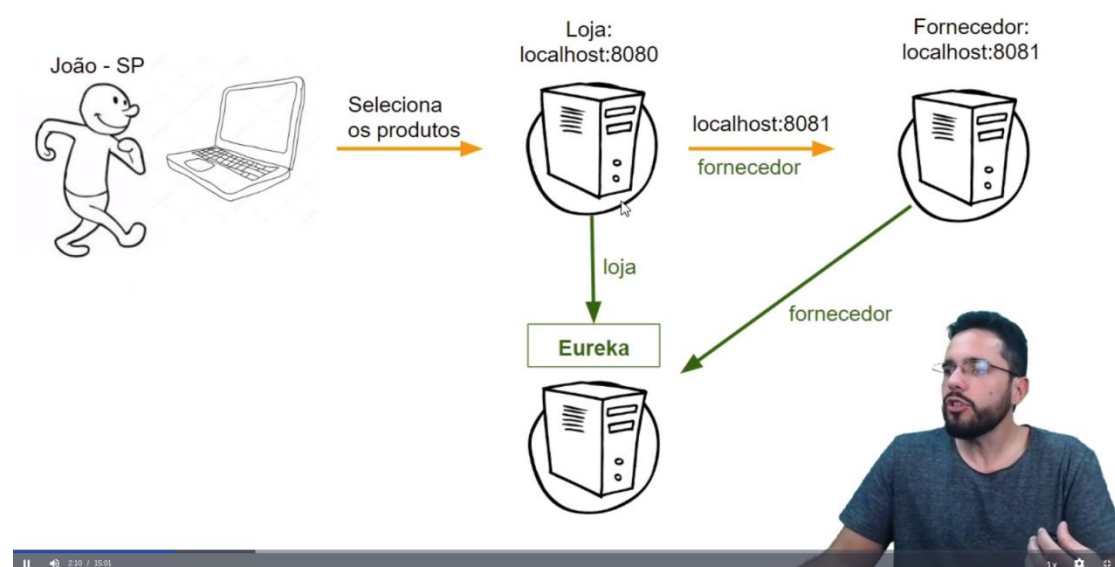
Ex.: `RestTemplate client = new RestTemplate();`  
`Client.exchange("http://localhost:8081/info", HttpMethod.GET, null, InfoDTO.class);`  
A porta é diferente, pois subiremos o outro microserviço em outra porta, claro.

Ao criar o segundo microserviço, usei o spring initializr também, porém, com o MongoDB do spring data e o JPA.

Para trocar a porta, basta ir no **application.properties** ou criar um arquivo **application.yml**

Porém, em código, deixamos hardcoded o localhost:8081.

Para resolver isso, usamos um **service discovery (nesse caso o Eureka)**. Ou seja, nossos serviços vão se registrar no service discovery, com o nome e endereço. Com isso, o Eureka seria o intermediador entre os serviços.



Para isso, criamos um novo projeto para o Eureka (padrao).

Ao selecionar as dependencias, selecionamos apenas o **Eureka server**.

Apenas configuramos o application.properties para remover o servidor eureka de se tornar um serviço, já que ele é o proprio eureka.

E passamos uma anotação `@EnableEurekaService` no starter.  
Alteramos a porta também, para a padrao do eureka (8761)

Quando passamos o endpoint <http://localhost:8761/eureka/apps>

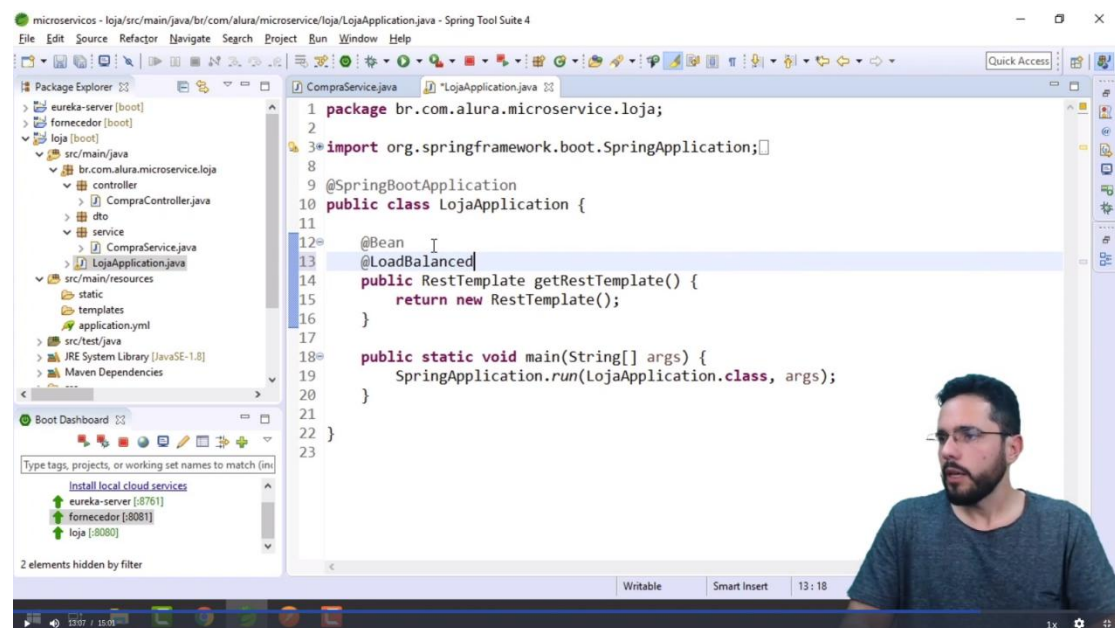
Ele nos lista **todas** as aplicações que se encontram registradas no eureka.

Para adicionarmos apps no eureka, passamos via **application.yml** as configurações do eureka, além disso, precisamos passar o **spring.application.name** com o nome da aplicação para o eureka fazer a “conexão” pelo nome da aplicação.

Precisamos adicionar no pom o **eureka discovery client** ou no STS simplesmente clicar com o botão direito no projeto e selecionar **spring > edit starter** e colocar a dependência do eureka.

Com isso, não usaremos mais o **restTemplate** hardcoded, precisamos configurar um **restTemplate** para usar o eureka na aplicação. Ou seja, precisamos editar o **application** (o arquivo de start dos projetos do spring boot).

Para ter a inteligência de resolução por nome de serviço, colocamos a anotação **@LoadBalanced** no bean, que faz com que o eureka resolva o IP e porta a partir do nome dado.



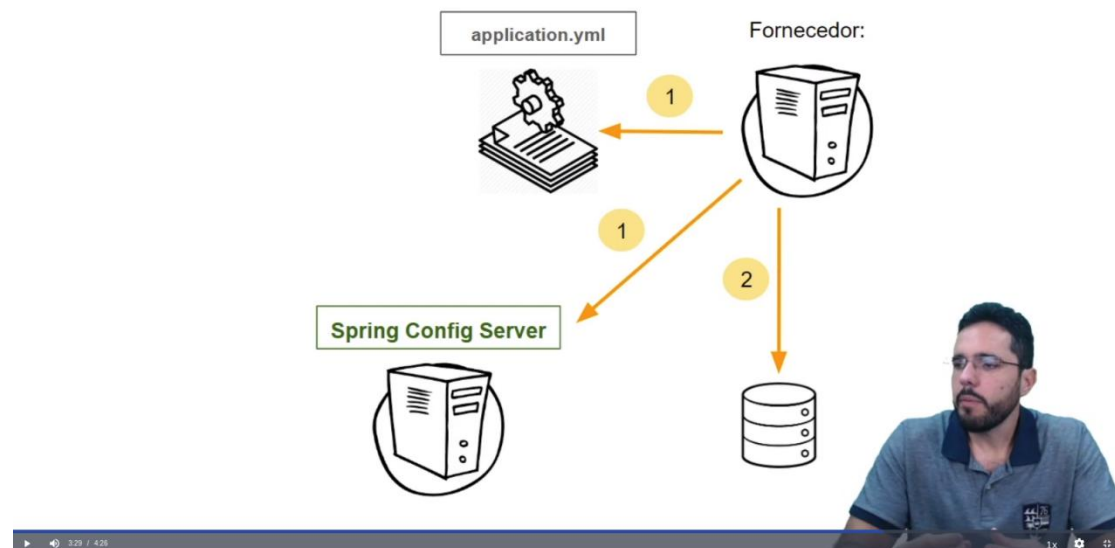
Resumindo:

1. Criamos um projeto usando o eureka discovery server;
2. Ajustamos as aplicações (microserviços) para usar o eureka discovery client, configuramos-o e consumimos o server.

### Aula 03:

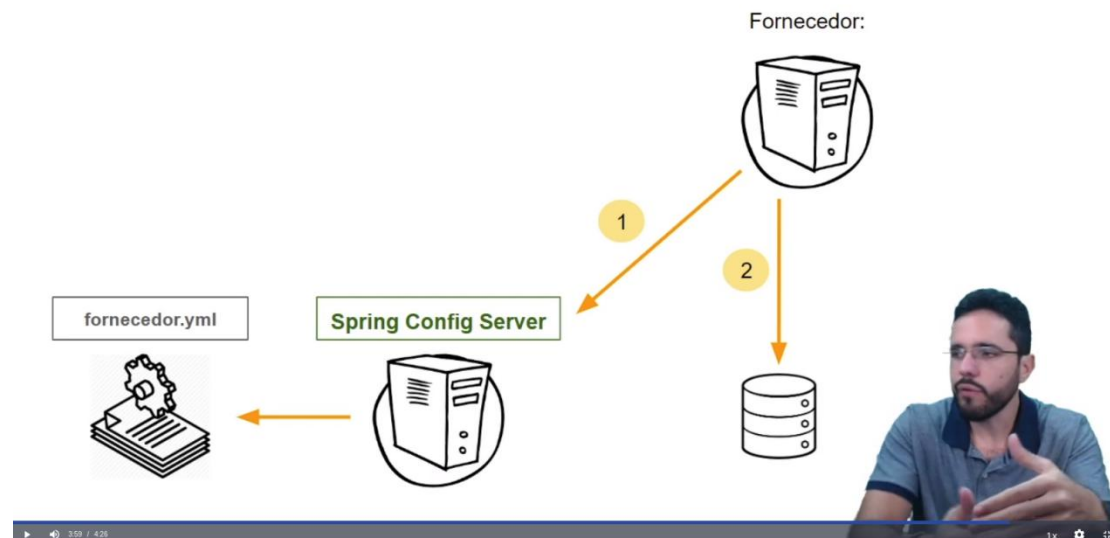
\* Configuração de ambiente usando o **Spring Config Server**. É muito trabalhoso configurarmos cada microserviço para utilizar usuário/senha e BD no application.yml, etc, etc. Para isso, utilizaremos o **Spring Config Server** do pacote **Spring Cloud**.

“ Os microserviços são preparados para um ambiente (cloud), cuja precificação é diretamente relacionada à quantidade de máquinas e ao uso de seus recursos de infraestrutura. Para reduzir esse custo, aplicações de microserviços se encaixam bem, pois é possível escalar automaticamente, de acordo com a demanda, e em questão de segundos, pedaços do que antes era uma única aplicação. Nesse cenário, configurar manualmente os servidores com as configurações necessárias para cada aplicação é impraticável.”



Logo, precisamos extrair os valores de configurações para salvarmos em um repositório que será utilizado pelo Spring Config Server.

Ele nos permite usarmos um endpoint REST para obtermos as configurações que precisamos para cada microserviço.

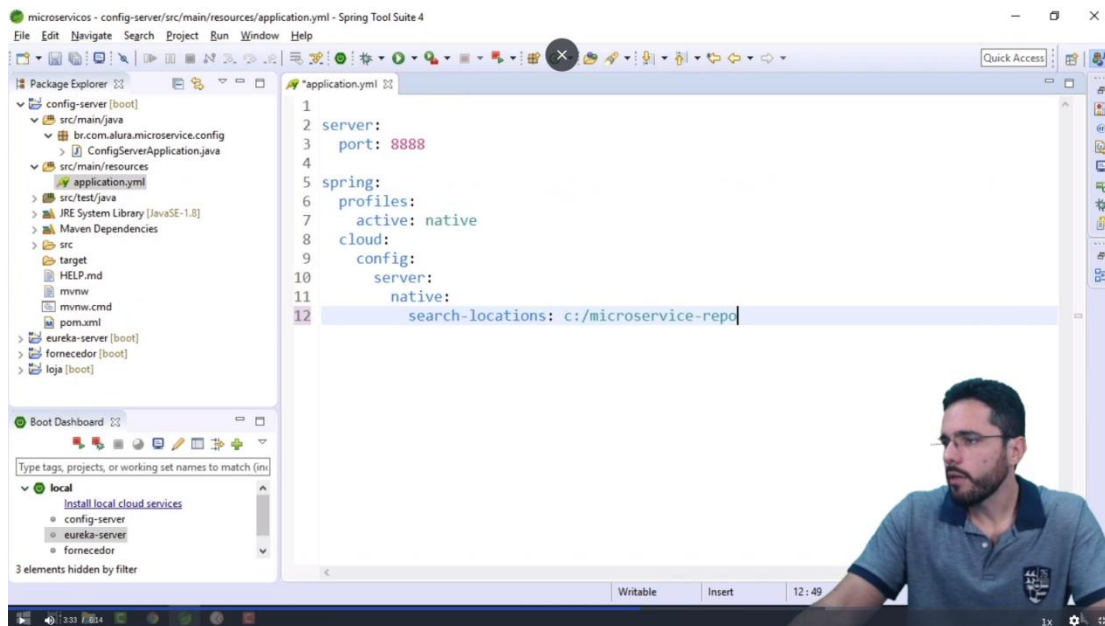


Criamos um “fornecedor.yml” em algum lugar do nosso sistema (system config, etc) e ensinamos o Spring Config Server qual o path para obter esses dados. (Nesse caso, as configurações que serão utilizados pelo serviço de fornecedor).  
Da seguinte forma:

```
spring:
  datasource:
    password: root
    username: root
    url: jdbc:mysql://localhost:3306/fornecedor
  jpa:
    hibernate:
      ddl-auto: update
```

Criamos então, mais um projeto com o nome de ConfigServer.

1. Colocar a anotação **@EnableConfigServer**;
2. Passar a porta no application.yml como 8888;
3. Essa configuração abaixo:



Com a configuração feita, podemos chamar o endpoint  
<http://localhost:8888/fornecedor/default>

Onde fornecedor é o nome do nosso arquivo (default é sem extensão), caso tenhamos outros arquivos “fornecedor-dev.yml”...

É só trocarmos o default do endpoint para dev, ou algo do tipo.

Agora, precisamos fazer com que o microserviço se comunique com o Spring Configuration Server.

Para isso, basta colocarmos o profile usado (default) e onde está o spring Config Server (porta 8888) no application.yml do fornecedor.

Porém, precisamos colocar essa configuração **antes** do spring subir, pois o **hibernate** vai querer fazer a configuração de banco, etc, antes dessa configuração subir (neste application.yml).

Para isso, usamos o **bootstrap.yml** para subir essa config antes.

Ou seja, a aplicação lê esse arquivo **ANTES** de subir a aplicação.

**Problema: não conseguir configurar o mongoDB com o spring cloud.**

**Solução:** Retirei o settings.xml do m2 para trazer os repositórios padrões, e foi ajustado os **pom.xml**

Pom do Config Server:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server-mongodb</artifactId>
  <version>0.0.3.BUILD-SNAPSHOT</version>
</dependency>
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
```

```

<url>https://repo.spring.io/libs-snapshot-local</url>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
</repository>
<repository>
  <id>ojo-snapshots</id>
  <name>OJO Snapshots</name>

<url>https://oss.jfrog.org/artifactory/libs-snapshot</url>
>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
</repository>
</repositories>

```

Além disso, no config server application, foi trocada a annotation:

```

@SpringBootApplication
@EnableMongoConfigServer
public class ConfigServerApplication {

```

Para usar o mongo.

Pom do Config Client: (Adicionada essa dependencia)

```

<properties>
  <java.version>1.8</java.version>

<spring-cloud.version>Hoxton.SR1</spring-cloud.version>
</properties>

```

```

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

```

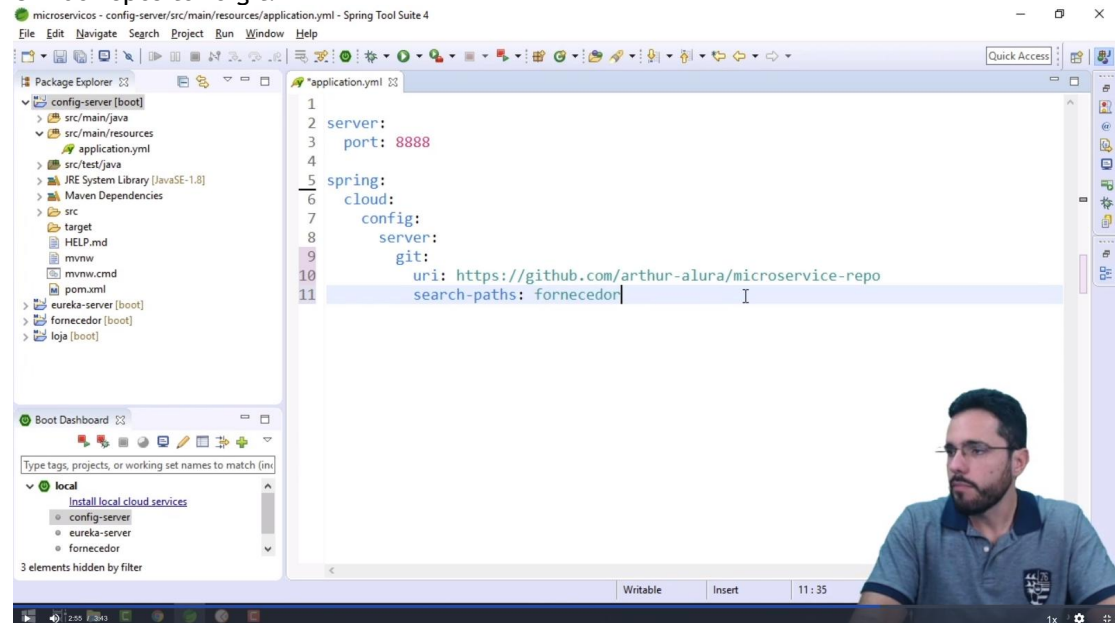
\* É necessário passar a URI vindo do atlas e a database escolhida no arquivo de configuração de repositórios;

Todo: Como encriptar o password??

Todo: Como usar o SHA key para configurar um repositório de configs no git com o Spring Cloud Config?

Podemos também criar o repositório de arquivos no github. Basta criarmos um projeto no github com as configurações.

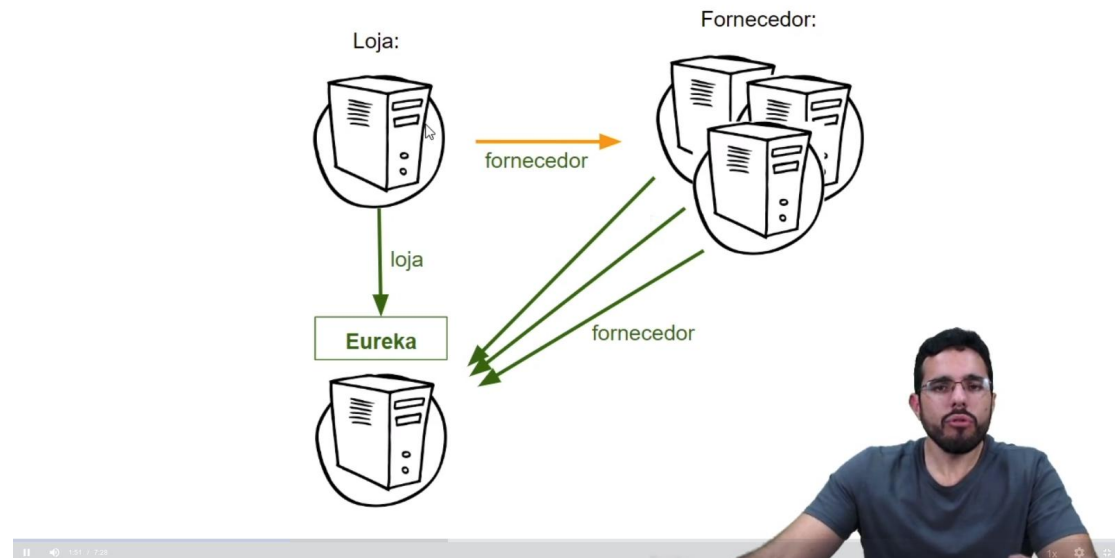
Com isso, podemos ir ao Config Server (application.yml) e trocar o **native** para **git** e passar a URI do repositório git.





## AULA 04:

Problema: E se tivermos várias instâncias de fornecedores, como saber qual usar? (O eureka não faz esse tipo de funcionalidade).



Para isso, usamos o Ribbon (uma lib que faz o client-side loadBalancer, ou seja, distribuição de carga).

\* Configuramos o Ribbon no `@LoadBalanced` na instância do `restTemplate`!

Podemos também saber as instâncias disponíveis usando o `DiscoveryClient`.

\* O Feign é uma lib usada para gerenciar os IP's e portas dos microserviços (integrado com o Ribbon) e nos ajuda a melhorar o código de comunicações entre microserviços. Ao invés de usarmos o `restTemplate` propriamente dito, usaremos serviços como no padrão MVC.

Para isso:

1. Adicionamos o OpenFeign no projeto que queremos usar (no caso a loja, que faz a comunicação com o fornecedor);
2. Criamos uma interface (`fornecedorClient`) com a anotação de `@FeignClient("idDoServico")` No caso, "fornecedor";
3. Colocamos apenas a referência para os métodos do `fornecedorService`;
4. Injetamos essa interface de serviço no `LojaService` e usamos os métodos criados;
5. Colocamos a anotação na aplicação que usará o feign (No caso, na loja) `@EnableFeignClients`.

## AULA 05: Rastreabilidade de logs

Cada microserviço gera um log, logo fica difícil de rastreá-los.

Existem alguns agregadores de log: **Kibana**, **PaperTrail**, **Spunk**...

Neste curso, usaremos o PaperTrail (que é bem simples e é online).

<https://www.papertrail.com/>

Para usá-lo, basta ir na documentação:

Criar um logback.xml (no Resources) e colocar as configs.

Porém, para cada usuário, ficaria difícil de rastrear isso, para isso, usaremos o **Spring Sleuth**.  
Usando o trace ID de cada requisição (correlation id).

### RESUMO:

Eureka - Service discovery e register;  
Spring Feign (e Ribbon) - Client Side loadBalance;  
Spring Config Server - Configura os microserviços;  
PaperTrail e Spring Sleuth - Agregador de logs;