Curso Alura SpringCloud (Microservices)

Aula 1

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

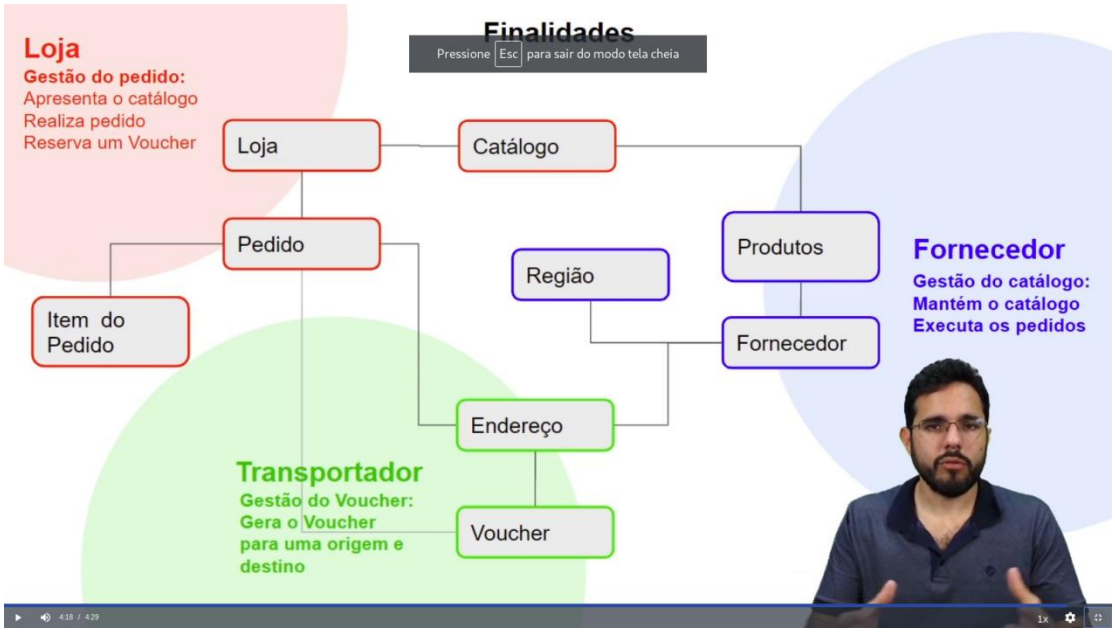
-> vou usar ultima versao de tudo com mongoDB

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

* 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 initialzr 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).

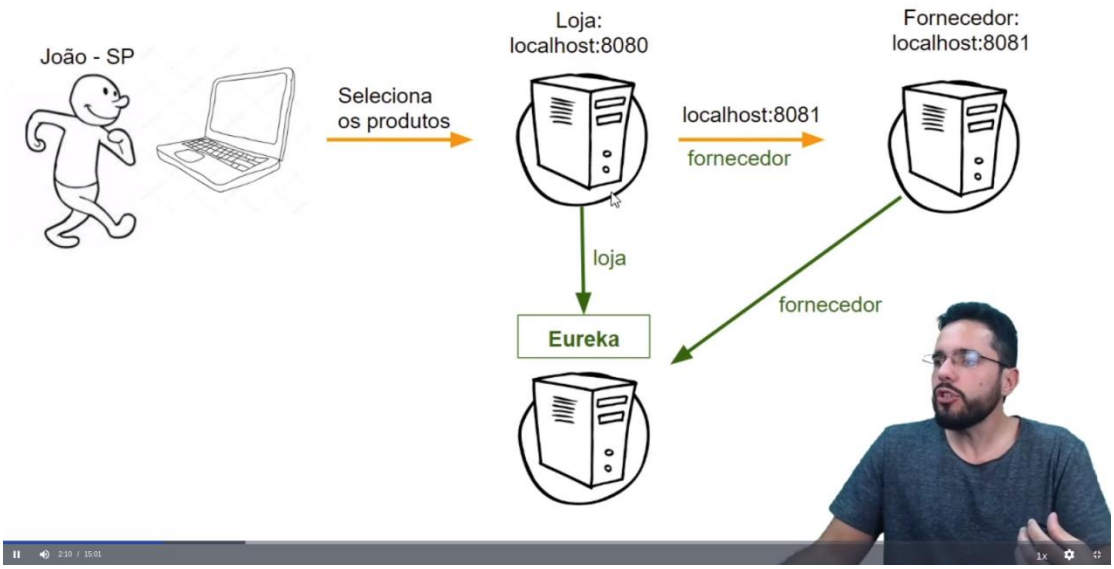
Ex.: RestTemplate client = new RestTemplate();
Client.exchange("http://localhost:**8081**/info", HttpMethod.GET, null, InfoDTO.class);
A porta é diferente, pois subiremos o outro microserviço em outra porta, claro.

Ao criar o segundo microserviço, usei o spring initialzr 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

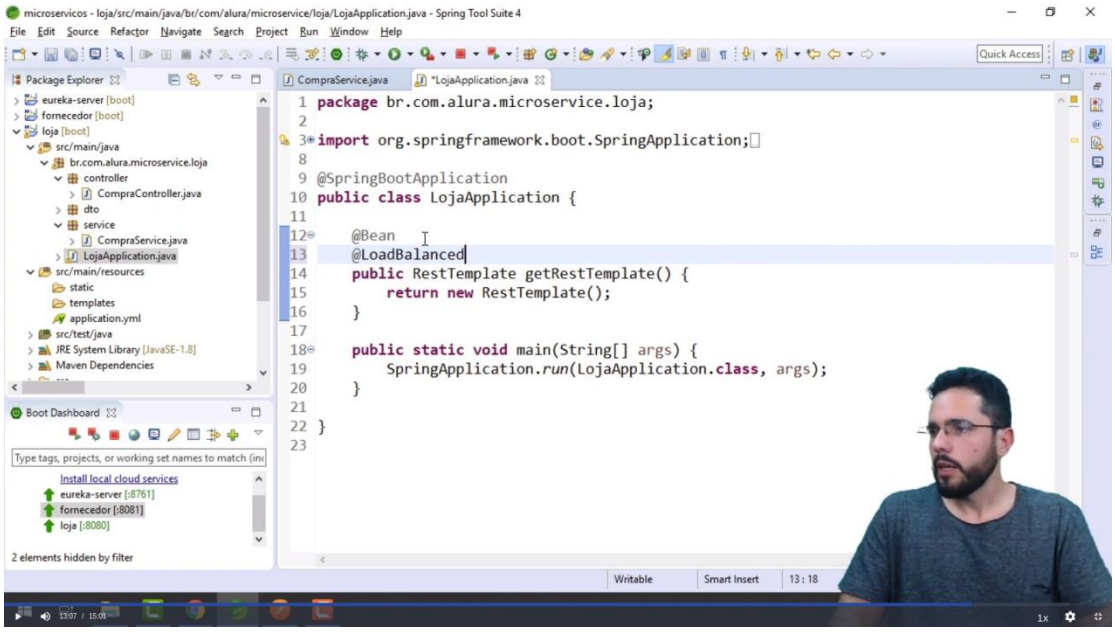
Ele nos lista **todas** as aplicações que se encontram registradas no eureka.

Para adicionarmos apps no eureka, passamos via **application.yml** as configuracoes 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 botao direito no projeto e selecionar spring > edit starter e colocar a dependencia 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 inteligencia 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.



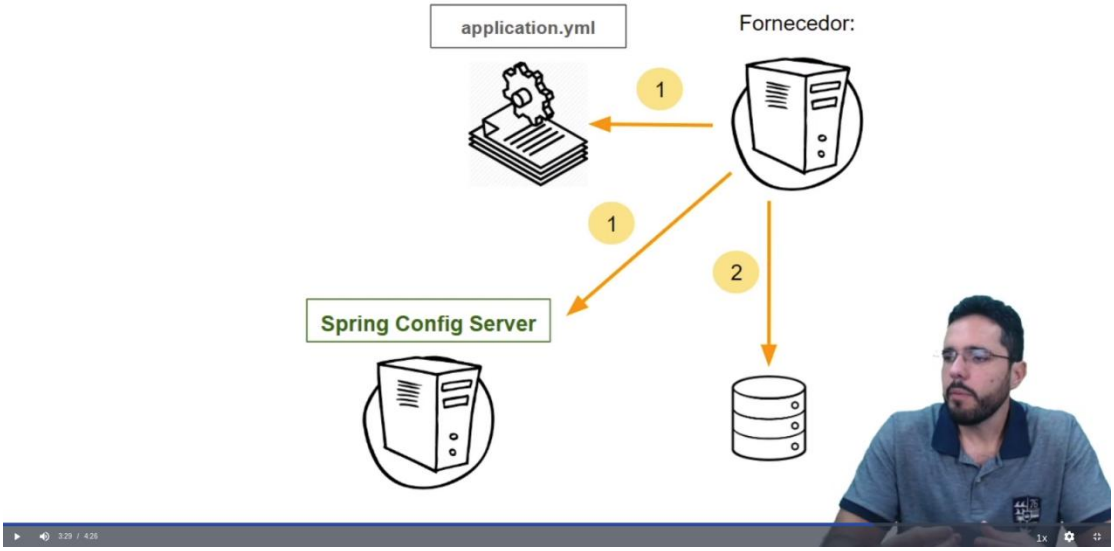
Resumindo:

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:

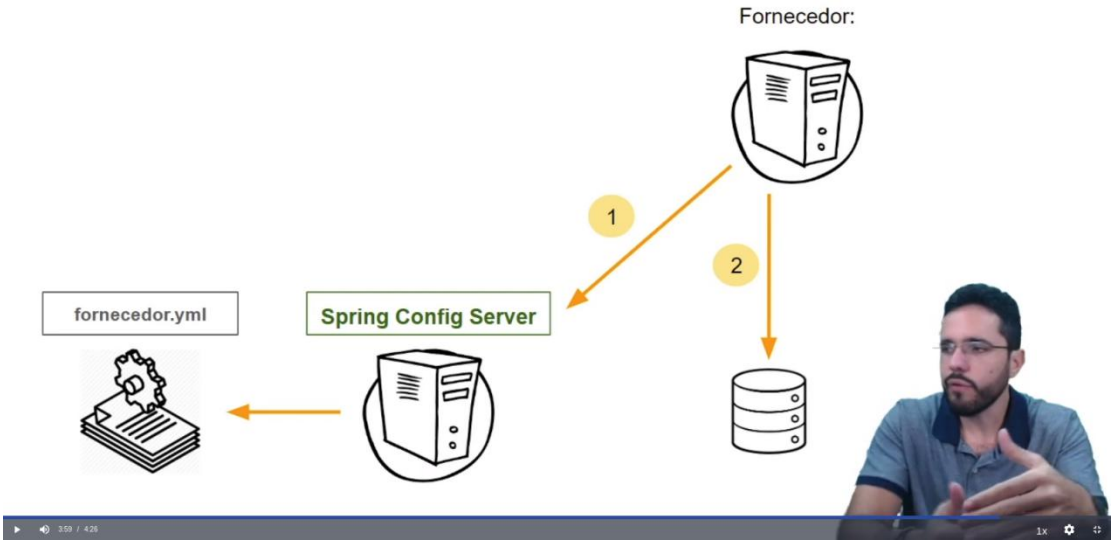
* Configuração de ambiente usando o **Spring Config Server**. É muito trabalhoso configurarmos cada microserviço para utilizar usuario/senha e BD no application.yml, etc, etc.P ara isso, utilizaremos o **Spring Config Server do pacote Spring Cloud.**

“ Os microsserviços 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 microsserviços 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.”

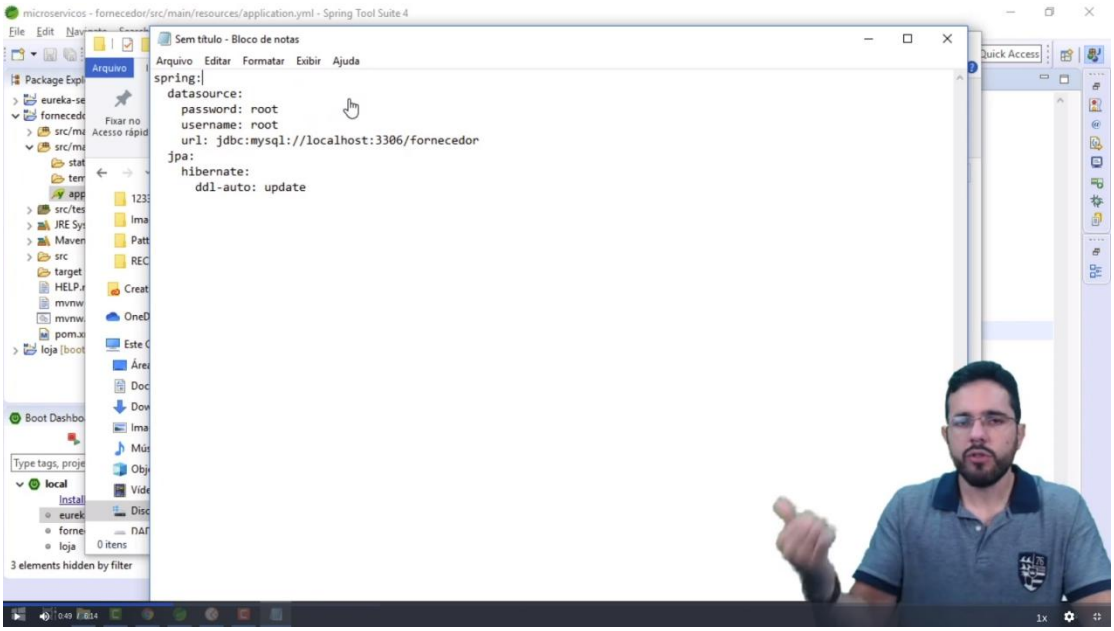


Logo, precisamos extrair os valores de configurações para salvarmos em um repositório que será utilizado pelo Spring Config Server.

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

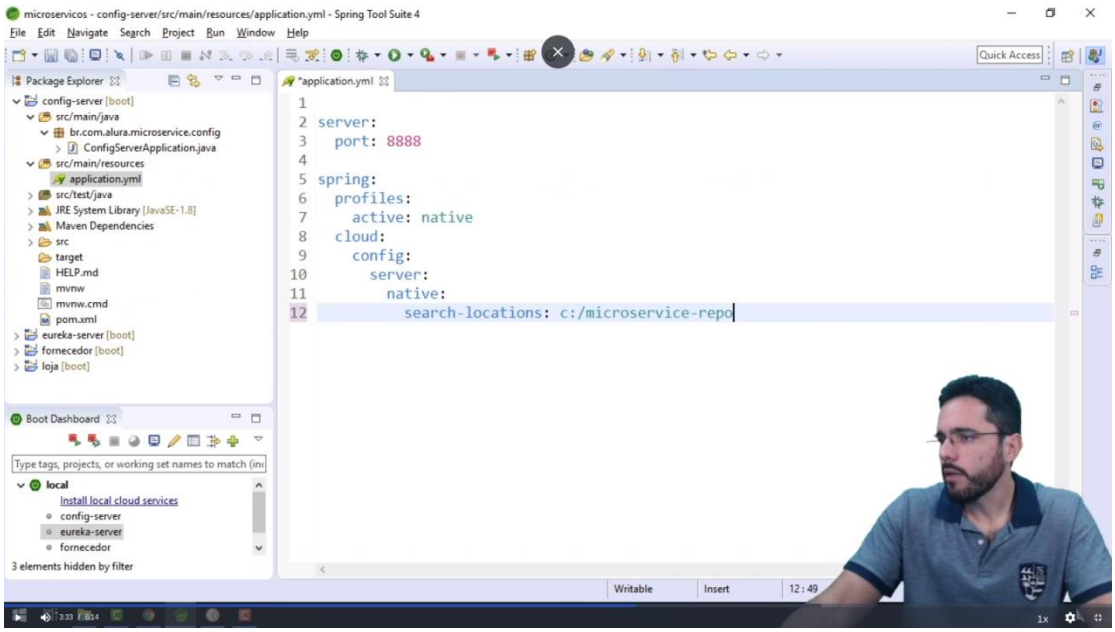


Criamos um “fornecedor.yml” em algum lugar do nosso sistema (system config, etc) e ensinamos o Spring Config Server qual o path para obter esses dados. (Nesse caso, as configurações que serão utilizados pelo serviço de fornecedor).
Da seguinte forma:



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

1. Colocar a anotacao **@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</artifactI
d>
   <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)

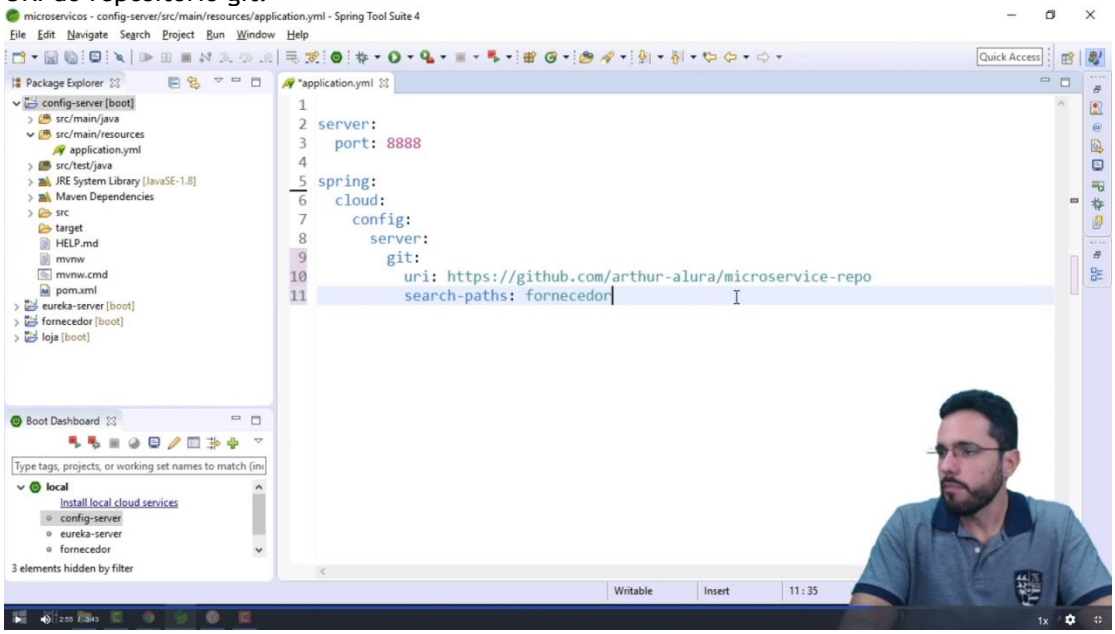
```
<properties>
   <java.version>1.8</java.version>

<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 repositorios;
Todo: Como encriptar o password??
Todo: Como usar o SHA key para configurar um repositorio de configs no git com o Spring Cloud Config?

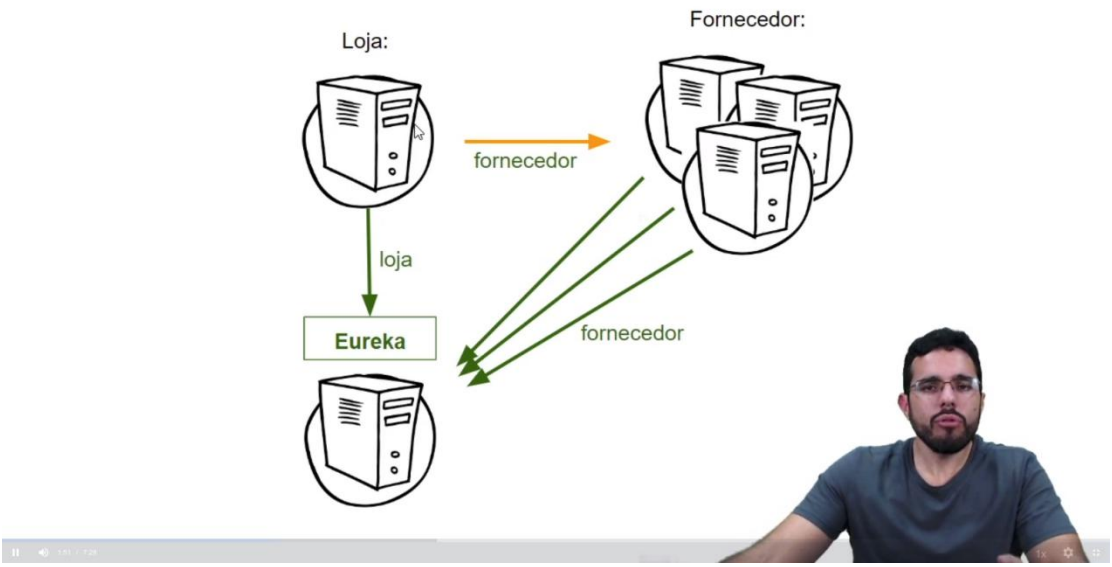
Podemos também criar o repositorio de arquivos no github. Basta criarmos um projeto no github com as configurações.

Com isso, podemos ir ao Config Server (application.yml) e trocar o **native** para **git** e passar a URI do repositorio 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 disponiveis 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 dificil de trackeá-los.

Existem alguns agredadores 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.

Porem, para cada usuario, ficaria dificil 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;

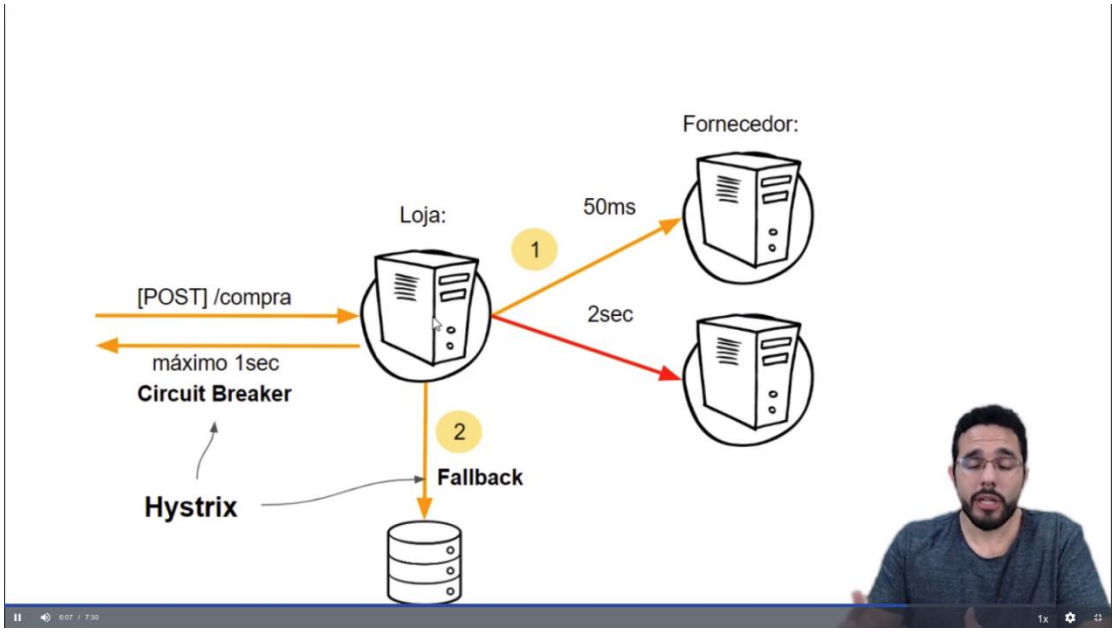
SEGUNDO CURSO (Avançado):

Microserviços com problemas (demora nas requisições, etc).

Resolução: monitorar as requisições e/ou definir um tempo limite (time-out) para a requisição.

Ou seja, se a requisição demorar > um tempo x, podemos cortar a requisição e retornar um erro 500 para o usuário. (errar rápido -> circuit breaker).
Ou entao, retornar um erro diferente, como por exemplo, cachear as informações recebidas do usuario e caso demore mais que o tempo estipulado na requisicao, podemos usar essas informações para tentar executar parte da requisição e devolver uma mensagem pro usuário. Essa técnica é chamada de fallback (um método que será chamado **depois** do tempo limite ter se passado, em caso de problemas).

E, uma lib que faz isso é o **Hystrix**, do Netflix:



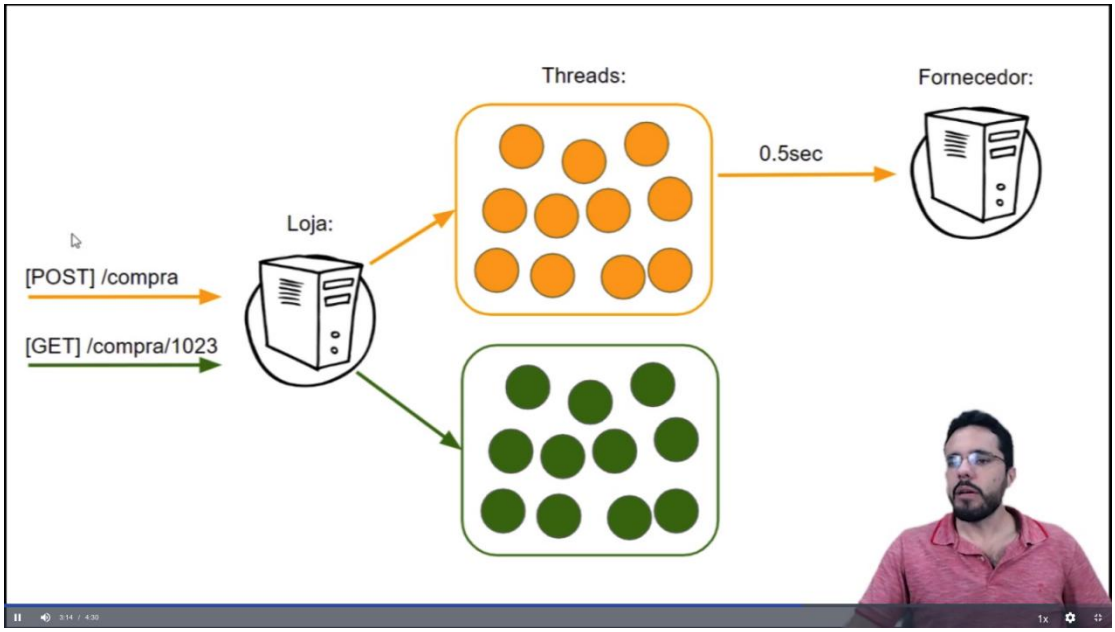
Para usarmos o **Hystrix**, temos que adicioná-lo ao projeto da loja (da mesma forma que fizemos com outras dependencias). Além disso, precisamos colocar a annotation @EnableCircuitBreaker para ativá-lo.

Para gerenciar os métodos, basta usar a annotation **@HystrixCommand.**
Podemos configurar o método de Fallback com **@HystrixCommand(fallbackmethod = "nome_do_metodo").**

* Software **JMETER** para testes de backend.

* Se as requisições demorarem muito (caírem muito nesse microserviço defeituoso), o Hystrix encaminhará direto para o Fallback, economizando tempo (e, de vez em quando, vai até a requisição para testá-la).

Caso a loja tenha TODAS as threads ocupadas com uma determinada requisição, usuários que, por ventura, tentarem fazer outra requisição, (diferente da que está travada) não vão conseguir. O ideal, é separar as threads por requisição (ou por grupo):



O nome dessa técnica é chamada de **bulkhead**.

Para isso, basta configurarmos as anotações dos métodos Hystrix com **threadPoolKey = "nome_das_threads"**.

PROBLEMA: Como configurar o mesmo cluster para ser usado mais de uma vez por vários microservices?? (No mongoDB atlas).

No MongoDB local, funciona bem!

Pode ser que o pedido foi processado, mas nao foi realizado com sucesso (ou seja, foi até o fornecedor OK, mas falhou no microserviço de transportador por algum motivo).
O cliente pode querer cancelar ou reprocessar o pedido.

Para isso, salvamos a compra no banco com diversos status (processsado, realizado, etc), à medida em que vamos obtendo sucesso na persistencia.

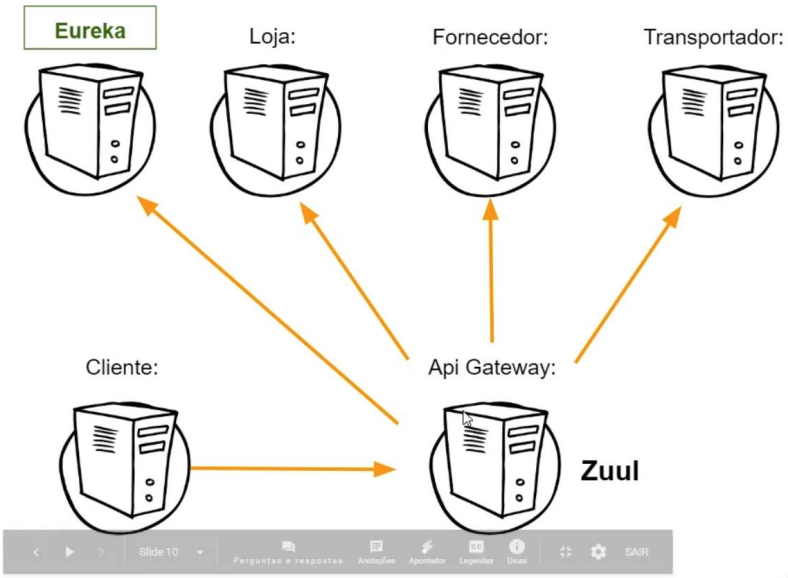
Caso dê erro, o Hystrix irá mandar para o fallbackMethod, e lá, nós devemos tratar a compra realizada (buscando o id da compra salva e retornando o estado atual da compra, e a partir disso, o cliente escolhe se quer reprocessar ou cancelar a compra, passando outros métodos HTTP para tal ação).

Poderiamos criar inclusive um **orquestrador** service, que chama o compra service, esse orquestrador é que decide o que fazer caso tenha um problema de infraestrutura na compra.

AULA 04: (Zuul)

Zuul é usado como um proxy e, a partir dele, as requisições são redirecionadas para os devidos microserviços.
Ele que faz a comunicação com o Eureka (para saber os microserviços disponiveis) e também faz o Client Side LoadBalance (usando o Ribbon já configurado).
* OBS.: Quem precisa do API gateway é o CLIENTE e não os microserviços.

“Uma aplicação rodando no navegador, ou mesmo em um aplicativo móvel, não deveria ter a inteligência de se comunicar com o Eureka, nosso Service Discovery, para descobrir as instâncias disponíveis. Aliás, faz sentido expor o Eureka na internet? Acho que não.”



Para vermos os microserviços que o zuul encontrou, usamos **localhost:<portaZuul>/actuator/routes.**

Logo,podemos acessar todos os endpoints que estávamos usando antes do zuul, ex:

Localhost:8080/compras -> localhost:<porta_zuul>/compras

Podemos acessar a documentação no spring io - Zuul

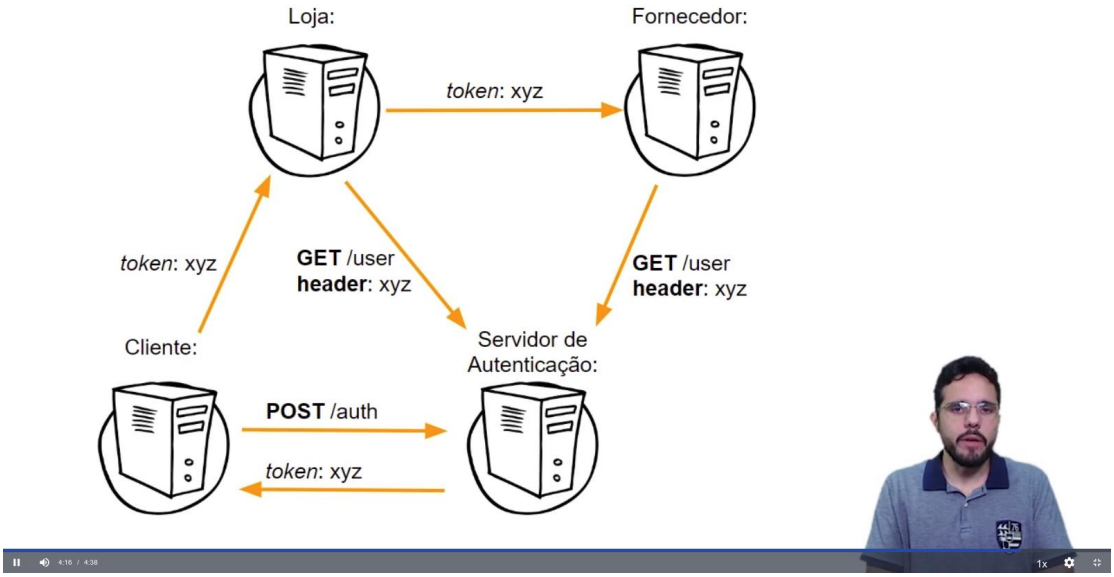
Inclusive o Zuul tem filtros que sao feitos antes e depois das requisições acontecerem.

AULA 05 - Auth

O servidor de autenticação apenas faz a **autenticação** propriamente dita.

A parte de validação e permissão daquele usuário fica à cargo do **microserviço** que o usuário está querendo processar.

Caso o serviço de loja necessite do serviço de fornecedor para efetuar uma compra (ou qualquer que seja o endpoint), é necessário passar o token para o serviço de fornecedor e ele também faz a validação do token passado.



Criamos o projeto com as dependencias: Spring web, spring security e OAuth 2 cloud.

Com isso, editamos o application com as annotations:
@EnableAuthorizationServer
@EnableResourceServer

Após isso, precisamos juntar o Security com o Oauth usando os adapters deles.

Para juntá-los, precisamos criar um configurer.

OBS.: O usuário se autentica através de uma **aplicação ou seja, primeiramente a requisição “bate” na aplicação (seja na loja, ou algo do tipo) e a loja repassa as infos para o servidor de autenticação, com isso, precisamos que a loja se identifique para o servidor de auth.(Basta criar um metodo configurer, mas com o clients e nao com os endpoints);

Ou seja> primeiramente passamos os dados da **loja** (aplicação) para autenticá-la no servidor de auth e a partir disso, autenticar o usuario.

(Por algum motivo, nao estou conseguindo fazer o cadastro da loja no servidor de autenticacao).

*Devemos fazer isso nos outros serviços, como no de fornecedor, para cadastrar a aplicação no servidor de autenticacao.

Além disso, temos que configurar o acesso do usuário dentro da aplicação em questão. (No caso, a fornecedor).

Problema: O Zuul (API-Gateway: que expõe nossa aplicação na web), não repassa os headers presentes na nossa requisição, por isso, ao tetarmos passar uma requisição que necessita de autenticação, dará erro. Para resolver isso, basta alterarmos o application do serviço. **Zuul.sensitive-headers** (com os headers que queremos, e.g: Cookie, Authorization, etc).

Problema: Depois que configuramos para a loja usar o Zuul para fazer uma requisição de compra, ela não está repassando o token de autenticação para o outro microserviço (no caso, o Pedido). Precisamos fazer com que a loja tenha essa inteligência.

A autenticação feita pelo Feign Client não repassa o token de um microserviço para outro.

Para fazer isso, o Feign aceita instâncias de **RequestInterceptor**, logo, isso será injetado diretamente na requisição.

Problema: Ao criar o um Interceptor do Feign, o Hystrix está gerenciando nossas pools de threads, logo, ao chamar o método realizaCompra, como são executadas em threads diferentes, a informação de autenticação não é compartilhada. Para isso, precisamos ir na configuração do projeto (application.yml) e usar uma configuração do Hystrix: **hystrix.shareSecurityContext** como true.