

# Orquestrando Microserviços com Spring Cloud e Netflix OSS

Por: Jonathas Lima

## Introdução

Neste artigo, irei tentar abordar um conteúdo introdutório, porém didático e explorativo sobre alguns assuntos que podemos abordar na orquestração e comunicação entre Microserviços utilizando Spring Cloud Config que possibilita a criação de uma configuração externalizada em um sistema distribuído.

Para isso, temos primeiramente que configurar um Server Config, mas o que é isso? Nada mais do que centralizar a configuração de seus microservices em um repositório remoto ou local, para que possamos ter uma gestão sobre os mesmos.

Que tipo de configurações poderemos ter no nosso ambiente server config? Podemos ter desde execução de portas onde os serviços irão ser descobertos como configurações de banco de dados da aplicação, basicamente, o que você desejar, filas, threads etc.

Após a criação de nosso Server Config, iremos continuar na stack Spring Cloud e conjuntos com o projeto Netflix OSS , e desvendar um pouco mais sobre Service Discovery (Eureka) e técnicas de Circuit Breaker (Hystrix/Feign), além disso, iremos ver como fornecer um token válido para autenticação utilizando JWT (Json Web Token).

## Vamos ao nosso Service Config.

Apesar do nome ser um pouco assustador, nossa, vamos criar um servidor...

Calma, iremos começar por acessar o nosso conhecido portal **Spring Initializr** (<https://start.spring.io/>) e criar uma aplicação que será nosso servidor de configuração.

Você deve estar se perguntando, iremos criar um Microserviço? Um Spring Boot? SIM..iremos.

Ao acessar, a interface é bem simples, e intuitiva , devemos apenas nos atentar para adicionarmos as seguintes dependências ao nosso projeto:

**web (spring-boot-starter-web)**

**Actuator (spring-boot-starter-actuator)**

**Config Server (spring-cloud-config-server)**

Explicando um pouco sobre as dependências acima temos:

**Web** – Dependência responsável por nossa interação com serviços web, como apis's restfull, tomcat embedded entre outras mais, talvez a dependência mais importante dos projetos utilizando microserviços.

**Actuator** – Dependência que irá realizar medições e dados estatísticos da saúde do seu microserviço, como por exemplo, se o mesmo encontra-se UP ou Down.

**Config Server** – Aqui onde temos a possibilidade de transformar nosso Microserviço em um ambiente server.

Vamos fazer uma pausa, pois é importante neste momento, conhecer 2 arquivos que iremos utilizar bastante ao longo do artigo, o application.yml e o bootstrap.yml

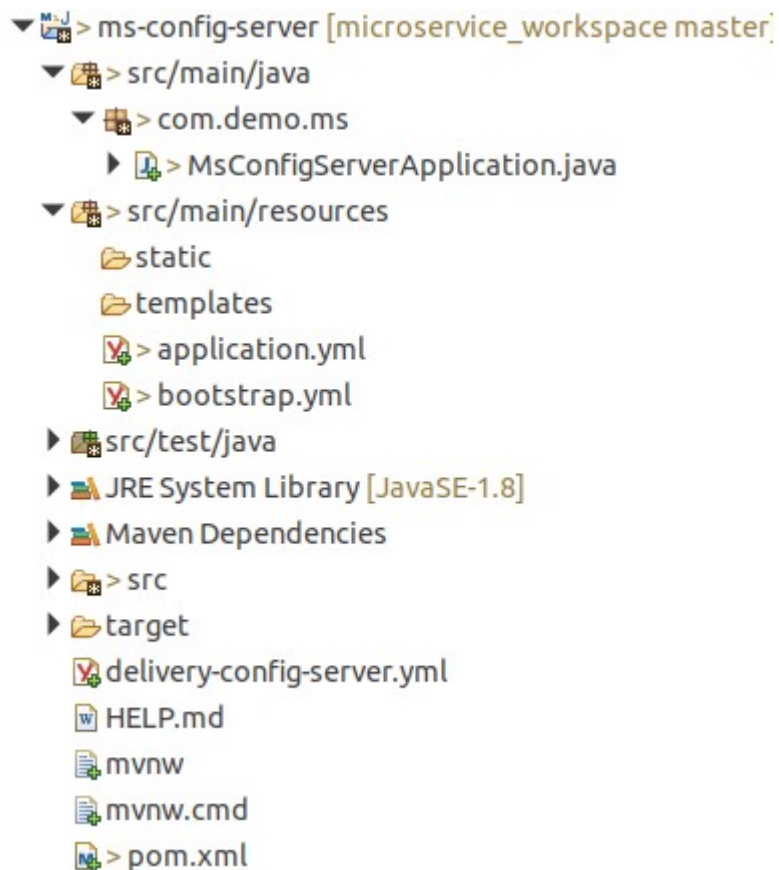
Ao gerarmos nosso projeto, e importarmos o mesmo em nossa ferramenta de trabalho, podemos visualizar um folder chamado resource, dentro deste pacote podemos realizar diversas configurações, através exatamente do arquivo application.yml e bootstrap.yml, mas qual a real função de ambos:

**application.yml ou application.properties** – É o arquivo responsável por substituir as configurações no momento de dar o start no nosso spring boot, como por exemplo substituir porta , declarar conexões com banco de dados, informar acesso a novos recursos como queues etc.

Na estrutura do spring boot, podemos ter essas duas extensões de arquivos, .yml e .properties, ambos realizam a mesma funcionalidade, porém prefiro usar yml pela legibilidade das declarações nos arquivos.

**Bootstrap.yml** - Esse arquivo é usado para realizar algumas configurações de inicialização, conexões com ambientes remotos etc.

Voltando ao nosso projeto, depois de configurado nossas dependências e nossos arquivos, teremos a seguinte estrutura.



## Application.yml

```
Y application.yml X
1 server:
2   port: 9090
3
4 spring:
5   cloud:
6     config:
7       server:
8         git:
9           uri: https://github.com/devjolima/ms-configs
10
```

No Arquivo `application.yml` temos a definição de qual porta a aplicação irá ser “startada”, no caso 9090 e temos também a definição de onde nosso server irá buscar as informações dos nossos Microserviços, vamos ver um pouco mais a frente que, todas as configurações serão dependentes deste endereçamento , no meu caso, aponte para um repositório no meu github, através da informação `spring.cloud.config.server.git.uri`, como estamos em um arquivo `yml`, essa informação encontra-se formatada de acordo com o padrão `yml`, se tivéssemos a mesma informação em um `applications.properties`, seria da forma como esta escrita, com separação em pontos.

Mas qual seriam as informações que serão buscadas no repositório remoto? Bem, diversos tipos de configurações, iremos ao longo do artigo, identificá-las juntos, de inicio, vamos criar um arquivo com o mesmo nome de nosso Microserviço no caso será o `MS-CONFIG-SERVER.yml`, o meu arquivo já esta no meu repositório, sugiro neste momento, criar um repositório remoto, publico, e subir seu arquivo de configuração para o mesmo.

Neste momento, suba apenas uma informação simples, contendo apenas o trecho abaixo.

```
4 lines (3 sloc) | 41 Bytes
1  delivery:
2    maxOrders: 10
3    minOrders: 1
```

A seguir nossa configuração do `bootstrap.yml` e nossa classe que irá executar o Microserviço.

### Bootstrap.yml

```
1  spring:
2    application:
3      name: ms-config-server
4
```

No nosso arquivo `bootstrap.yml`, iremos apenas carregar o nome de nossa aplicação , isso será bastante útil, pois é através desta nomenclatura que o nosso micro-serviço será identificado no nosso Service Discovery mais a frente.

Com esta configuração finalizada, temos nosso micro-serviço configurado como um server, bastando apenas fazer uso de nossa anotação , disponibilizada pela dependência `config server`, na nossa classe de inicialização do boot.

```

1 package com.demo.ms;
2
3 import org.springframework.boot.SpringApplication;
4
5
6
7 @SpringBootApplication
8 @EnableConfigServer
9 public class MsConfigServerApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(MsConfigServerApplication.class, args);
13     }
14
15 }
16

```

Através da anotação `@EnableConfigServer`, quando essa classe for executada, o Spring irá identificar esta anotação e imediatamente atribuir o nosso micro-serviço a um server que irá ficar monitorando todas as requisições para os demais micro-serviço, mais a frente vamos entender como isso será possível.

Pronto pessoal, finalizado nosso **Server Config**, agora no próximo passo, vamos implementar como descobrir esse serviço dentro de nossa estrutura de cloud, para isso, vamos utilizar o Eureka, um Service Discovery que vai atuar no monitoramento de nossos micro-serviços.

Ao acessar o serviço <http://localhost:9090/ms-config-server/default>, deverá receber a seguinte resposta:

```

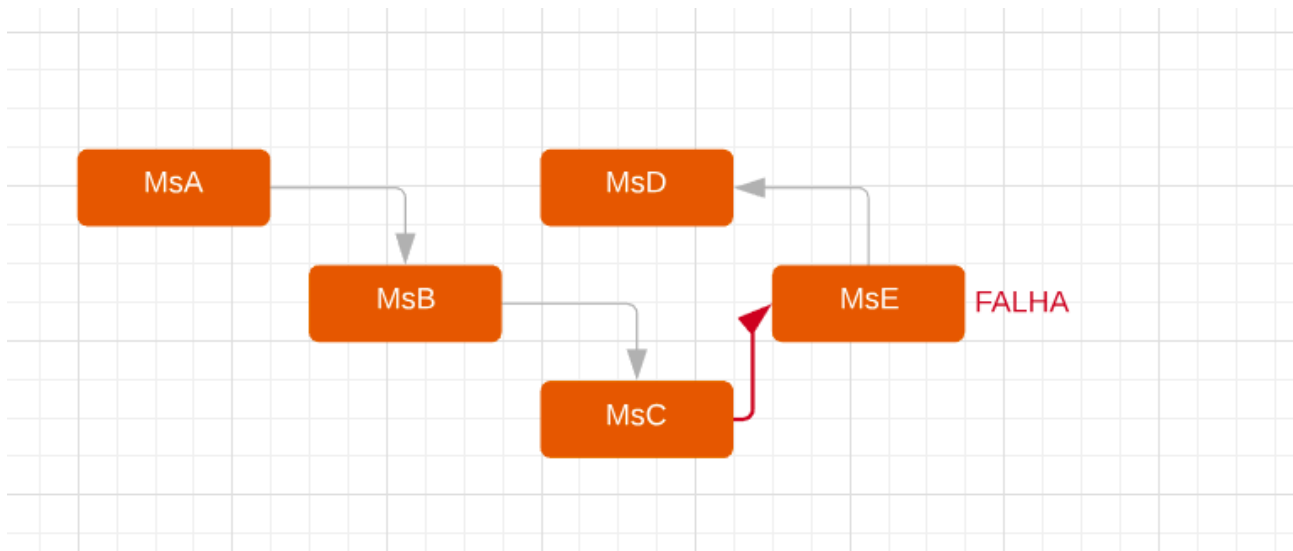
▼ object {6}
  name : ms-config-server
  ▼ profiles [1]
    0 : default
  label : null
  version : 45448db0268664aa20403c5f7d76d79bc078596c
  state : null
  ▼ propertySources [1]
    ▼ 0 {2}
      name : https://github.com/devjolima/ms-configs/ms-config-server.yml
      ▼ source {2}
        delivery.maxOrders : 10
        delivery.minOrders : 1

```

Perceba que sua declaração de `maxOrders` e `minOrders` esta devolvida no json de resposta, com isso, seu arquivo esta sendo monitorado perfeitamente pelo config server. Isto é possível devido nossa dependência do actuator, que devolve o status do serviço.

## O Eureka ...

O que é o Eureka, Eureka é um framework disponibilizado pela stack Spring Cloud, que irá realizar o trabalho do Service Discovery, para clarear as ideias, vamos ver a figurinha abaixo, e entender bem o que seria um discovery dentro de um cenário de ambientes distribuídos.



Vamos imaginar o cenário, que o MsE (Microserviço E), encontra-se DOWN, ou seja, sem comunicação, como saberemos deste problema? E sabendo do problema, como reagir? (Aí é outra história, já veremos)

Vamos a mão na massa, vamos montar o micro-serviço para subirmos um cliente do Eureka e começar a monitorar nossos serviços que estão no radar..

Mesmo passo anterior, vamos entrar no site Spring Initializr e seguir os passos para a criação do nosso micro-serviço, PORÉM, existem novas dependências que iremos conhecer neste momento.

Nossas dependências para este serviço serão as seguintes:

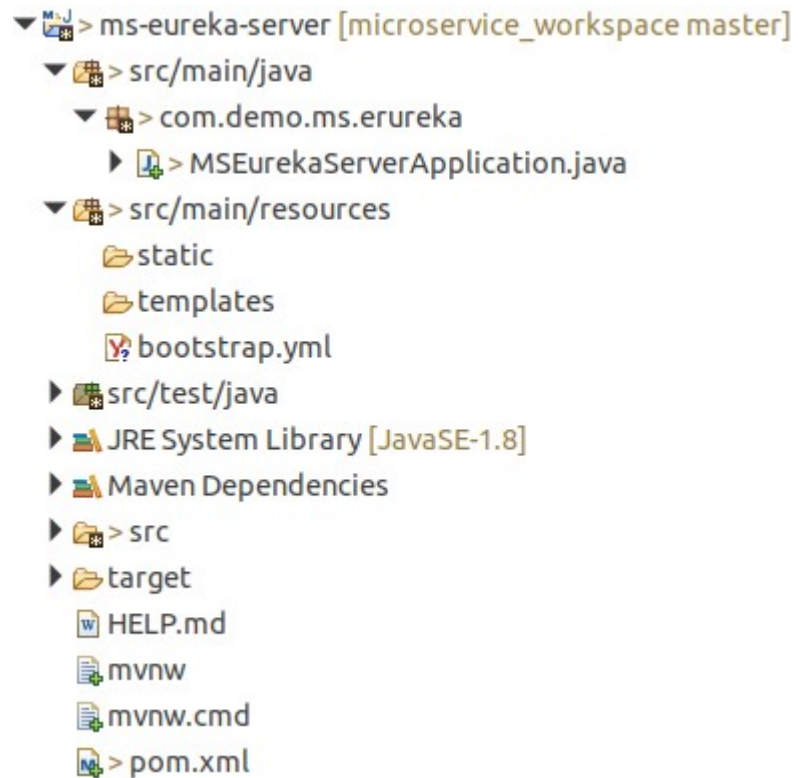
spring-boot-starter-actuator – Já conhecemos de nosso projeto anterior.

spring-boot-starter-web – Já conhecemos de nosso projeto anterior.

spring-cloud-starter-config – Já conhecemos de nosso projeto anterior.

**spring-cloud-starter-netflix-eureka-server** – Aqui temos uma novidade, esta dependência permite que utilizemos nosso framework Eureka no nosso projeto, um pequeno cuidado, pois o site de dependências do Maven, disponibiliza versões desta dependência ainda no pacote antigo, desta forma, spring-cloud-starter-eureka-server, PORÉM, como estamos utilizando a stack do Netflix OSS, o correto, é utilizar da forma como esta descrito.

Após a criação do seu projeto, sua estrutura deve ter ficado da seguinte forma



Perceba que trata-se da mesma estrutura anterior, pode ser que surja uma dúvida , ah..mas meu discovery também é um micro serviço? Sim, ele é, e vamos ver um pouco mais a frente o porque.

Voltando a estrutura do projeto , como já sabemos, temos que configurar nosso bootstrap.yml, pois ele é quem direciona nosso classloader digamos assim, e nos informa detalhes de infra estrutura e condições específicas do projeto, porém, vamos ver abaixo como fica o nosso arquivo, preste atenção neste ponto.

```
1 spring:
2   application:
3     name: ms-eureka-server
4   cloud:
5     config:
6       uri: http://localhost:9090
```

A propriedade spring.application.name é conhecida nossa, trata-se apenas de como o nome vai ser EXPOSTO no nosso service discovery, mas vejamos a propriedade cloud.config.uri, quem é ela?

Ela é o endereço de nosso service config, que esta exposto na porta 9090 lembra? Aqui esta o pulo do gato, ou seja, como temos uma estrutura distribuída, todos os micro-serviços terão que se reportar a um server, o pai de todos, e ele saberá redirecionar corretamente a requisição, esse é o papel do service config, centralizar informações e saber gerenciar para quem é responsável a requisição, nesse

nosso caso, quando o server config receber uma informação nele, ele vai simplesmente olhar OPA, quem é o nome do serviço que me chama, e delegar para a configuração do mesmo... simples não é?

Ah Jonathas, mas qual é a configuração do ms-eureka-config?? Ainda não criamos, vamos fazê-lo agora.

```
server:
  port: 9091

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: true
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
  server:
    wait-time-in-ms-when-sync-empty: 3000
```

Vamos analisar as informações, nossa primeira informação server.port , ate ai tudo bem, esta me dizendo que meu Eureka irá subir nesta porta.

Eureka.instance.hostname, esta apontando para localhost, ou seja, localhost:9091 irá ser meu dashboard do Eureka.

Muito importante nesse momento essas duas propriedades cliente.registerWithEureka e fetchRegistry

Vamos a elas:

**eureka.client.registerWithEureka :** Mesmo sendo um Servidor do Eureka, essa aplicação não deixa de ser um Microservice como os outros, portanto ele deve se auto registrar no eureka, o registerWithEureka informa que a aplicação irá ser reconhecida pelo mesmo.

**eureka.client.fetchRegistry:** Os clientes usam essa informação para encontrar outros serviços. Como essa aplicação é o próprio Servidor do Eureka, desabilitamos esta busca, mas em caso de criamos outros serviços, que iremos fazer a seguir, essa informação deve ser true, senão não é capaz de propagar os micro-serviços.

eureka.client.serviceUrl.defaultZone: É a referencia do eureka, todos os demais serviços que o eureka irá se registrar, deve ter essa propriedade.

**eureka.server.wait-time-in-ms-when-sync-empty:** Tempo em milissegundos que o Eureka Server irá esperar entre as sincronizações com os clientes.

OBS: LEMBRE-SE ESSE ARQUIVO DEVE ESTAR NO SEU GIT, OU REPOSITÓRIO REMOTO DE SUA PREFERENCIA, JUNTAMENTE COM O DEMAIS QUE VOCÊ INFORMOU AQUI:



```

application.yml
1 server:
2   port: 9090
3
4 spring:
5   cloud:
6     config:
7       server:
8         git:
9           uri: https://github.com/devjolina/ms-configs
10

```

Simples não? No início parece um pouco confuso, mas resumindo em um cenário mais intuitivo, pense como um hub, onde requisições neste hub são delegadas para seus donos, nosso hub seria o server config, o Eureka é um orquestrador dessas informações, apresentando quem está UP, Down e diversas outras funcionalidades e informações.

Seguindo com o nosso artigo, falta apenas a classe principal do spring boot

```

@SpringBootApplication
@EnableEurekaServer
public class MSEurekaServerApplication {


    public static void main(String[] args) {
        SpringApplication.run(MSEurekaServerApplication.class, args);
    }

}

```

Vamos atentar apenas para a anotação `@EnableEurekaServer`, ela que faz a nossa magia e sobre nosso servidor Eureka.

Acessando o endereço <http://localhost:9091> teremos a seguinte tela.


HOME
LAST 1000 SINCE STARTUP

### System Status

Environment	test	Current time	2019-08-23T18:36:04 -0300
Data center	default	Uptime	00:12
		Lease expiration enabled	true
		Renews threshold	1
		Renews (last min)	6

### DS Replicas

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MS-AUTH-SERVER	n/a (1)	(1)	UP (1) - 192.168.15.60:ms-auth-server:9092
MS-BUSINESS-TEST	n/a (1)	(1)	UP (1) - 192.168.15.60:ms-business-test:9093
MS-EUREKA-SERVER	n/a (1)	(1)	UP (1) - 192.168.15.60:ms-eureka-server:9091



Lindo não? Sim :-). ..perceba que, no meu Eureka tem alguns micro-serviços declarados, mas no seu é para aparecer apenas o EUREKA neste momento.

Muita informação até aqui, e neste momento, sabemos o que é um Service Discovery.

Tudo bem vamos SEGUIR ...

Neste momento, eu criei um micro-serviço apenas para interagirmos com ele, você pode criar o seu, com o conteúdo de sua preferência, tenha apenas em suas ideias que precisamos de um micro-serviço que te forneça alguma informação, o meu, criei um micro-serviço que irá retornar um Token, com JWT, que logo mais, outros serviços irão realizar uma requisição, em um caso teremos sucesso, o outro falha, e aí é onde vamos conhecer o Hystrix, o famoso conceito de Circuit Breaker, que irá tratar nossas falhas.

O nome do meu micro-serviço é ms-auth-server, como objetivo principal, é fornecer um token para o uso de requisições REST, o mesmo pode ser encontrado no meu repositório em (<https://github.com/devjolima/microservicos.git>)

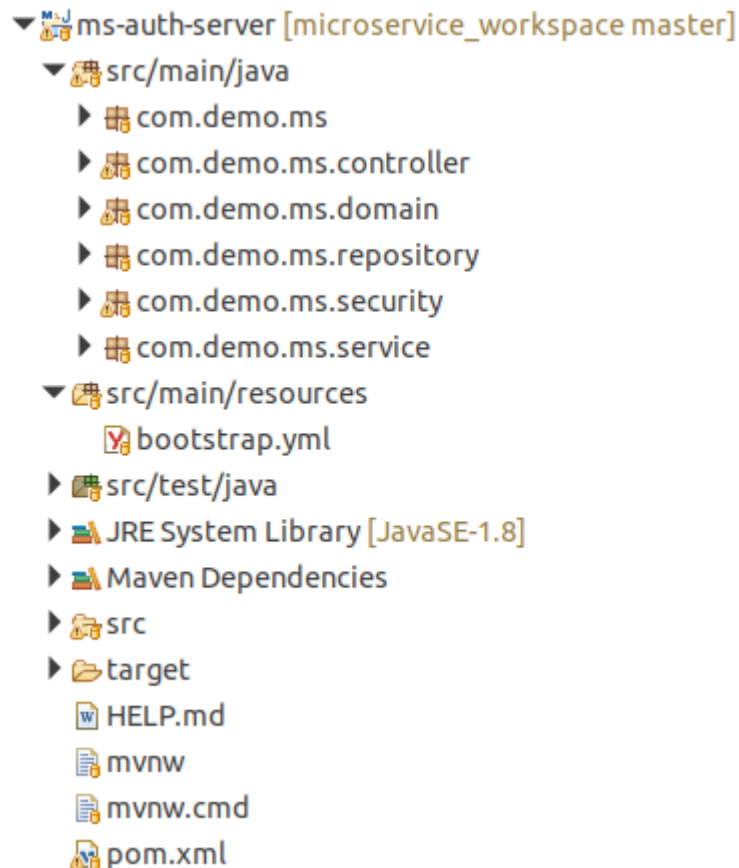
Suas dependências já são conhecidas nossas, segue:

Desta vez, irei por um print e pedir uma observação para com relação a apenas a spring-security-config, esta dependência habilita a parte de segurança da aplicação, então quando o micro-serviço for iniciado, uma validação de chave deve ocorrer, mas como trata-se apenas de um artigo educacional, teremos uma opção para desabilitarmos, que irei mostrar no momento.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2.1.2.RELEASE</version>
  </dependency>

  <!-- JWT -->
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.7.0</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework.security/spring-security-config -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.1.6.RELEASE</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework.security/spring-security-web -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.1.6.RELEASE</version>
  </dependency>
</dependencies>
```

Nossa estrutura do projeto ficou desta forma:



Eu criei esse micro-serviço com uma estrutura de pacotes, pois a ideia é evoluir para ficar algo que eu reuse em meus projetos, sua estrutura é bem simples, dentro do pacote controller, temos alguns endpoints que irão trabalhar para realizar a requisição do nosso token, e a partir daí, seguir o caminho que precisamos, qualquer dúvida vocês podem me enviar um e-mail que tento ajudar.

[Jonathas.o.lima@gmail.com](mailto:Jonathas.o.lima@gmail.com)

Como em todos os passos temos o nosso bootstrap.yml, o nosso ficou assim:

```
1 spring:
2   application:
3     name: ms-auth-server
4   cloud:
5     config:
6       uri: http://localhost:9090
7
8
```

Como vimos em etapas mais atrás do nosso artigo, nome é o nome que será registrado no nosso Eureka, e a uri, que aponta para o nosso HUB, que é o server config.

Estamos ESQUECENDO DE ALGO??? SIMMMMMMMMM

Vamos lembrar de nosso ms-auth-server.yml, que precisamos enviar para nosso repositório remoto, com o objetivo de o server config delegar sua inicialização.

```
1  server:
2    port: 9092
3
4  eureka:
5    instance:
6      hostname: localhost
7      port: 9091
8    client:
9      registerWithEureka: true
10     fetchRegistry: true
11     serviceUrl:
12       defaultZone: http://${eureka.instance.hostname}:${eureka.instance.port}/eureka/
13   server:
14     wait-time-in-ms-when-sync-empty: 3000
15
16  spring:
17    datasource:
18      driver-class-name: org.postgresql.Driver
19      password: '12345'
20      platform: postgres
21      url: jdbc:postgresql://localhost:5432/postgres
22      username: postgres
23    jpa:
24      hibernate:
25        ddl-auto: create
26        show-sql: true
27        database: postgresql
28        database-platform: org.hibernate.dialect.PostgreSQL9Dialect
```

Pessoal, neste arquivo, tenho apenas 2 observações, visto que já conhecemos essa estrutura, a primeira é fetchRegistry, queremos que esse serviço seja visto pelo Eureka, então, declaremos isso com TRUE.

E na sequência do arquivo, temos aí o carregamento de informações de banco de dados, como estamos trabalhando com cloud, essa configuração fica aqui também, mas poderíamos declarar isso dentro do projeto junto do nosso application.yml.

Sugiro para quem estiver seguindo um passo a passo do artigo, olhar o projeto no git, e tentar entender um pouco sobre o JWT, o objetivo deste artigo não é ele, posso depois estar realizando um artigo somente sobre ele, hoje, estamos abordando Service Discovery, Circuit Breaker e Feign client.

Mas quero ilustrar com esse endpoint no meu restcontroller

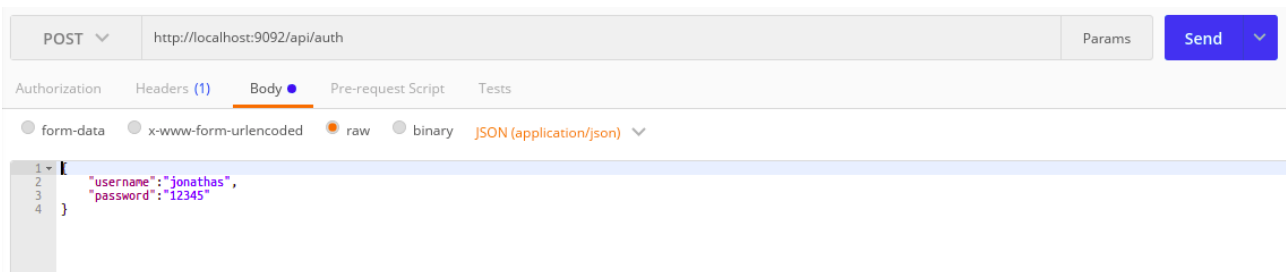
```
@Autowired
private JwtTokenProvider jwtTokenProvider;

@PostMapping("/auth")
public ResponseEntity<String> signin(@RequestBody AuthenticationRequest data) {
    String token="";
    try {
        token = jwtTokenProvider.createToken(data.getUsername(), Arrays.asList("ADMIN"));
        return ResponseEntity.ok().body(token);
    } catch (Exception e) {
        return new ResponseEntity<String>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

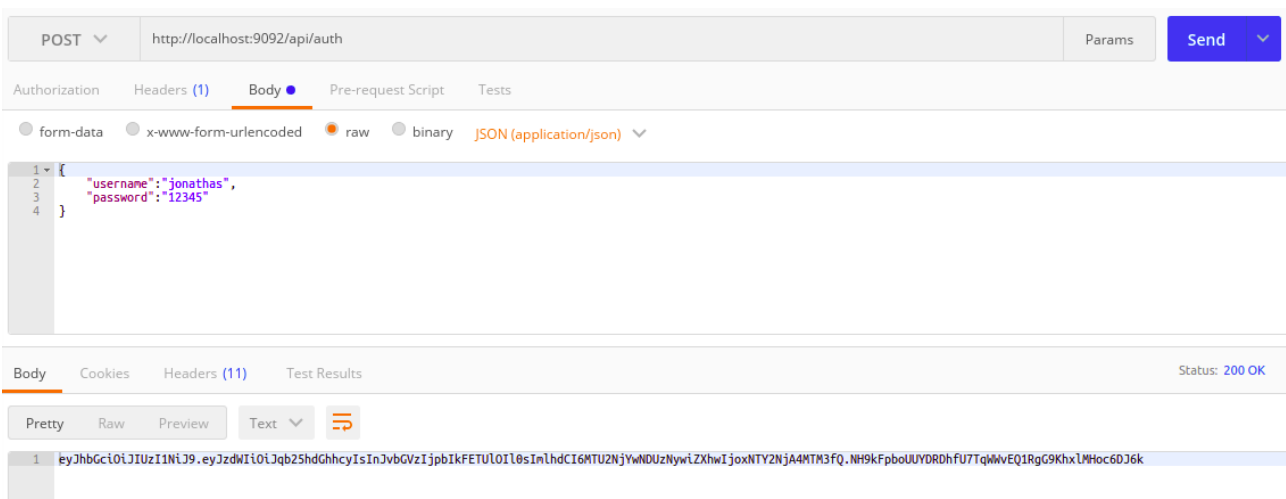
Bom, explicando rapidamente, esse endpoint irá receber um objeto do tipo `AuthenticationRequest`, que contém apenas `name` e `password`, e devolver um token assinado e validado com a role `ADMIN`.

Neste ponto, vamos iniciar nosso micro-serviço, eu declarei o mesmo na porta 9092, então quando acessarmos o dashboard do Eureka, o mesmo vai estar sendo monitorado nesta porta.

Bom, eu sempre uso o postman para testar minhas requisições, neste projeto não configurei o swagger, que nos ajudaria com esses testes. Realizando um teste com o postman, de acordo com a imagem abaixo, teremos:



E como resultado:



Ai está o meu token, devolvido na requisição, que irei logo mais usar para realizarmos nossos testes de falha.

Bom, você pode criar qualquer micro-serviço neste ponto, um endpoint que retorna uma mensagem qualquer, e que você possa simular um erro para testarmos com o Hystrix, fique a vontade e use a criatividade.

Chegou a hora do Hystrix ...

O Hystrix é mais uma solução do grupo Netflix OSS, ele implementa o padrão Circuit Breaker e visa lidar com as possíveis falhas nas chamadas entre microsserviços. O Hystrix como sua função principal é reagir de acordo com possíveis falhas de micro-serviços que estão down ou lance exceções não esperadas, mas o mesmo tem outras funcionalidades, como controle de threads, tempo limite de respostas e outras mais.

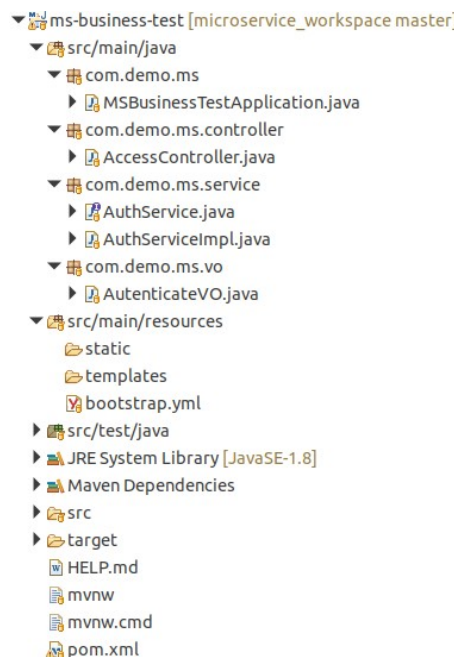
Pegando o exemplo anterior, no meu caso, o que fazer quando eu tentar acessar um método, que precise de um token valido e eu passar errado? OU, simplesmente eu não enviar um token, ou enviar um token que não é mais válido?

Bom, podemos reagir a isso de diversas maneiras, inclusive uma delas imaginando um cenário que, enviei um token, o mesmo não conseguiu validar, mas mesmo assim, eu tenho o token armazenado numa base de dados que poderá ser verificado em caso do micro-serviço principal falhar, bem existem 1001 maneiras de você tratar suas falhas, o que vou mostrar é apenas como é possível realizar esse tratamento , e fazer como que nosso serviço não seja afetado em caso de falhas de terceiros.

Estive a pouco tempo no TDC SP , e participando de algumas palestras muito interessantes, temos diversos problemas no decorrer do nosso dia a dia que não podemos prever, problema que impactam diversas áreas, saúde, divertimento, finanças, e esses tipos de falhas nem sempre sua solução é imediata, e enquanto a falha não é solucionada , nossos serviços não podem serem impactados.

Neste momento,para brincarmos um pouco com o Hystrix, eu criei um micro-serviço que irá no meu ms-auth-server, e irá tentar validar o token que eu gerei, no caso, irei forçar a dar um erro para que vejamos como o Hystrix se comporta com a falha.

Minha estrutura do projeto ficou desta forma.



```
ms-business-test [microservice_workspace master]
├── src/main/java
│   ├── com.demo.ms
│   │   └── MSBusinessTestApplication.java
│   ├── com.demo.ms.controller
│   │   └── AccessController.java
│   ├── com.demo.ms.service
│   │   ├── AuthService.java
│   │   └── AuthServiceImpl.java
│   ├── com.demo.ms.vo
│   │   └── AuthenticateVO.java
│   └── src/main/resources
│       ├── static
│       ├── templates
│       └── bootstrap.yml
├── src/test/java
├── JRE System Library [JavaSE-1.8]
├── Maven Dependencies
├── src
├── target
├── HELP.md
├── mvnw
├── mvnw.cmd
└── pom.xml
```



Vamos as dependências de nossa aplicação, neste projeto, teremos novidades

Optei mais uma vez pelo print, pois conhecemos em sua maioria estas dependências, vamos atentar para 2 novas, que serão muito importantes.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  </dependency>

  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-openfeign -->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
    <version>2.1.2.RELEASE</version>
  </dependency>
</dependencies>
```

**Spring-cloud-starter-netflix-hystrix** – Esta é nossa dependência que irá nos ajudar a fazer o trabalho de reação as falhas com o Hystrix.

**Spring-cloud-starter-openfeign** – Neste momento, apresento uma nova dependência, chamada Feign, para contextualizar melhor do que se trata, todos nós conhecemos e acessamos nossos serviços Rest com o famoso RestTemplate, que já é nativo do Spring , um Bean, e apenas necessitamos usá-lo, esta dependência é também fornecida pela stack Netflix OSS, e de forma simples , podemos realizar nossas requisições HTTP's aos nossos micro-serviços distribuídos.

Como queremos que nosso micro-serviço seja descoberto pelo Eureka, o que devemos fazer??? Isso mesmo, configurar nosso bootstrap.yml.

```
1 spring:
2   application:
3     name: ms-business-test
4   cloud:
5     config:
6       uri: http://localhost:9090
7
8   feign:
9     hystrix:
10      enabled: true
```

Nesta configuração uma informação nova, a ativação do Hystrix, devemos ativá-lo, porém nada demais, simplesmente dando true a sua propriedade.

Como desta vez não estamos esquecendo de mais nada, vamos criar nossa configuração do projeto, e enviá-la para nosso repositório não é mesmo?

Aqui nosso ms-business-test.yml:

```
1 server:
2   port: 9093
3
4 eureka:
5   instance:
6     hostname: localhost
7     port: 9091
8   client:
9     registerWithEureka: true
10    fetchRegistry: true
11    serviceUrl:
12      defaultZone: http://${eureka.instance.hostname}:${eureka.instance.port}/eureka/
13 server:
14   wait-time-in-ms-when-sync-empty: 3000
15
```

Desta vez apenas informando qual porta o micro-serviço irá iniciar, e claro, informando o Eureka para que o mesmo seja descoberto por ele com o nome que definimos no nosso bootstrap.yml.

Irei retornar um pouco na estrutura do micro-serviço e abordar 3 classes importantes para que tanto o nosso Hystrix funcione, como também o Feign.

A primeira delas nosso start do Spring Boot.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@ComponentScan("com.demo.ms")
@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)
@EnableCircuitBreaker
public class MSBusinessTestApplication {

    public static void main(String[] args) {
        SpringApplication.run(MSBusinessTestApplication.class, args);
    }
}
```



Abordando um pouco essas anotações, temos:

**@EnableDiscoveryClient** – Já conhecemos, pois ela habilita nosso discovery do Eureka.

**@EnableFeignClient** – Esta anotação é nova para nós, ela que é responsável por habilitar o uso do Feign no projeto.

**@ComponentScan** – Já conhecida nossa, ela identifica quais os pacotes serão escaneados pelo Spring para que possamos realizar nossas injeções de dependências e outras funcionalidades do framework.

**@EnableAutoConfiguration** – Essa anotação é bem legal, ela habilita a configuração de tudo que está dentro de nosso application.yml/properties, porém, quando eu digo, EXCLUDE, ela simplesmente ignora o que está como parâmetro, no caso, eu informei para ignorar a criação de um DataSource, ou seja, é um micro-serviço que eu não irei neste momento, necessitar de uma base de dados.

**@EnableCircuitBreaker** – Essa anotação habilita nosso Hystrix.

Outras duas classes que eu queria abordar são as do pacote service:

```
▼ com.demo.ms.service
  ▶ AuthService.java
  ▶ AuthServiceImpl.java
```

Vamos à interface AuthService

```
@FeignClient(value = "authService", url = "http://localhost:9092/", fallbackFactory=AuthServiceImpl.class)
public interface AuthService {

    @RequestMapping(value="/api/feign-test", method = RequestMethod.GET, produces = "application/json")
    String getAuth();

}
```

Podemos ver acima que nossa interface está anotada com @FeignClient, com as propriedades, value, url, fallbackFactory.

Value – é o nome pelo qual vamos injetar essa interface.

Url - Neste momento, vamos informar qual micro-serviço queremos acessar, quando esta interface for requisitada, no caso, estamos direcionando para o micro-serviço ms-auth-server, que detém minha regra de negócio de disponibilização do Token.

FallbackFactory – Neste momento, informamos qual classe irá responder em caso de falha, aqui onde o Hystrix entra em ação.

Vamos a nossa implementação desta interface.

```
@Component
public class AuthServiceImpl implements FallbackFactory<AuthService> {

    @Override
    public AuthService create(Throwable cause) {
        // TODO Auto-generated method stub
        return new AuthService() {

            @Override
            public String getAuth() {

                return "INVOKING LOCAL STORAGE ..." + cause.getMessage();
            }
        };
    }
}
```

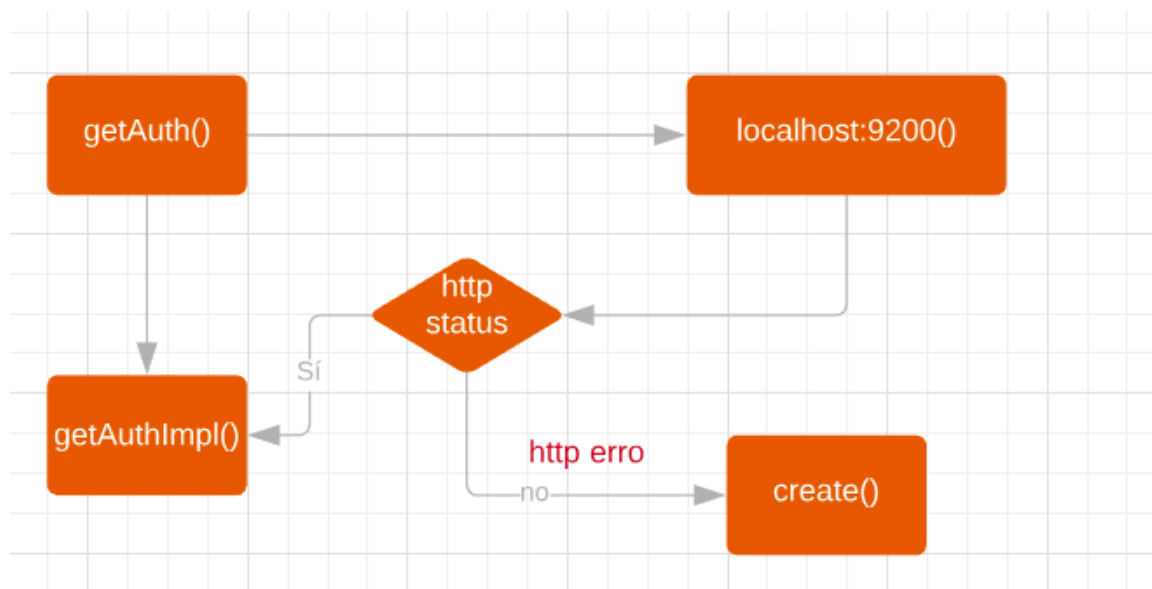
Nossa implementação deve ser OBRIGATORIAMENTE um `@Component`, ou algo que seja filho dele, um `@Service` por exemplo, precisa ser gerenciada pelo Spring, além disso, estamos implementando a classe `FallbackFactory<AuthService>`, informando qual é nossa interface que irá realizar o acesso ao Feign.

Pessoal, essa configuração pode ser realizada simplesmente a nível de método, eu não preciso criar uma fábrica para isso, poderíamos realizar essas funcionalidades a nível de método, porém, optei pela interface para o exemplo ficar mais rico.

Nesse exemplo, utilizei Feign em conjunto com Hystrix, o que também é um ponto novo, o que poderíamos utilizar apenas o Hystrix, sem a necessidade do Feign.

Mas vamos seguir, na nossa implementação, obrigatoriamente a implementação nos da um método `create`, que é do `fallbackFactory`, ele é a fábrica, ele é o que irá tratar o que queremos fazer como reação para a falha, no nosso caso, como eu tenho uma interface, e preciso implementar o que a interface me pede, que é o nosso método `getAuth`, ele que vai nos trazer a resposta do acesso ao micro-serviço de autorização que requisitamos.

Então, o processo em um gráfico seria assim:



Minha interface irá acionar o micro-serviço de autorização, e em caso de sucesso, vida segue. Em caso de falhas, a implementação irá acionar o método create do FallbackFactory, e como tenho minha implementação de falha do getAuth, irá executar alguma funcionalidade que você ache interessante.

Vamos ver funcionando?

Criei no meu projeto um controller, que irá apenas requisitar meu método getAuth, perceba que estou recebendo como header o token, não usarei o mesmo, justamente para apresentar o erro, vamos lá.

No meu projeto iniciado em localhost:9300, que é o meu projeto ms-business-test, criei o seguinte rest.

```
@RestController
@RequestMapping("/api")
public class AccessController {

    @Autowired
    private AuthService service;

    @GetMapping("/me")
    public ResponseEntity<String> getName(@RequestHeader (name="Authorization") String token) {
        return ResponseEntity.ok().body(service.getAuth());
    }
}
```

Apenas requisitei meu método na interface

```
@FeignClient(value = "authService", url = "http://localhost:9092/", fallbackFactory=AuthServiceImpl.class)
public interface AuthService {

    @RequestMapping(value="/api/feign-test", method = RequestMethod.GET, produces = "application/json")
    String getAuth();
}
```

Quando acesso esta interface, ela vai chamar meu serviço em localhost:9092/api/feign-test, um GET. Quem é ele?

```
@GetMapping("/feign-test")
public ResponseEntity<String> feignTest() {

    try {

        AutenticatioVO auth = jwtTokenProvider.getUsername("");
        return ResponseEntity.ok().body(auth.name);

    } catch (Exception e) {
        return new ResponseEntity<String>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

No meu controller em ms-auth-server, tenho esse rest, que irá fazer uma validação se o token existe ou não, propositalmente eu deixei em branco, não estou recebendo o token, e apenas forçando a ele dar um internal server error.

Quando realizo essa chamada e o erro ocorre, ele irá chamar minha classe FallBack , meu método create, que é esse:

```
@Override
public AuthService create(Throwable cause) {
    // TODO Auto-generated method stub
    return new AuthService() {

        @Override
        public String getAuth() {

            return "INVOKING LOCAL STORAGE ..." + cause.getMessage();
        }
    };
}
```

E neste momento, podemos ver o hystrix funcionando , em conjunto com o Feign, segue o Postman com todo o fluxo.

GET	http://localhost:9093/api/me	Params	Sending...	
Authorization	Headers (2)	Body	Pre-request Script	Tests
Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> Content-Type	application/json			
<input checked="" type="checkbox"/> Authorization	eyJhbGciOiJIUzI1NiJ9.eyJzdWliOiJqb25hdGhhcy...			
New key	Value	Description		

Loading...

Cancel Request

```
@RestController
@RequestMapping("/api")
public class AccessController {

    @Autowired
    private AuthService service;

    @GetMapping("/me")
    public ResponseEntity<String> getName(@RequestHeader (name="Authorization") String token) {
        return ResponseEntity.ok().body(service.getAuth());
    }
}
```

Após isso, ele irá requisitar o ms-auth-server

```
@GetMapping("/feign-test")
public ResponseEntity<String> feignTest() {

    try {

        AutenticatioV0 auth = jwtTokenProvider.getUsername("");
        return ResponseEntity.ok().body(auth.name);

    } catch (Exception e) {
        return new ResponseEntity<String>(HttpStatus.INTERNAL_SERVER_ERROR);
    }

}
```

Como o serviço vai registrar uma exceção , ele vai retornar o que esta escrito no nosso método reativo a falha.

GET ▼ http://localhost:9093/api/me Params Send ▼

Authorization Headers (2) Body Pre-request Script Tests

	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	Content-Type	application/json			
<input checked="" type="checkbox"/>	Authorization	eyJhbGciOiJIUzI1NiJ9.eyJzdWliOiJqb25hdGhhcy...			
	New key	Value	Description		

Body Cookies Headers (3) Test Results Status: 200 OK

Pretty Raw Preview Text ▼ ⇒

1 INVOKING LOCAL STORAGE ...

O cause que esta concatenado, você pode injetá-lo em um construtor para ter acesso ao método implementado.

Bom pessoal, este é o fim do meu artigo, espero ter sido claro e conseguido fomentar a estratégia de service discovery e circuit breaker, nos projetos e arquiteturas que utilizamos nos dias de hoje, fugimos a muito tempo das estruturas monolíticas, temos diversas estratégias que nos auxiliam boas arquiteturas distribuídas, desacopladas, e que isolem nossas funcionalidades em serviços realmente necessários, hoje não podemos mais pensar em acoplamentos e sim serviços, e principalmente devemos ser afetados por um serviço que momentaneamente esta com interrupções.

É isso, obrigado aos que tomaram um tempo para ler o artigo, e qualquer dúvidas meu linkedIn e e-mail .

LinkedIn: <https://www.linkedin.com/in/jonathas-lima-a3812953/>

E-Mail: jonathas.o.lima@gmail.com