

Arquitetura de MicroServices com Spring Cloud e Spring Boot — Parte 5



Diego Brener da Silva [Follow](#)

Dec 18, 2017 · 5 min read



Sumário

1. Introdução
2. Implementando o Config Server.
3. Subindo um Eureka Server e conectando-o ao Config Server.
4. Construindo um Servidor de Autorização OAuth2.
5. Implementando Serviço de Pedidos.

Microserviço de Pedido

Olá pessoal, enfim voltamos com nossas publicações. Dando continuação a nossa sequencia de stories sobre microservice, nessa storie vamos criar nosso microserviço de pedido, que será responsável por criar, listar e apagar um pedido criado. Veremos também como integrar esse microserviço ao servidor de autorização criado na storie anterior, então bora lá.

Gerando o Projeto

Acesse o site do Spring Initializr e preencha as configurações como na imagem. Escolhemos qual vai ser o gerenciador de dependências (*Maven*), a linguagem de programação (*Java*), as dependências necessárias para o projeto (*Cloud OAuth2, Actuator, Config Client e Eureka Discovery, JPA, MySql, web*) e configuramos os metadados do Maven. Por fim geramos o projeto clicando em *Generate Project, Ctrl + Enter* ou *Command + Enter*.
(Atenção para a versão do Spring, estamos usando a 1.5 nesse projeto)

Project Metadata

Artifact coordinates

Group

com.coderef

Artifact

delivery-order-service

Name

delivery-order-service

Description

Order Service

Package Name

com.coderef.delivery

Packaging

Jar

Java Version

8

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Actuator

Cloud OAuth2

JPA

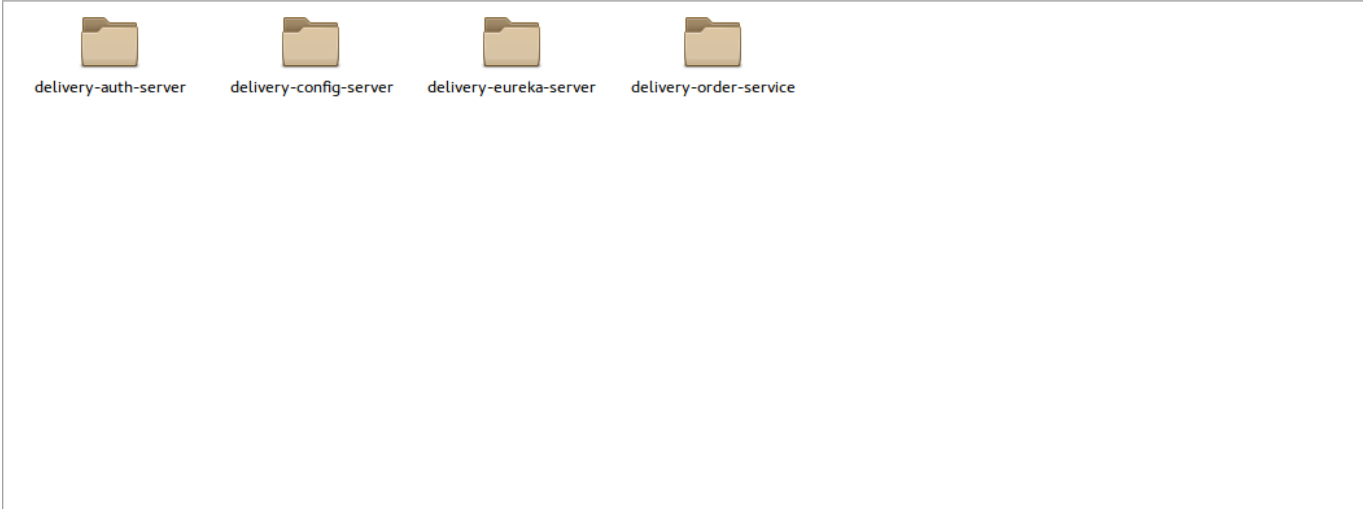
MySQL

Config Client

Eureka Discovery

Web

Vamos descompactar o projeto na pasta delivery, assim como fizemos antes.



Implementando a aplicação

Como sempre fazemos, iremos abrir o `pom.xml` e ver as dependências geradas, a maioria já conhecemos, mas vou frizar uma:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-i
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4      <modelVersion>4.0.0</modelVersion>
5
6      <groupId>com.coderef</groupId>
7      <artifactId>delivery-order-service</artifactId>
8      <version>0.0.1-SNAPSHOT</version>
9      <packaging>jar</packaging>
10
11     <name>delivery-order-service</name>
12     <description>Delivery order service</description>
13
14     <parent>
15         <groupId>org.springframework.boot</groupId>
16         <artifactId>spring-boot-starter-parent</artifactId>
17         <version>1.5.9.RELEASE</version>
18         <relativePath/> <!-- lookup parent from repository -->
19     </parent>
20
21     <properties>
22         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
23         <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
24         <java.version>1.8</java.version>
25         <spring-cloud.version>Edgware.RELEASE</spring-cloud.version>
26     </properties>
27
```


```
28     <dependencies>
29         <dependency>
30             <groupId>org.springframework.boot</groupId>
31             <artifactId>spring-boot-starter-actuator</artifactId>
32         </dependency>
33         <dependency>
34             <groupId>org.springframework.cloud</groupId>
35             <artifactId>spring-cloud-starter-eureka</artifactId>
36         </dependency>
37         <dependency>
38             <groupId>org.springframework.cloud</groupId>
39             <artifactId>spring-cloud-starter-oauth2</artifactId>
40         </dependency>
41         <dependency>
42             <groupId>org.springframework.boot</groupId>
43             <artifactId>spring-boot-starter-data-jpa</artifactId>
44         </dependency>
45         <dependency>
46             <groupId>org.springframework.boot</groupId>
47             <artifactId>spring-boot-starter-web</artifactId>
48         </dependency>
49         <dependency>
50             <groupId>org.springframework.cloud</groupId>
51             <artifactId>spring-cloud-starter-config</artifactId>
52         </dependency>
53
54         <dependency>
55             <groupId>mysql</groupId>
56             <artifactId>mysql-connector-java</artifactId>
57             <scope>runtime</scope>
58         </dependency>
59         <dependency>
60             <groupId>org.springframework.boot</groupId>
61             <artifactId>spring-boot-starter-test</artifactId>
62             <scope>test</scope>
63         </dependency>
64     </dependencies>
65
66     <dependencyManagement>
67         <dependencies>
68             <dependency>
69                 <groupId>org.springframework.cloud</groupId>
70                 <artifactId>spring-cloud-dependencies</artifactId>
71                 <version>${spring-cloud.version}</version>
72                 <type>pom</type>
73                 <scope>import</scope>
74             </dependency>
75         </dependencies>
76     </dependencyManagement>
77
78     <build>
79         <plugins>
80             <plugin>
81                 <groupId>org.springframework.boot</groupId>
82                 <artifactId>spring-boot-maven-plugin</artifactId>
83             </plugin>
84         </plugins>
85     </build>
86
87
88 </project>
```

- **spring-cloud-starter-oauth2:** Colocamos novamente essa dependência e iremos coloca-la novamente em projetos futuros, pois ela será

necessária para implementarmos a conexão com o authorization server.

Vamos criar os arquivos de configuração, neles serão definidas algumas propriedades importantes que serão utilizadas na nossa aplicação. Vamos começar criando nosso `bootstrap.yml`, salve esse arquivo em `delivery-order-service/src/main/resouces`.

```
1  spring:
2    application:
3      name: delivery-order-service
4    cloud:
5      config:
6        uri: http://localhost:9090
```

bootstrap.yml hosted with  by GitHub [view raw](#)

Como vimos na parte 3, nossa aplicação terá que buscar as configurações no Config Server, para isso especificamos novamente a propriedade `spring.cloud.config.uri` . Após criado nosso `bootstrap.yml` vamos criar nosso arquivo de configuração no repositório que o Config Server está usando tal, como vimos na parte 2, crie `delivery-order-service.yml` com:

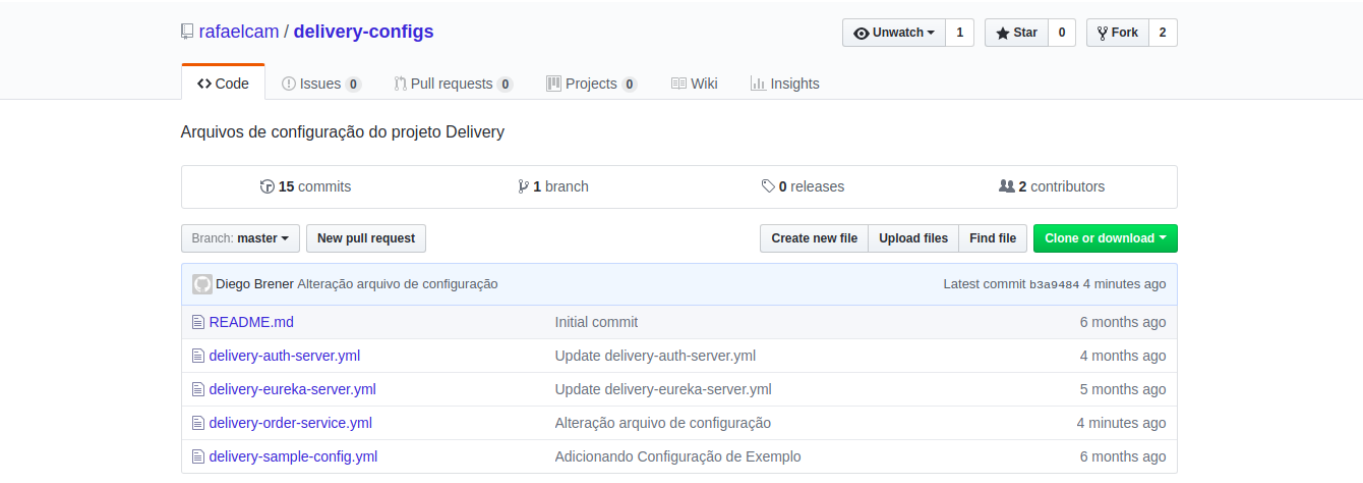
```
1  server:
2    port: 9093
3
4  eureka:
5    instance:
6      hostname: localhost
7    port: 9091
8    client:
9      registerWithEureka: true
10     fetchRegistry: false
11     serviceUrl:
12       defaultZone: http://${eureka.instance.hostname}:${eureka.instance.port}/eureka/
13   server:
14     wait-time-in-ms-when-sync-empty: 3000
15
16  spring:
17    datasource:
18      driver-class-name: com.mysql.jdbc.Driver
19      password: '1234'
20      platform: mysql
21      url: jdbc:mysql://localhost/delivery-order?verifyServerCertificate=false&useSSL=false&re
22      username: root
23    jpa:
24      database-platform: org.hibernate.dialect.MySQLDialect
25      generate-ddl: false
26      hibernate:
27        ddl-auto: create
28      show-sql: true
29
30  authserver:
31    hostname: http://localhost:9092
32  security:
33    oauth2:
34      resource:
35        userInfoUri: ${authserver.hostname}/user
```

delivery-order-service.yml hosted with  by GitHub [view raw](#)

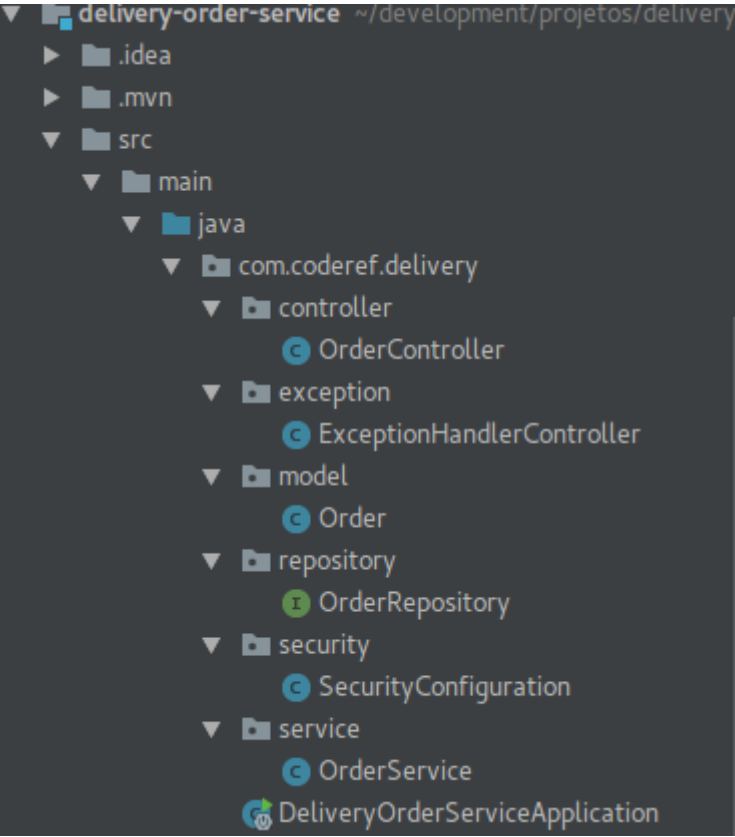
Vamos repassar algumas configurações que são novidades para nós até o momento:

- **authserver.hostname:** Parametro apenas para ser reutilizado em userInfoUri.
- **security.oauth2.resource.userInfoUri:** Url que o spring irá utilizar para validar o token recebido na requisição.

Agora temos 4 arquivos no repositório de configuração contando com o arquivo de exemplo:



Considerando que você já esteja com o projeto aberto na sua IDE favorita, vamos criar a seguinte estrutura de pacotes com as seguintes classes:



Desconsidere os diretórios .idea e .mvn .

Agora vamos implementar cada classe e ver seu conteúdo.

Order

```
1 package com.coderef.delivery.model;
2
3 import org.hibernate.validator.constraints.NotEmpty;
4
```

```
5  import javax.persistence.*;
6  import javax.validation.constraints.NotNull;
7  import java.io.Serializable;
8
9  @Entity
10 @Table(name = "`order`")
11 public class Order implements Serializable {
12
13     private static final long serialVersionUID = 1L;
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.AUTO)
17     private Integer id;
18
19     @NotEmpty(message = "Product required")
20     private String product;
21
22     @NotNull(message = "Price required")
23     private Double price;
24
25     public Integer getId() {
26         return id;
27     }
28
29     public void setId(Integer id) {
30         this.id = id;
31     }
32
33     public String getProduct() {
34         return product;
35     }
36
37     public void setProduct(String product) {
38         this.product = product;
39     }
40
41     public Double getPrice() {
42         return price;
43     }
44
45     public void setPrice(Double price) {
46         this.price = price;
47     }
48 }
```

Order.java hosted with ❤ by GitHub

[view raw](#)

Entidade que representará nosso pedido, iremos utilizar as anotações

`@NotEmpty` e `@NotNull` na classe `OrderService` para validar as propriedades do nosso pedido.

OrderRepository

Essa interface será responsável por fazer a “ponte” entre nossa camada de negócio e o banco de dados, podemos ver que ela está estendendo a Interface `CrudRepository` ela é uma interface do **Spring Data JPA** que fornece uma funcionalidade CRUD completa para a entidade que está sendo gerenciada.

```
1  package com.coderef.delivery.repository;
2
3  import com.coderef.delivery.model.Order;
```

```
3 import com.coderef.delivery.model.Order;
4 import org.springframework.data.repository.CrudRepository;
5
6 public interface OrderRepository extends CrudRepository<Order, Integer> {
7 }
```

OrderRepository.java hosted with ❤ by GitHub

[view raw](#)

OrderService

Aqui estará nosso `@Service` nele poderíamos implementar ou chamar um fluxo de negócio relativo a nossa entidade.

```
1 package com.coderef.delivery.service;
2
3 import com.coderef.delivery.model.Order;
4 import com.coderef.delivery.repository.OrderRepository;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7 import org.springframework.validation.annotation.Validated;
8
9 @Service
10 public class OrderService {
11
12     @Autowired
13     private OrderRepository orderRepository;
14
15     public Order save(@Validated Order order) {
16         return orderRepository.save(order);
17     }
18
19     public Order findById(Integer id){
20         return orderRepository.findOne(id);
21     }
22
23     public Iterable<Order> findAll(){
24         return orderRepository.findAll();
25     }
26
27     public void delete(Integer id) {
28         orderRepository.delete(id);
29     }
30 }
```

OrderService.java hosted with ❤ by GitHub

[view raw](#)

OrderController

Aqui vamos expor nossos endpoints, para que seja possível o acesso.

```
1 package com.coderef.delivery.controller;
2
3 import com.coderef.delivery.model.Order;
4 import com.coderef.delivery.service.OrderService;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.http.MediaType;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.*;
9
10 @RestController
11 @RequestMapping(value = "/api/orders", produces = MediaType.APPLICATION_JSON_VALUE)
12 public class OrderController {
13
```



```
14     @Autowired
15     private OrderService orderService;
16
17     @RequestMapping(method = RequestMethod.POST)
18     public ResponseEntity<Order> save(@RequestBody Order order){
19         return ResponseEntity.ok(orderService.save(order));
20     }
21
22     @RequestMapping(value =("/{id}", method = RequestMethod.GET)
23     public ResponseEntity<Order> findById(@PathVariable("id") Integer id){
24         return ResponseEntity.ok(orderService.findById(id));
25     }
26
27     @RequestMapping(method = RequestMethod.GET)
28     public ResponseEntity<Iterable<Order>> findAll(){
29         return ResponseEntity.ok().body(orderService.findAll());
30     }
31
32     @RequestMapping(value =("/{id}", method = RequestMethod.DELETE)
33     public ResponseEntity<?> delete(@PathVariable("id") Integer id){
34         orderService.delete(id);
35         return ResponseEntity.ok().build();
36     }
37 }
```

OrderController.java hosted with  by GitHub

[view raw](#)

ExceptionHandlerController

Essa `controller` será responsável por interceptar e tratar as exceções lançadas pela nossa aplicação:

```
1  package com.coderef.delivery.exception;
2
3  import org.springframework.http.HttpStatus;
4  import org.springframework.http.ResponseEntity;
5  import org.springframework.web.bind.annotation.ControllerAdvice;
6  import org.springframework.web.bind.annotation.ExceptionHandler;
7
8  import javax.validation.ConstraintViolationException;
9  import java.util.stream.Collectors;
10
11  @ControllerAdvice
12  public class ExceptionHandlerController {
13
14      @ExceptionHandler(ConstraintViolationException.class)
15      public ResponseEntity<?> validateError(ConstraintViolationException ex){
16          return ResponseEntity.badRequest().body(ex.getConstraintViolations().stream().map(cv ->
17      }
18
19      @ExceptionHandler(Exception.class)
20      public ResponseEntity<?> otherErrors(Exception ex){
21          return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(ex.getMessage());
22      }
23  }
```

ExceptionHandlerController.java hosted with  by GitHub

[view raw](#)

SecurityConfiguration

Essa classe vai ativar nosso Resource Server e mapear cada `role` para seu `method` específico:


```
1 package com.coderef.delivery.security;
2
3 import org.springframework.boot.autoconfigure.security.oauth2.client.EnableOAuth2Sso;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.http.HttpMethod;
6 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
7 import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;
8 import org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;
9 import org.springframework.security.oauth2.config.annotation.web.configurers.ResourceServerSecurityConfigurer;
10
11
12 @Configuration
13 @EnableResourceServer
14 public class SecurityConfiguration extends ResourceServerConfigurerAdapter {
15
16     private final static String resourceId = "resources";
17
18     @Override
19     public void configure(HttpSecurity http) throws Exception {
20         http.requestMatchers()
21             .antMatchers("/**")
22             .and()
23             .authorizeRequests()
24             .anyRequest()
25             .authenticated()
26             .antMatchers(HttpMethod.GET, "/**").access("#oauth2.hasScope('read')")
27             .antMatchers(HttpMethod.OPTIONS, "/**").access("#oauth2.hasScope('read')")
28             .antMatchers(HttpMethod.POST, "/**").access("#oauth2.hasScope('write')")
29             .antMatchers(HttpMethod.PUT, "/**").access("#oauth2.hasScope('write')")
30             .antMatchers(HttpMethod.PATCH, "/**").access("#oauth2.hasScope('write')")
31             .antMatchers(HttpMethod.DELETE, "/**").access("#oauth2.hasScope('write')");
32     }
33
34     @Override
35     public void configure(ResourceServerSecurityConfigurer resources){
36         resources.resourceId(resourceId);
37     }
38 }
```

SecurityConfiguration.java hosted with  by GitHub [view raw](#)

DeliveryOrderServiceApplication

Por último implementamos nossa classe principal, que será responsável por inicializar nossa aplicação Spring Boot como vimos na parte 2. Aqui temos uma anotação nova a `@EnableEurekaClient`. Essa anotação torna nossa aplicação visível ao Eureka Server com base nas configurações que definimos no `application.yml`.

```
1 package com.coderef.delivery;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
6 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
7
8 @SpringBootApplication
9 @EnableEurekaClient
10 public class DeliveryOrderServiceApplication {
11
12     public static void main(String[] args) {
13         SpringApplication.run(DeliveryOrderServiceApplication.class, args);
14     }
15 }
```

DeliveryOrderServiceApplication.java hosted with  by GitHub

[view raw](#)


Subindo a aplicação

Agora chegou a hora subir o nosso MicroService, antes de inciar será necessário subir o Config Server , Eureka Server e o Authorization Server.

Quando os três estiverem online vamos executar a classe

`DeliveryOrderServiceApplication.java`. Finalizado, acesse o endereço

`http://localhost:9091/` nosso servidor vai aparecer listado no Eureka Server:

HOME LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2017-12-17T16:26:23 -0200
Data center	default	Uptime	00:22
		Lease expiration enabled	true
		Renews threshold	1
		Renews (last min)	6

DS Replicas



Instances currently registered with Eureka


Application	AMIs	Availability Zones	Status
DELIVERY-AUTH-SERVER	n/a (1)	(1)	UP (1) - 172.17.0.1:delivery-auth-server:9092
DELIVERY-EUREKA-SERVER	n/a (1)	(1)	UP (1) - 172.17.0.1:delivery-eureka-server:9091
DELIVERY-ORDER-SERVICE	n/a (1)	(1)	UP (1) - 172.17.0.1:delivery-order-service:9093

Criando um Pedido

Antes de fazermos a requisição para o nosso MicroService, será necessário solicitarmos um `token` para nosso Authorization Server. Vamos abrir o Postman e criar uma nova requisição:


http://localhost:9092/ + ...


No Environment  

POST 

http://localhost:9092/oauth/token?grant_type=password&username=admin&password=123456

Params

Send 

Save 

Authorization

Headers (1)

Body

Pre-request Script

Tests

Code

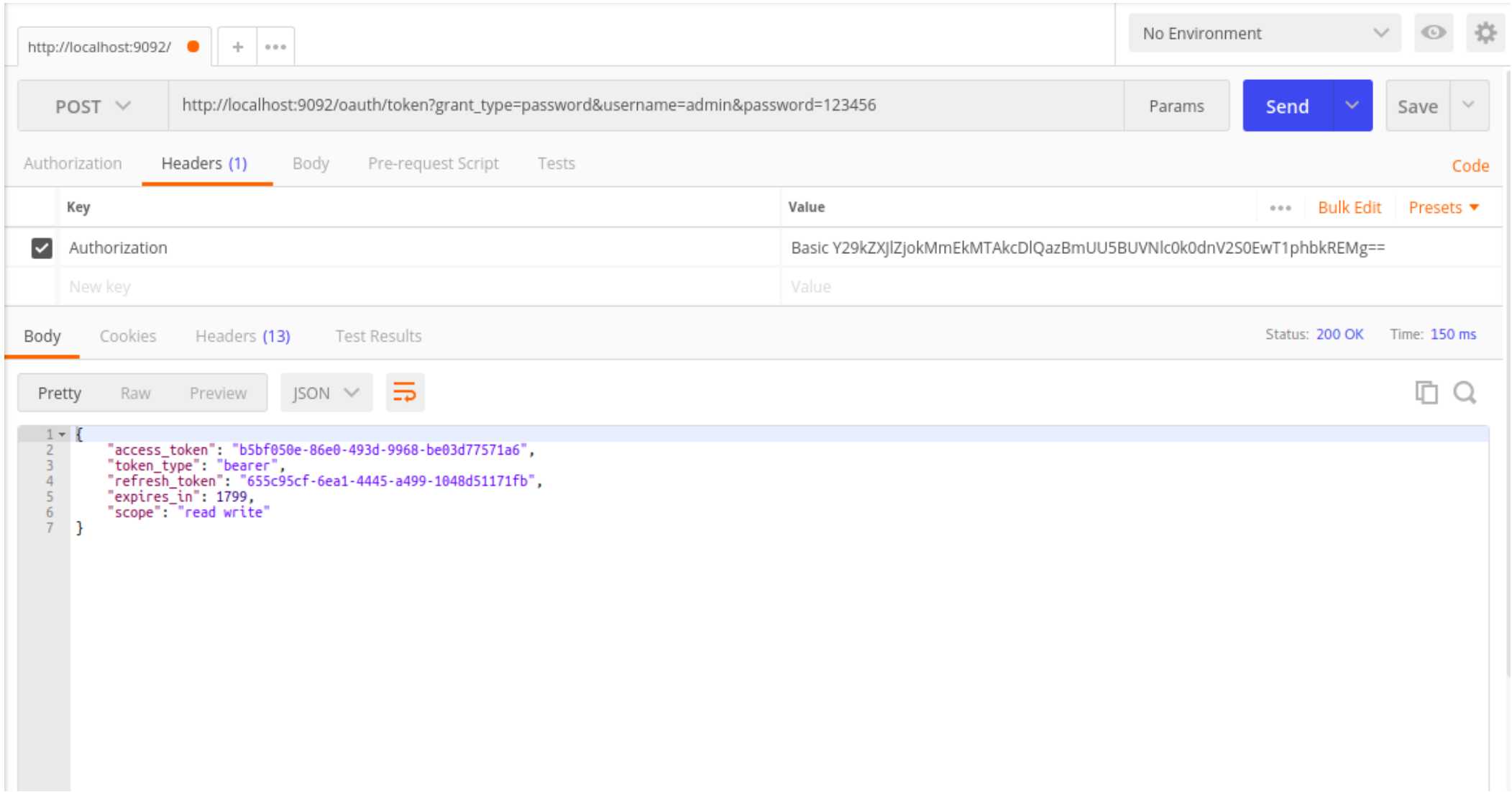
	Key	Value	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	Basic Y29kZXJlZjokMmEkMTAkcdIQazBmUU5BUVNlc0k0dnV2S0EwT1phbkREMg==			
	New key	Value			

Response

- **URL:** `http://localhost:9092/oauth/token?`
`grant_type=password&username=admin&password=123456`
- **Authorization:** Basic
`Y29kZXJlZjokMmEkMTAkcdIQazBmUU5BUVNlc0k0dnV2S0EwT1phbk`
`REMg= =`

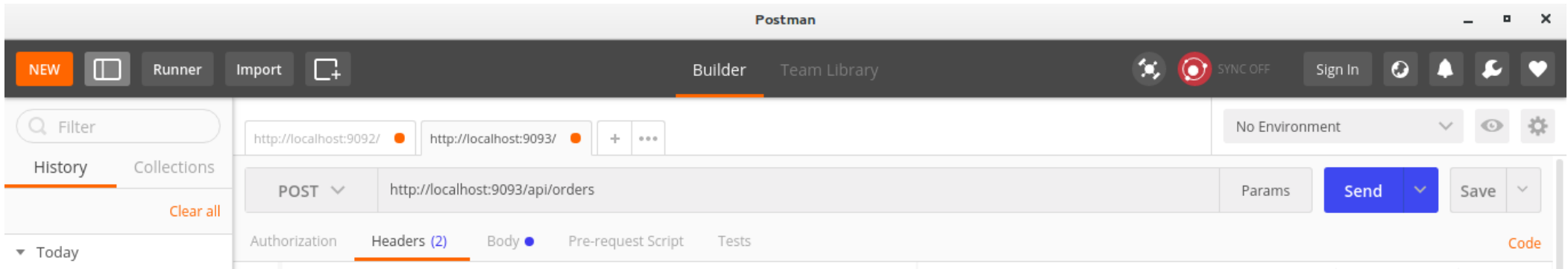
- **Method:** POST.
- **Content-Type:** application/json.

Clique em **Send**. Uma resposta similar a imagem abaixo será retornada:



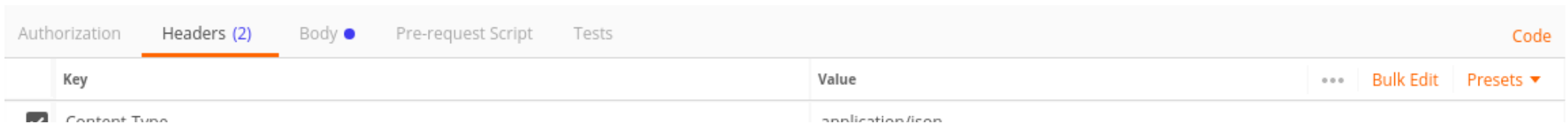
Copiaremos o `access_token` e o `token_type` , pois será eles que iremos utilizar na próxima requisição.

Agora que já temos um `token` vamos criar nosso primeiro pedido. Vamos até o **Postman** novamente e preencher:



- **URL:** http://localhost:9093/api/orders
- **Mehod:** POST

Clique em **Headers** e preencha com:



26/10/2019Arquitetura de MicroServices com Spring Cloud e Spring Boot — Parte 5

<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	Authorization	Bearer b5bf050e-86e0-493d-9968-be03d77571a6
	New key	Value

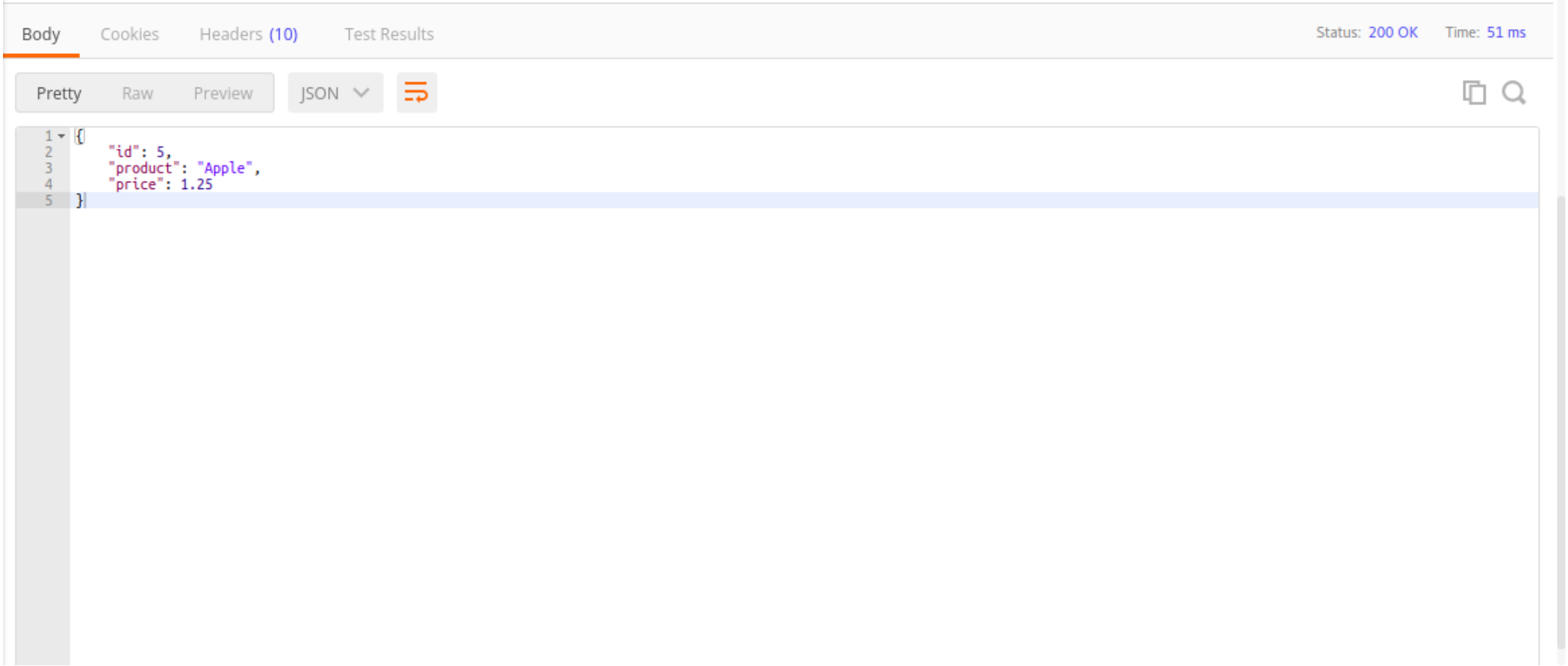
- **Content-Type:** application/json.
- **Authorizarion:** cole o `token_type` e o `access_token` com um espacinho entre os dois.

Clique em **body**, depois em **raw** e selecione **JSON**(application/json), preencha o **TextArea** com um **json** similar ao conteúdo abaixo:

```
1 {
2   "product": "Apple",
3   "price": "1.25"
4 }
```

product.json hosted with ❤ by GitHubview raw

Clique em **Send**, um pedido com `id` será retornado na requisição representando que o pedido foi criado com sucesso.



Pronto, agora podemos realizar os mesmo passos para os endpoints de **busca** e **apagar**, não podemos esquecer que alguns possuem parametros na requisição e que se não forem passados pode ocorrer erro.

Referências

<http://projects.spring.io/spring-security-oauth/https://tools.ietf.org/pdf/draft-ietf-oauth-v2-31.pdf>
<https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html>

Repositório

<https://github.com/diegosilva13/delivery>

Thanks to Matheus Rodrigues.

[Spring Boot](#) [Spring Cloud](#) [Spring Security](#) [Spring](#) [Microservices](#)

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just \$5/month. Upgrade

[About](#) [Help](#) [Legal](#)