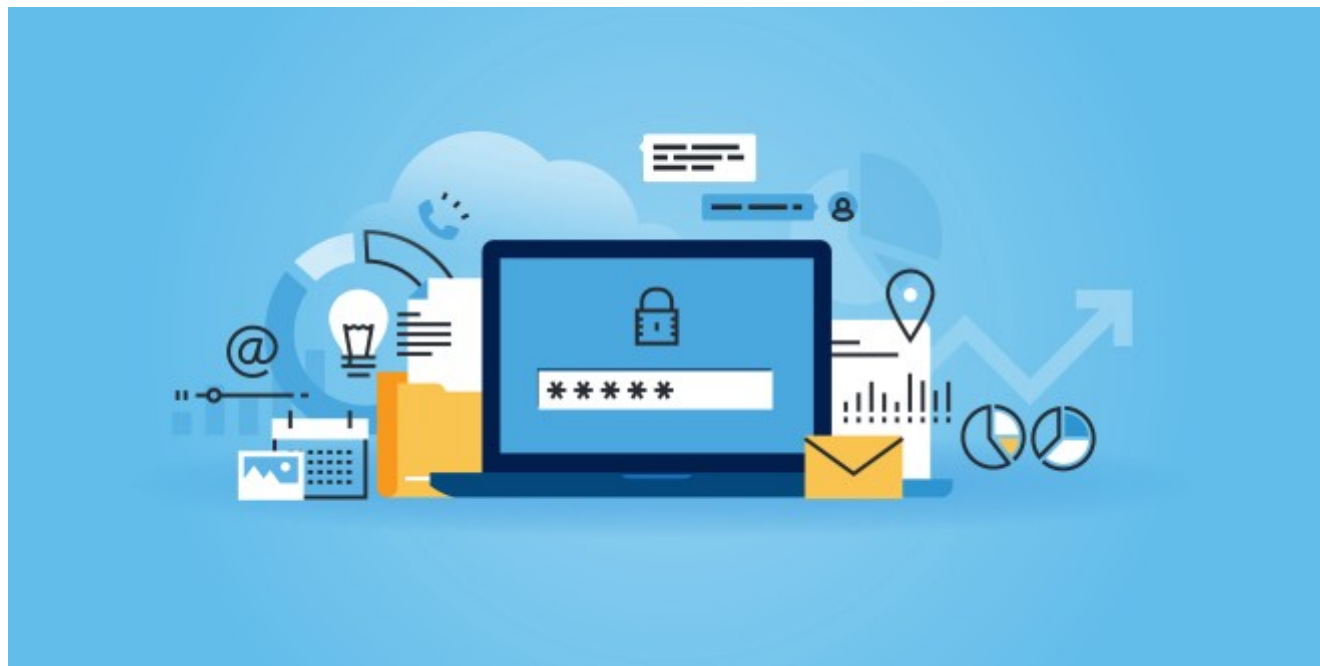


# Arquitetura de MicroServices com Spring Cloud e Spring Boot — Parte 4



Diego Brener da Silva [Follow](#)

Aug 31, 2017 · 9 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.

## O Servidor de Autorização

Olá pessoal, dando continuidade a nossa sequência de stories sobre Microservices, hoje vamos criar nosso servidor de autorização que vai ser responsável por controlar a autenticação e autorização de nossos recursos. Nessa story vamos utilizar o Spring Security OAuth uma ferramenta poderosa disponibilizada pela Pivotal. Antes de iniciar a implementação do nosso Servidor de Autorização, vamos conhecer alguns conceitos importantes, isso é necessário para entendermos como funciona e qual o objetivo das tecnologias utilizadas. Vamos lá?

## OAuth2





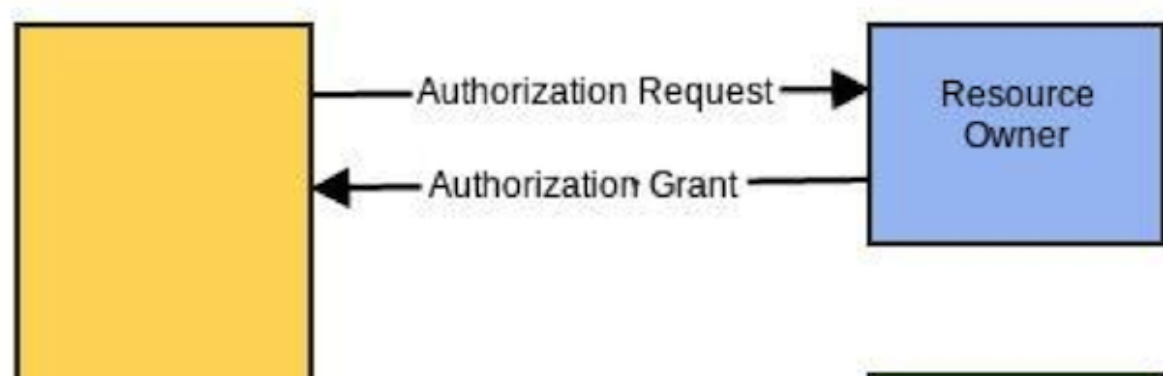
O Spring Security OAuth2 tem como base o OAuth2 um framework de autenticação e autorização aberto, poderoso e flexível permitindo que sua aplicação não fique manipulando diretamente as credenciais dos usuários.

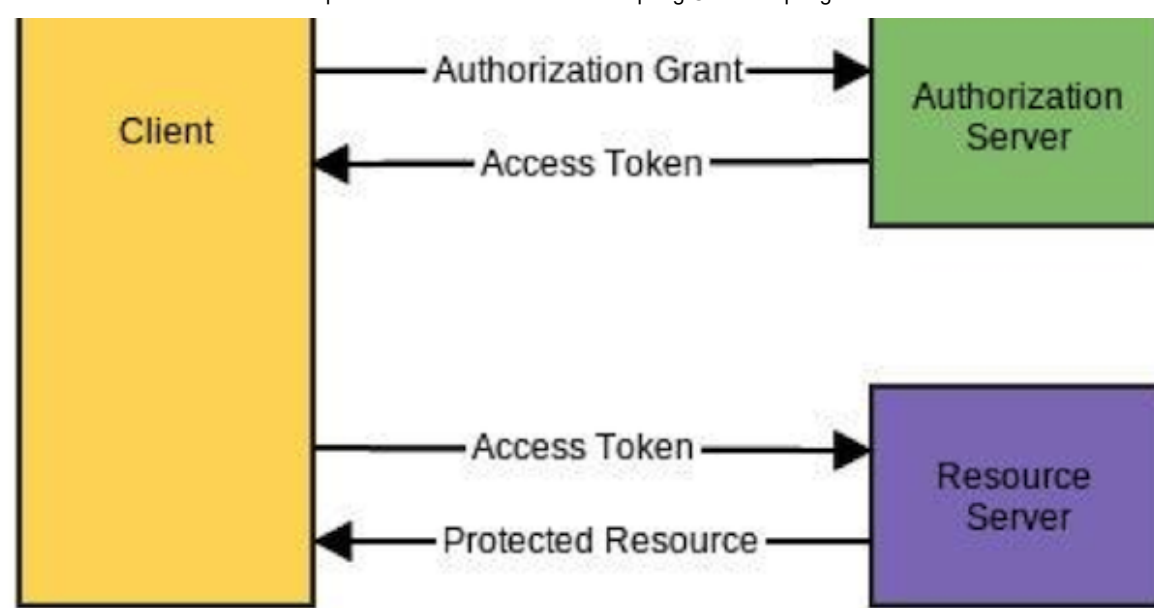
Em um modelo tradicional de autenticação o cliente entra com suas credenciais diretamente e obtém acesso ao recurso protegido. Ficar trafegando as credenciais do usuário em cada requisição pode causar vários problemas e limitações, exemplo:

- Aplicações de terceiros terão que armazenar as credenciais do usuário para solicitar um acesso no futuro.
- O servidor de recursos tem que garantir e se preocupar sempre com um bom suporte a autenticação.
- Alteração das credenciais dos usuários no servidor de recursos afetará todas as aplicações de terceiros.

O OAuth2 busca solucionar esses problemas definindo papéis e criando uma camada intermediária de autenticação, com isso a aplicação cliente solicita uma concessão de autorização ao usuário e caso seja concedida ela é enviada ao servidor de autorização, esse por sua vez autentica e valida o usuário, se tudo estiver correto um token é emitido para que o cliente possa acessar o servidor de recursos. Confuso, não é mesmo? Calma, vamos identificar e explicar passo a passo o fluxo abaixo:

- **Resource Owner** é o proprietário do recurso, ou seja, o usuário que irá entrar com as credenciais de acesso.
- **Client** é a aplicação que irá acessar o servidor de recursos. Ela pode ser uma aplicação mobile, desktop, etc.
- **Authorization server** é o servidor autenticação e autorização. É o que nós vamos implementa-lo nessa story.
- **Resource server** são os recursos protegidos.





Fluxo do Oauth2

## Fluxo de Autenticação

- O **Client** solicita autorização ao **Resource Owner**.
- O **Client** recebe uma concessão de autorização do **Resource Owner**, que é uma credencial seguindo um dos padrões definidos pelo OAuth2.
- O **Client** solicita um token de acesso ao **Authorization Server**.
- O **Authorization Server** autentica o **Client** e valida a credencial informada pelo **Resource Owner**. Se for válida, um token de acesso é emitido..
- O **Client** solicita o acesso ao recurso protegido do **Resource Server** informando o **Token** de acesso.
- O **Resource Server** valida o token de acesso. Se for válido ele atende a solicitação.

Agora que deixamos claro o funcionamento de um Servidor de Autorização com OAuth2, vamos implementá-lo.

## Gerando a aplicação no Spring Initializr

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*) 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**)

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.4

Project Metadata

Dependencies

Project metadata

Artifact coordinates

Group

com.coderef

Artifact

delivery-auth-server

Name

delivery-auth-server

Description

Oauth Server

Package Name

com.coderef.delivery

Packaging

Jar

Java Version

1.8

Too many options? [Switch back to the simple version.](#)

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

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

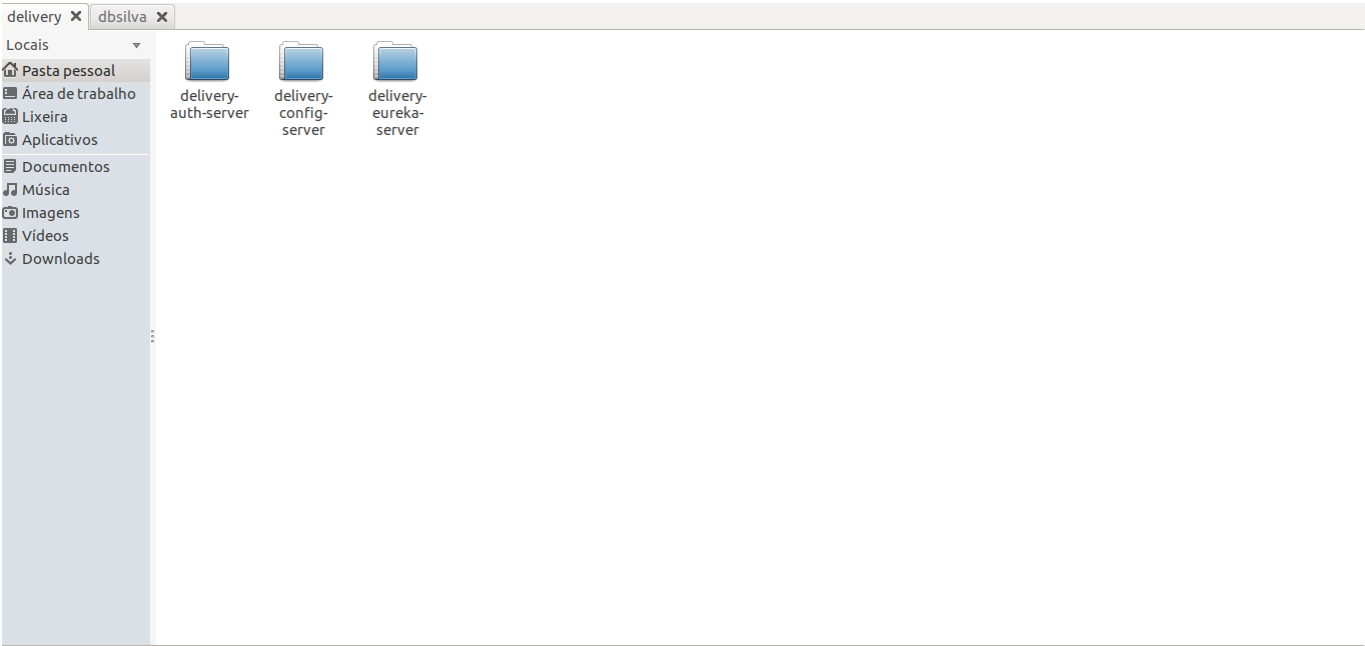
Selected Dependencies

Cloud OAuth2 × Actuator × Config Client × Eureka Discovery ×

JPA × MySQL ×

Generate Project alt + ↵

Vamos descompactar esse projeto na pasta delivery, onde está o Config Server e o Eureka Server.



## Implementando a Aplicação

Como de costume, não vamos discutir as configurações que já foram vistas nas stories anteriores, assim o conteúdo fica mais objetivo.

Abaixo está o `pom.xml` gerado, vamos analisar as dependências inéditas.

```
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-auth-server</artifactId>
8      <version>0.0.1-SNAPSHOT</version>
9      <packaging>jar</packaging>
10
11     <name>delivery-auth-server</name>
12     <description>Oauth Server</description>
13
14     <parent>
15         <groupId>org.springframework.boot</groupId>
16         <artifactId>spring-boot-starter-parent</artifactId>
17         <version>1.5.4.RELEASE</version>
18         <relativePath/> <!-- lookup parent from repository -->
19     </parent>
```

```
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>Dalston.SR2</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-config</artifactId>
36     </dependency>
37     <dependency>
38         <groupId>org.springframework.cloud</groupId>
39         <artifactId>spring-cloud-starter-eureka</artifactId>
40     </dependency>
41     <dependency>
42         <groupId>org.springframework.cloud</groupId>
43         <artifactId>spring-cloud-starter-oauth2</artifactId>
44     </dependency>
45     <dependency>
46         <groupId>org.springframework.boot</groupId>
47         <artifactId>spring-boot-starter-data-jpa</artifactId>
48     </dependency>
49     <dependency>
50         <groupId>mysql</groupId>
51         <artifactId>mysql-connector-java</artifactId>
52         <scope>runtime</scope>
53     </dependency>
54
55     <dependency>
56         <groupId>org.springframework.boot</groupId>
57         <artifactId>spring-boot-starter-test</artifactId>
58         <scope>test</scope>
59     </dependency>
60 </dependencies>
61
62 <dependencyManagement>
63     <dependencies>
64         <dependency>
65             <groupId>org.springframework.cloud</groupId>
66             <artifactId>spring-cloud-dependencies</artifactId>
67             <version>${spring-cloud.version}</version>
68             <type>pom</type>
69             <scope>import</scope>
70         </dependency>
71     </dependencies>
72 </dependencyManagement>
73
74 <build>
75     <plugins>
76         <plugin>
77             <groupId>org.springframework.boot</groupId>
78             <artifactId>spring-boot-maven-plugin</artifactId>
79         </plugin>
80     </plugins>
81 </build>
82
83
84 </project>
```

Comparando com os projetos anteriores, temos apenas 3 dependências que ainda não vimos, são elas:

- **spring-cloud-starter-oauth2:** Essa é a principal dependência desse projeto. Ela oferece toda a implementação necessária para a aplicação funcionar como servidor de autorização Oauth2.
- **spring-boot-starter-data-jpa:** Como o próprio nome diz, usaremos essa dependência para trabalhar com nossa camada de persistência. Ela já fornece tudo que precisamos como por exemplo: spring-data-jpa, spring-orm e o hibernate.
- **mysql-connector-java:** Aqui temos o driver JDBC do MySql(Banco de Dados que vamos utilizar).

Antes de implementarmos as classes do projeto, 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-auth-server/src/main/resouces`.

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

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

Nele definimos a propriedade `spring.cloud.config.uri` com o host do Config Server onde a aplicação irá buscar as propriedades de configuração durante a inicialização com base no valor de `spring.application.name` como vimos na parte 3.

Após criar o `bootstrap.yml` vamos criar o arquivo de configuração no repositório do Config Server assim como vimos na parte 2. Crie o arquivo `delivery-auth-server.yml` com o conteúdo abaixo:

```
1  server:
2    port: 9092
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 security:
17   oauth2:
18     client:
19       access-token-validity-seconds: 1800
20       authorized-grant-types: password,refresh_token
21       client-id: coderef
22       client-secret: $2a$10$p9Pk0fQNAQSeSI4vuvKA00ZanDD2
23       resource-ids: resources
24       scope: read,write
25
26 spring:
27   datasource:
28     data: classpath:/sql/data.sql
29     driver-class-name: com.mysql.jdbc.Driver
30     password: '1234'
31     platform: mysql
32     schema: classpath:/sql/schema.sql
33     url: jdbc:mysql://localhost/oauth?verifyServerCertificate=false&useSSL=false&requireSSL=
34     username: root
35   jpa:
36     database-platform: org.hibernate.dialect.MySQLDialect
37     generate-ddl: false
38     hibernate:
39       ddl-auto: none
40     show-sql: true
```

delivery-auth-server.yml hosted with ❤ by GitHub [view raw](#)

Como podemos perceber existem algumas configurações que ainda não conhecemos, primeiro vamos ver as relacionadas com a parte de segurança depois vamos conferir as configurações de persistência:

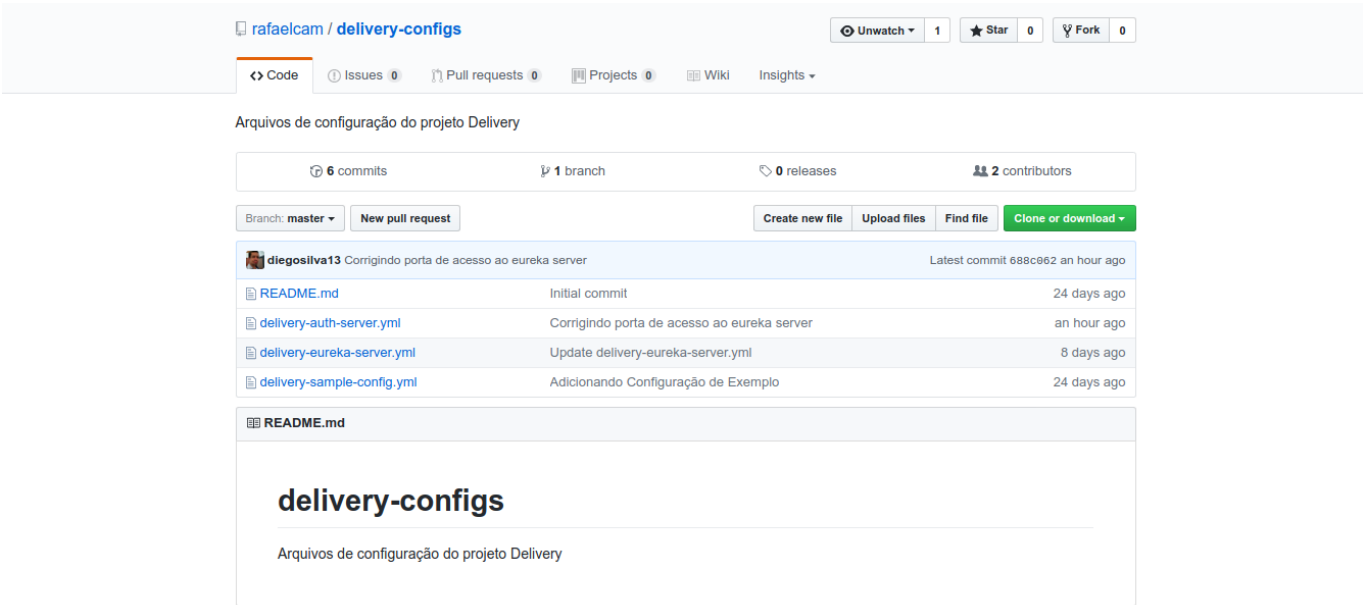
Configurações de segurança:

- **security.oauth2.client.access-token-validity-seconds:** Tempo, em segundos, de duração do token de acesso.
- **security.oauth2.client.authorized-grant-types:** Qual o tipo do pedido de concessão de acesso. Se o Client vai solicitar um token utilizando usuário e senha ou se vai renovar um token já existente.
- **security.oauth2.client.client-id:** Identificador único que será usado pelo Client para solicitar concessão de acesso.
- **security.oauth2.client.client-secret:** Uma senha usada em conjunto com o client-id como autenticação básica do client que está tentando obter um token de acesso.
- **security.oauth2.client.resource-ids:** Identificador dos recursos disponíveis, se especificado aqui também deve ser especificado no servidor de recursos.
- **security.oauth2.client.scope:** Os escopos que o Client terá acesso.

Configurações para o JPA:

- **spring.datasource.data:** Um arquivo .sql que será executado na inicialização da aplicação, ele será executado após a propriedade **schema**.
- **spring.datasource.driver-class-name:** Classe do driver JDBC.
- **spring.datasource.password:** Senha de acesso ao banco de dados.
- **spring.datasource.platform:** Plataforma de persistência.
- **spring.datasource.schema:** Arquivo com a estrutura do banco de dados. Como as classes que mapeiam os dados de segurança então nas bibliotecas do Spring, elas não são criadas automaticamente pelo JPA.
- **spring.datasource.url:** Url de acesso ao banco de dados.
- **spring.datasource.username:** Usuário de acesso ao banco de dados.
- **spring.datasource.jpa.database-platform:** Classe que possui a implementação que possibilita a geração SQL para a plataforma de persistência especificada, para que todo o SQL gerado pelo JPA seja compatível com o banco de dados escolhido.
- **spring.datasource.jpa.generate-ddl:** Informar se durante a inicialização será necessário gerar as estruturas do banco de dados com base nas anotações JPA.
- **spring.datasource.jpa.hibernate.ddl-auto:** Caso a propriedade **generate-ddl** esteja marcada como **true**, aqui deverá ser informada qual a política de geração do schema do banco de dados.
- **spring.datasource.jpa.hibernate.show-sql:** Se habilitada, será gerado um log com o sql gerado pelo JPA. Use somente em Homologação.

Agora temos 3 arquivos de configuração no repositório:




Bom, retornando a nossa aplicação. Se você percebeu durante nossa análise do arquivo de configuração `delivery-auth-server.yml` nós configuramos as propriedades `spring.datasource.data` e `spring.datasource.schema` que fazem referência a um arquivo `.sql`. para hospedar esses dois arquivos



vamos criar uma pasta `sql` no mesmo lugar onde colocamos o arquivo `bootstrap.yml` . Primeiro vamos criar o arquivo que conterà a estrutura do nosso banco de dados de autenticação, além de possuir algumas tabelas importantes que serão utilizadas pelo Spring para gerenciar a sessão do **usuário** logado. Crie um arquivo `schema.sql` com o seguinte conteúdo:

```
1 CREATE TABLE IF NOT EXISTS user (
2     username VARCHAR(50) NOT NULL PRIMARY KEY,
3     email VARCHAR(50),
4     password VARCHAR(500),
5     activated BOOLEAN DEFAULT FALSE,
6     activationkey VARCHAR(50) DEFAULT NULL,
7     resetpasswordkey VARCHAR(50) DEFAULT NULL
8 );
9
10 CREATE TABLE IF NOT EXISTS authority (
11     name VARCHAR(50) NOT NULL PRIMARY KEY
12 );
13
14 CREATE TABLE IF NOT EXISTS user_authority (
15     username VARCHAR(50) NOT NULL,
16     authority VARCHAR(50) NOT NULL,
17     FOREIGN KEY (username) REFERENCES user (username),
18     FOREIGN KEY (authority) REFERENCES authority (name),
19     UNIQUE INDEX user_authority_idx_1 (username, authority)
20 );
21
22 CREATE TABLE IF NOT EXISTS oauth_access_token (
23     token_id VARCHAR(256) DEFAULT NULL,
24     token BLOB,
25     authentication_id VARCHAR(256) DEFAULT NULL,
26     user_name VARCHAR(256) DEFAULT NULL,
27     client_id VARCHAR(256) DEFAULT NULL,
28     authentication BLOB,
29     refresh_token VARCHAR(256) DEFAULT NULL
30 );
31
32 CREATE TABLE IF NOT EXISTS oauth_refresh_token (
33     token_id VARCHAR(256) DEFAULT NULL,
34     token BLOB,
35     authentication BLOB
36 );
37
38 CREATE TABLE IF NOT EXISTS oauth_client_details (
39     client_id VARCHAR(255) PRIMARY KEY,
40     resource_ids VARCHAR(255),
41     client_secret VARCHAR(255),
42     scope VARCHAR(255),
43     authorized_grant_types VARCHAR(255),
44     web_server_redirect_uri VARCHAR(255),
45     authorities VARCHAR(255),
46     access_token_validity INTEGER,
47     refresh_token_validity INTEGER,
48     additional_information VARCHAR(4096),
49     autoapprove VARCHAR(255)
50 );
```

schema.sql hosted with  by GitHub [view raw](#)

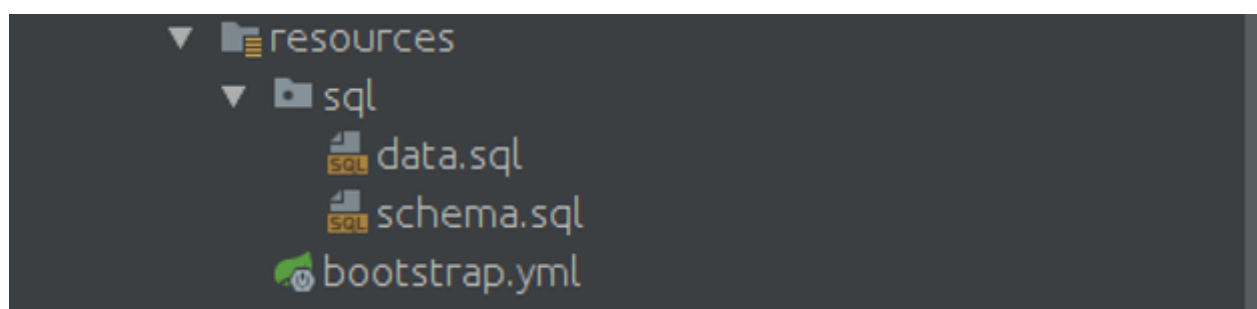
Agora vamos criar o arquivo que ira apagar os Clients salvos e inserir os dados do primeiro usuário da aplicação, o **administrador**. Criaremos um arquivo com o nome `data.sql` com o seguinte conteúdo:

```
1
2  DELETE FROM oauth_client_details;
3
4  INSERT INTO user (username, email, password, activated)
5  SELECT * FROM (SELECT 'admin', 'admin@admin.com', '$2a$10$r0RFDmpneBVryx.iHhK9gu6FFJQi4nTxQUqzdS'
6  WHERE NOT EXISTS (
7      SELECT username FROM user WHERE username = 'admin'
8  ) LIMIT 1;
9
10 INSERT INTO authority (name)
11 SELECT * FROM (SELECT 'ROLE_USER') AS tmp
12 WHERE NOT EXISTS (
13     SELECT name FROM authority WHERE name = 'ROLE_USER'
14 ) LIMIT 1;
15
16 INSERT INTO authority (name)
17 SELECT * FROM (SELECT 'ROLE_ADMIN') AS tmp
18 WHERE NOT EXISTS (
19     SELECT name FROM authority WHERE name = 'ROLE_ADMIN'
20 ) LIMIT 1;
21
22 INSERT INTO user_authority (username, authority)
23 SELECT * FROM (SELECT 'admin', 'ROLE_USER') AS tmp
24 WHERE NOT EXISTS (
25     SELECT username, authority FROM user_authority WHERE username = 'admin' and authority = 'ROLE_USER'
26 ) LIMIT 1;
27
28 INSERT INTO user_authority (username, authority)
29 SELECT * FROM (SELECT 'admin', 'ROLE_ADMIN') AS tmp
30 WHERE NOT EXISTS (
31     SELECT username, authority FROM user_authority WHERE username = 'admin' and authority = 'ROLE_ADMIN'
32 ) LIMIT 1;
```

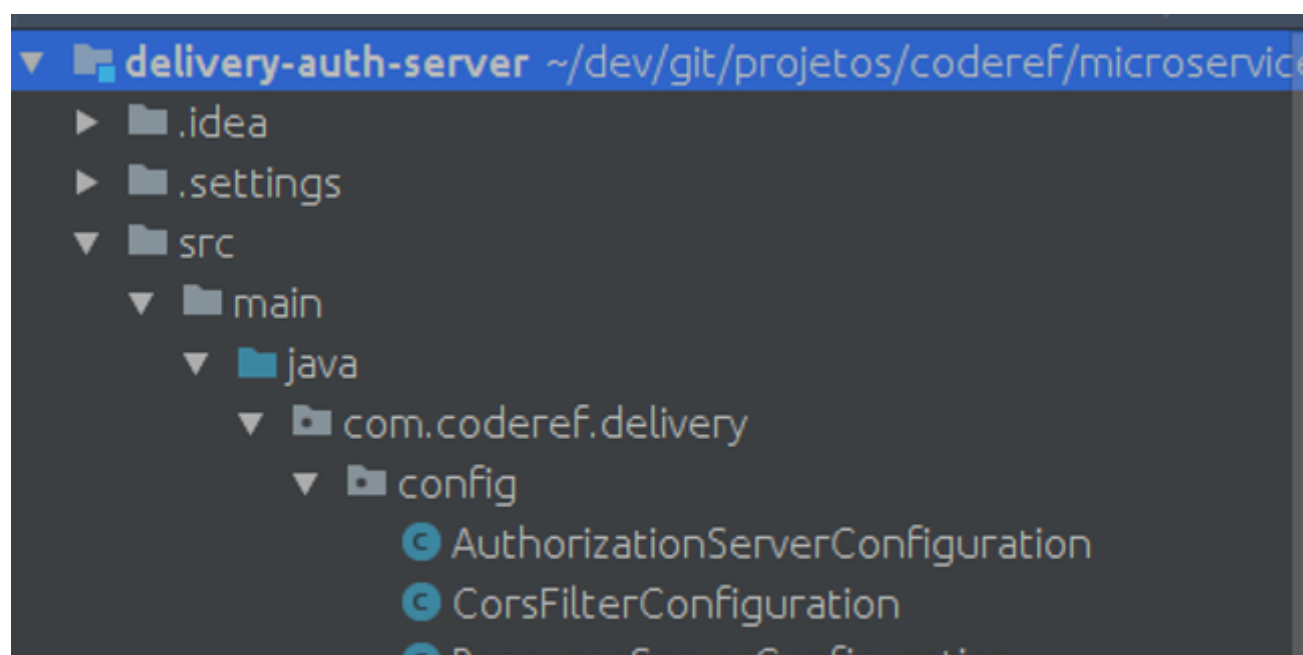
data.sql hosted with ❤ by GitHub

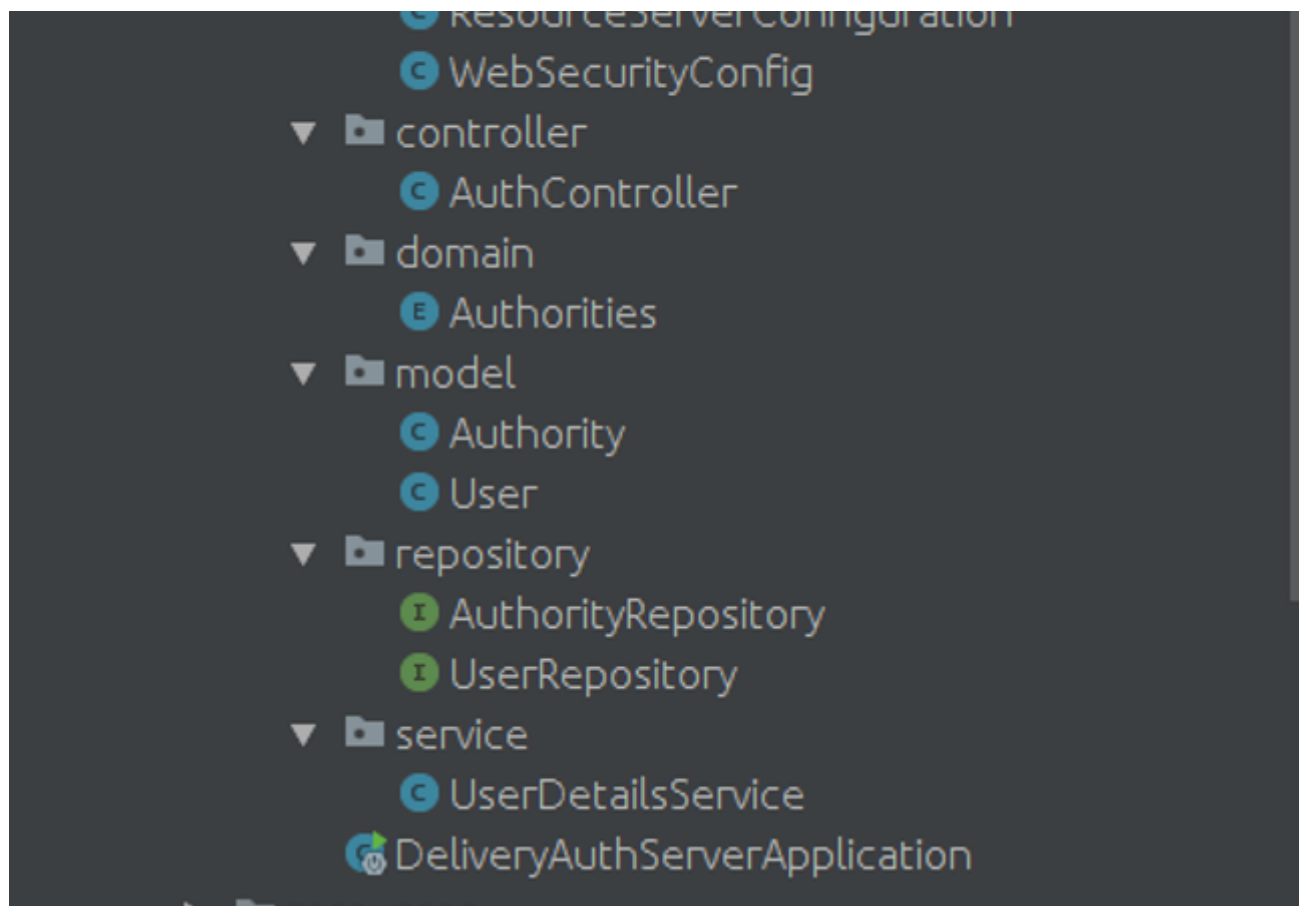
[view raw](#)

Após criar esses dois arquivos temos a seguinte estrutura:



Considerando que você já tenha importado o projeto na sua IDE vamos criar os seguintes pacotes e classes:





Agora vamos implementar cada classe e ver seu conteúdo.

## AuthorizationServerConfiguration

```

1  package com.coderef.delivery.config;
2
3  import javax.sql.DataSource;
4
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.beans.factory.annotation.Qualifier;
7  import org.springframework.beans.factory.annotation.Value;
8  import org.springframework.context.annotation.Bean;
9  import org.springframework.context.annotation.Configuration;
10 import org.springframework.security.authentication.AuthenticationManager;
11 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
12 import org.springframework.security.crypto.password.PasswordEncoder;
13 import org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;
14 import org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;
15 import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;
16 import org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerConfigurerAdapter;
17 import org.springframework.security.oauth2.provider.token.store.JdbcTokenStore;
18
19 import com.coderef.delivery.domain.Authorities;
20
21
22 @Configuration
23 @EnableAuthorizationServer
24 public class AuthorizationServerConfiguration extends
25     AuthorizationServerConfigurerAdapter {
26
27     private static PasswordEncoder encoder;
28
29     @Value("${security.oauth2.client.client-id}")
30     private String clientId;
31
32     @Value("${security.oauth2.client.authorized-grant-types}")
33     private String[] authorizedGrantTypes;
34
35     @Value("${security.oauth2.client.resource-ids}")
36     private String resourceIds;
37
38     @Value("${security.oauth2.client.scope}")
39     private String[] scopes;
40

```

```
41     @Value("${security.oauth2.client.client-secret}")
42     private String secret;
43
44     @Value("${security.oauth2.client.access-token-validity-seconds}")
45     private Integer accessTokenValiditySeconds;
46
47     @Autowired
48     DataSource dataSource;
49
50     @Autowired
51     @Qualifier("authenticationManagerBean")
52     private AuthenticationManager authenticationManager;
53
54     @Bean
55     public JdbcTokenStore tokenStore() {
56         return new JdbcTokenStore(dataSource);
57     }
58
59     @Override
60     public void configure(AuthorizationServerEndpointsConfigurer endpoints)
61         throws Exception {
62         endpoints.authenticationManager(this.authenticationManager).tokenStore(tokenStore());
63     }
64
65     @Override
66     public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
67         clients.jdbc(dataSource)
68             .withClient(clientId)
69             .authorizedGrantTypes(authorizedGrantTypes)
70             .authorities(Authorities.names())
71             .resourceIds(resourceIds)
72             .scopes(scopes)
73             .secret(secret)
74             .accessTokenValiditySeconds(accessTokenValiditySeconds);
75     }
76
77     @Bean
78     public PasswordEncoder passwordEncoder() {
79         if (encoder == null) {
80             encoder = new BCryptPasswordEncoder();
81         }
82         return encoder;
83     }
84 }
```

AuthorizationServerConfiguration.java hosted with ❤ by GitHub

[view raw](#)

Nessa classe vimos a anotação `@Configuration` que diz ao Spring que essa classe é uma classe de configuração e ela deverá ser instanciada na inicialização da aplicação. A anotação `@EnableAuthorizationServer` que habilita o `AuthorizationServer`, disponibilizando um **AuthorizationEndpoint** e um **TokenEndpoint**. Vimos também que injetamos nossas propriedades que definimos anteriormente. Temos dois métodos `configure` o primeiro define o gerenciador de autenticação do **AuthorizationEndpoint** e o segundo registra um **Client**, segundo as configurações que definimos.

## CorsFilterConfiguration

```
1 package com.coderef.delivery.config;
```

```

2
3  import org.springframework.boot.web.servlet.FilterRegistrationBean;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.Configuration;
6  import org.springframework.core.Ordered;
7  import org.springframework.web.cors.CorsConfiguration;
8  import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
9  import org.springframework.web.filter.CorsFilter;
10
11  @Configuration
12  public class CorsFilterConfiguration {
13
14      @Bean
15      public FilterRegistrationBean corsFilter() {
16          UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
17          CorsConfiguration config = new CorsConfiguration();
18          config.setAllowCredentials(true);
19          config.addAllowedOrigin("*");
20          config.addAllowedHeader("*");
21          config.addAllowedMethod("*");
22          source.registerCorsConfiguration("/**", config);
23          FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));
24          bean.setOrder(Ordered.HIGHEST_PRECEDENCE);
25          return bean;
26      }
27
28  }

```

CorsFilterConfiguration.java hosted with ❤ by GitHub

[view raw](#)

Esse `Filter` será invocado no início de uma requisição ao Servidor de autorização, ele permite que uma requisição chegue ao Servidor independente das **credenciais, origin, header e method**.

## ResourceServerConfiguration

```

1  package com.coderef.delivery.config;
2
3  import org.springframework.beans.factory.annotation.Value;
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 ResourceServerConfiguration extends ResourceServerConfigurerAdapter {
15
16
17      @Value("${security.oauth2.client.resource-ids}")
18      private String RESOURCE_ID;
19
20      @Override
21      public void configure(ResourceServerSecurityConfigurer resources) {
22          resources.resourceId(RESOURCE_ID);
23      }
24
25      @Override
26      public void configure(HttpSecurity http) throws Exception {
27          http.requestMatchers()
28              .antMatchers("/")
29              .and()

```

```

29         .and()
30         .authorizeRequests()
31         .anyRequest()
32         .authenticated()
33         .antMatchers(HttpMethod.GET, "/*").access("#oauth2.hasScope('read')")
34         .antMatchers(HttpMethod.OPTIONS, "/*").access("#oauth2.hasScope('read')")
35         .antMatchers(HttpMethod.POST, "/*").access("#oauth2.hasScope('write')")
36         .antMatchers(HttpMethod.PUT, "/*").access("#oauth2.hasScope('write')")
37         .antMatchers(HttpMethod.PATCH, "/*").access("#oauth2.hasScope('write')")
38         .antMatchers(HttpMethod.DELETE, "/*").access("#oauth2.hasScope('write')");
39     }
40 }

```

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

Aqui definimos a anotação `@EnableResourceServer` que habilita um `Filter` Spring Security que autentica as requisições com base no token OAuth2 recebido. Temos um método `configure` que restringe as requisições com base nos escopos disponíveis.

## WebSecurityConfig

```

1  package com.coderef.delivery.config;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.boot.autoconfigure.security.SecurityProperties;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7  import org.springframework.core.annotation.Order;
8  import org.springframework.security.authentication.AuthenticationManager;
9  import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
10 import org.springframework.security.config.annotation.web.builders.WebSecurity;
11 import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
12 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
13 import org.springframework.security.crypto.password.PasswordEncoder;
14
15 import com.coderef.delivery.service.UserDetailsService;
16
17 @Configuration
18 @EnableWebSecurity
19 @Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
20 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
21
22     @Autowired
23     private UserDetailsService userDetailsService;
24
25     @Autowired
26     PasswordEncoder passwordEncoder;
27
28     @Override
29     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
30         auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder);
31     }
32
33     @Override
34     @Bean
35     public AuthenticationManager authenticationManagerBean() throws Exception {
36         return super.authenticationManagerBean();
37     }
38
39     @Override
40     public void configure(WebSecurity web) throws Exception {
41         web.ignoring().antMatchers("/oauth/register");
42     }

```



```
43 }

WebSecurityConfig.java hosted with ❤ by GitHub view raw
```

O Spring já possui uma classe interna anotada com `@EnableWebSecurity` para sobrescrever as configurações dessa classe anotamos a nossa com a mesma anotação e colocamos também a anotação `@Order` com a propriedade `ACCESS_OVERRIDE_ORDER`. Definimos também dois métodos `configure` um responsável pela codificação da senha que será informada pelo **usuário a Aplicação Client** e outro informando ao Spring Security que torne o endpoint `/oauth/register` público.

## AuthController

```
1 package com.coderef.delivery.controller;
2
3 import java.security.Principal;
4
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 @RequestMapping("/")
10 public class AuthController {
11
12     @RequestMapping("/user")
13     public Principal getCurrentLoggedInUser(Principal user) {
14         return user;
15     }
16 }
```

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

Aqui implementamos uma `Controller` muito importante, pois ela será consultada sempre pelo Client para validar se o token informado ainda é válido, caso não tenha acesso ou o token informado seja inválido uma mensagem de não autorizado é retornada.

## Authorities

```
1 package com.coderef.delivery.domain;
2
3 public enum Authorities {
4
5     ROLE_USER,
6     ROLE_ADMIN;
7
8     public static String[] names() {
9         String[] names = new String[values().length];
10         for(int index = 0; index < values().length; index++) {
11             names[index] = values()[index].name();
12         }
13
14         return names;
15     }
16 }
```

Authorities.java hosted with ❤ by GitHub

[view raw](#)

Nesse **ENUM** colocamos as duas regras de acesso que estamos utilizando na aplicação.

## Authority

```
1 package com.coderef.delivery.model;
2
3 import java.io.Serializable;
4
5 import javax.persistence.Entity;
6 import javax.persistence.Id;
7 import javax.validation.constraints.NotNull;
8 import javax.validation.constraints.Size;
9
10 @Entity
11 public class Authority implements Serializable {
12
13     private static final long serialVersionUID = 1L;
14
15     @Id
16     @NotNull
17     @Size(min = 0, max = 50)
18     private String name;
19
20     public String getName() {
21         return name;
22     }
23
24     public void setName(String name) {
25         this.name = name;
26     }
27 }
```

Authority.java hosted with ❤ by GitHub

[view raw](#)

Aqui implementamos a a classe que representa nossa tabela de regras no banco de dados.

## User

```
1 package com.coderef.delivery.model;
2
3
4 import java.io.Serializable;
5 import java.util.Set;
6
7 import javax.persistence.Column;
8 import javax.persistence.Entity;
9 import javax.persistence.Id;
10 import javax.persistence.JoinColumn;
11 import javax.persistence.JoinTable;
12 import javax.persistence.ManyToMany;
13 import javax.validation.constraints.Size;
14
15 import org.hibernate.validator.constraints.Email;
16
17 @Entity
18 public class User implements Serializable {
```

```
19
20     private static final long serialVersionUID = 1L;
21
22     @Id
23     @Column(updatable = false, nullable = false)
24     private String username;
25
26     @Size(min = 0, max = 500)
27     private String password;
28
29     @Email
30     @Size(min = 0, max = 50)
31     private String email;
32
33     private boolean activated;
34
35     @Size(min = 0, max = 100)
36     @Column(name = "activationkey")
37     private String activationKey;
38
39     @Size(min = 0, max = 100)
40     @Column(name = "resetpasswordkey")
41     private String resetPasswordKey;
42
43     @ManyToMany
44     @JoinTable(
45         name = "user_authority",
46         joinColumns = @JoinColumn(name = "username"),
47         inverseJoinColumns = @JoinColumn(name = "authority"))
48     private Set<Authority> authorities;
49
50     public User() {
51     }
52
53     public User(String username, String password, String email,
54         boolean activated, String firstName, String lastName,
55         String activationKey, String resetPasswordKey,
56         Set<Authority> authorities) {
57         this.username = username;
58         this.password = password;
59         this.email = email;
60         this.activated = activated;
61         this.activationKey = activationKey;
62         this.resetPasswordKey = resetPasswordKey;
63         this.authorities = authorities;
64     }
65
66     public String getUsername() {
67         return username;
68     }
69
70     public void setUsername(String username) {
71         this.username = username;
72     }
73
74     public String getPassword() {
75         return password;
76     }
77
78     public void setPassword(String password) {
79         this.password = password;
80     }
81
82     public String getEmail() {
83         return email;
84     }
85
86     public void setEmail(String email) {
87         this.email = email;
```


```
88     }
89
90     public boolean isActivated() {
91         return activated;
92     }
93
94     public void setActivated(boolean activated) {
95         this.activated = activated;
96     }
97
98     public String getActivationKey() {
99         return activationKey;
100    }
101
102    public void setActivationKey(String activationKey) {
103        this.activationKey = activationKey;
104    }
105
106    public String getResetPasswordKey() {
107        return resetPasswordKey;
108    }
109
110    public void setResetPasswordKey(String resetPasswordKey) {
111        this.resetPasswordKey = resetPasswordKey;
112    }
113
114    public Set<Authority> getAuthorities() {
115        return authorities;
116    }
117
118    public void setAuthorities(Set<Authority> authorities) {
119        this.authorities = authorities;
120    }
121 }
```

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

Essa classe será a representação da tabela de usuários que criamos anteriormente via SQL onde ficarão salvas as credenciais dos usuários.

## AuthorityRepository

```
1  package com.coderef.delivery.repository;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4
5  import com.coderef.delivery.model.Authority;
6
7  public interface AuthorityRepository extends JpaRepository<Authority, String>{
8
9      Authority findByName(String name);
10
11 }
```

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

Esse repository será responsável pelas consultas das regras do usuário no banco de dados.

## UserRepository

```
1 package com.coderef.delivery.repository;
2
3 import java.util.Optional;
4
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.data.jpa.repository.Query;
7 import org.springframework.data.repository.query.Param;
8
9 import com.coderef.delivery.model.User;
10
11 public interface UserRepository extends JpaRepository<User, String> {
12
13     @Query("SELECT u FROM User u WHERE LOWER(u.username) = LOWER(:username)")
14     Optional<User> findByUsername(@Param("username") String username);
15
16 }
```

UserRepository.java hosted with ❤ by GitHub

[view raw](#)

Aqui definimos as consultas referentes aos usuários.

## UserDetailsService

```
1 package com.coderef.delivery.service;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5 import java.util.Optional;
6
7 import javax.transaction.Transactional;
8
9 import org.springframework.beans.factory.annotation.Autowired;
10 import org.springframework.security.core.GrantedAuthority;
11 import org.springframework.security.core.authority.SimpleGrantedAuthority;
12 import org.springframework.security.core.userdetails.UserDetails;
13 import org.springframework.security.core.userdetails.UsernameNotFoundException;
14 import org.springframework.stereotype.Service;
15
16 import com.coderef.delivery.model.Authority;
17 import com.coderef.delivery.model.User;
18 import com.coderef.delivery.repository.UserRepository;
19
20 @Service
21 @Transactional
22 public class UserDetailsService implements org.springframework.security.core.userdetails.UserDet
23
24     UserRepository userRepository;
25
26     @Override
27     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
28         return userRepository.findByUsername(username)
29                                     .map(user -> new org.springframework
30                                     .orElseThrow(() -> new UsernameN
31     }
32
33     @Autowired
34     public void setUserRepository(UserRepository userRepository) {
35         this.userRepository = userRepository;
36     }
37
38     private Collection<GrantedAuthority> getGrantedAuthorities(User user){
39     Collection<GrantedAuthority> grantedAuthorities = new ArrayList<>();
40     for (Authority authority : user.getAuthorities()) {
41         GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(authority.getName());
42         grantedAuthorities.add(grantedAuthority);
43     }
44 }
```

```
43         }
44
45         return grantedAuthorities;
46     }
47 }
```

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

Aqui implementamos um `@Service`. O Spring Security fornece uma interface utilizada internamente chamada `UserDetailsService` que precisamos implementar quando queremos alterar o comportamento do `UserDetail` utilizado.

## DeliveryAuthServerApplication

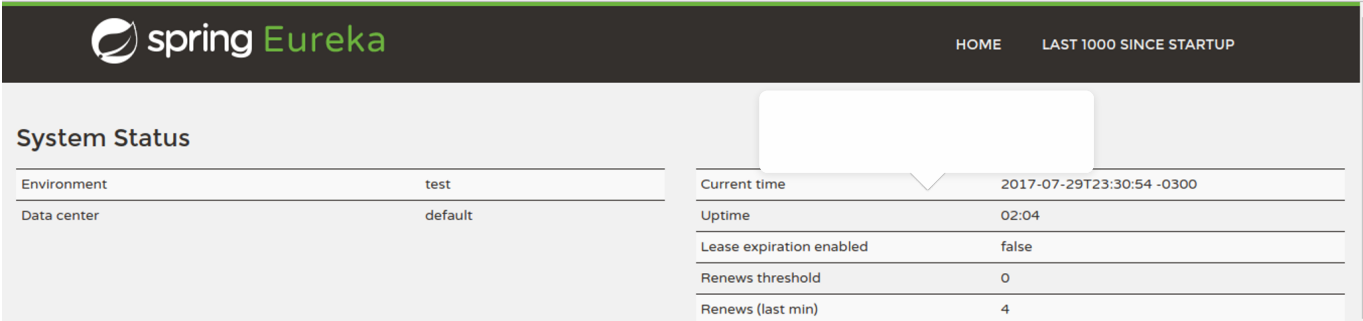
```
1 package com.coderef.delivery;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
6
7 @SpringBootApplication
8 @EnableEurekaClient
9 public class DeliveryAuthServerApplication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

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

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`.

## Subindo a aplicação

Agora chegou a hora subir o nosso Authorization Server para saber se tudo está funcionando corretamente. Antes de inciar nosso servidor de autorização será necessário subir o Config Server e depois o Eureka Server. Quando os dois estiverem online vamos executar a classe `DeliveryAuthServerApplication.java`. Finalizado, acesse o endereço `http://localhost:9091/` nosso servidor vai aparecer listado no Eureka Server:





DS Replicas

Instances currently registered with Eureka

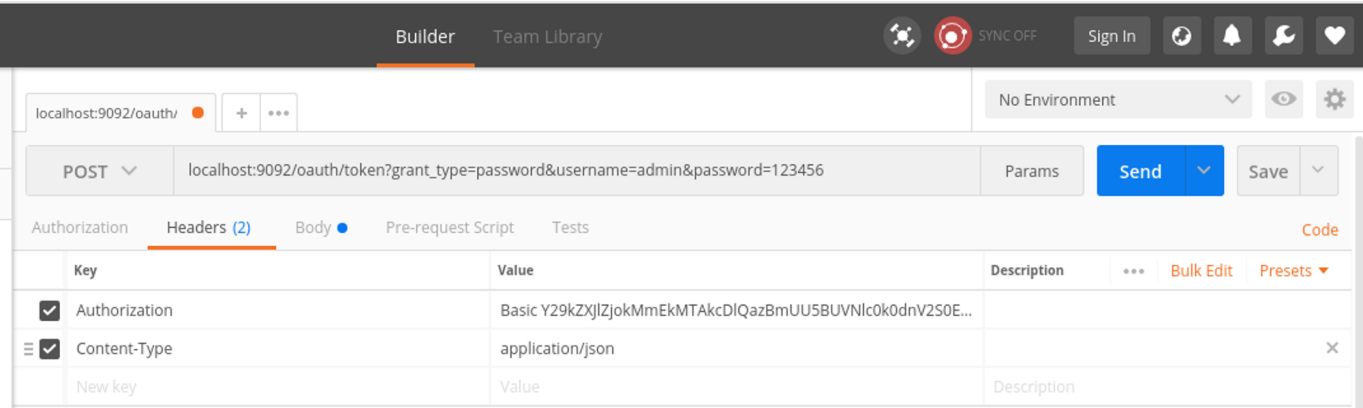
Application	AMIs	Availability Zones	Status
DELIVERY-AUTH-SERVER	n/a (1)	(1)	UP (1) - 192.168.42.84:delivery-auth-server:9092
DELIVERY-EUREKA-SERVER	n/a (1)	(1)	UP (1) - 192.168.42.84:delivery-eureka-server:9091

Solicitando um Token de Acesso

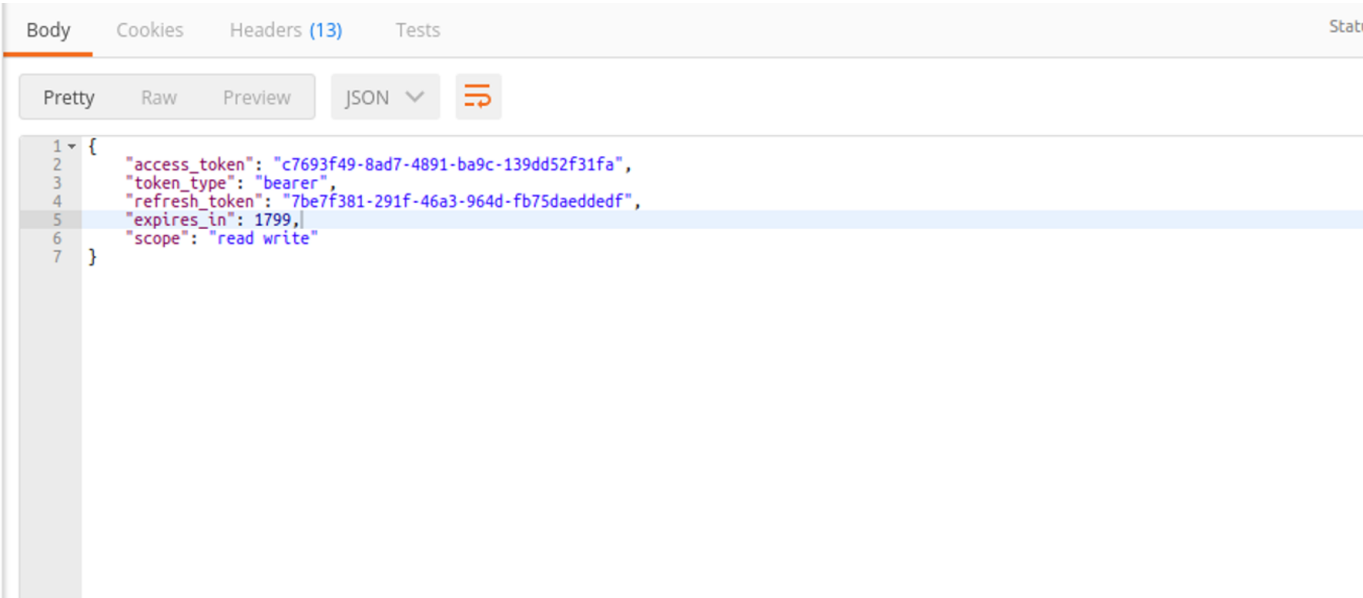
O framework Oauth2 define um padrão para a solicitação do token e um a resposta. Para solicitar um token o Client deverá estar registrado no servidor de autorização e deve enviar o `client-id` e o `secret` que configuramos no `application.yml` codificado no formato Base 64, respeitando o padrão `client-id:secret` . Para fazer essa requisição teremos que enviar os seguintes dados:

- **Authorization:** Basic  
Y29kZXJlZjokMmEkMTAkcdlQazBmUU5BUVNlc0k0dnV2S0EwT1phbkREMg== (verifique a imagem abaixo).
- **URL:** `http://localhost:9092/oauth/token?grant_type=password&username=admin&password=123456`
- **Method:** POST.
- **Content-Type:** `application/json`.

Para facilitar vamos utilizar o Postman, uma ferramenta poderosa com uma interface simples e fácil de manipular. Após instalar e abrir o Postman, os dados acima serão inseridos, tal como abaixo:



Após clicar em **Send** um json será recebido:



O OAuth2 tem um padrão de resposta, vou detalhar abaixo o que significa cada propriedade do JSON:

- **access\_token:** token de acesso que o Client irá utilizar para solicitar acesso aos recursos protegidos.
- **token\_type:** O tipo de token que o Client deverá enviar no Header das requisições para os recursos.
- **refresh\_token:** A cada interação entre client e o resource server esse token pode ser utilizado para renovar o token sendo utilizado atualmente. Isso evita que o token expire enquanto o usuário ainda está interagindo com o site.
- **expire\_in:** Tempo de vida do token, em segundos.
- **scope:** Escopo de permissão que o token terá para acessar os recursos.

## Repositórios do Projeto

Configurações: <https://github.com/rafaelcam/delivery-configs>

Projeto: <https://github.com/rafaelcam/delivery>

## Referências

<http://projects.spring.io/spring-security-oauth/>

<https://tools.ietf.org/pdf/draft-ietf-oauth-v2-31.pdf>

Thanks to João Rafael Campos da Silva.

[Spring Security](#)

[Spring Boot](#)

[OAuth2](#)

[Microservices](#)

[Authorization](#)

### 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](#)