Certified Tech Developer
The Ultimate Degree

Especialização em Back End I

Como implementar o OAuth?

Vamos ao passo-a-passo.

1. Primeiro, devemos nos registrar na API do provedor, que nos dará um URL (um clientId e um secret). No caso do Google, é o Console API do Google. Além de lhes dar isto, eles devem fornecer um endereço de callback para o qual o Google deve enviar as informações de login. Por exemplo: http://localhost:8081/login/oauth2/code/google. O caminho **/login/oauth2/code/ProviderName** é um padrão (a chamada ao localhost é para que eles possam testar localmente).

2. Na configuração **/src/main/resources/application.yml** você deve colocar o seguinte:

```
spring:
 security:
   oauth2:
     client:
       registration:
         google:
           client-id: your-google-client-id
       client-secret: your-google-client-secret
```

   Na prática, os secrets são armazenados em um servidor de configuração.

3. Na classe **SecurityConfig** (se quando eles geraram o projeto com Initialzr colocaram um módulo de segurança nele, eles já estão criados, caso contrário eles devem criá-lo). Aplique a seguinte alteração:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter
```

```
    @Override
    protected void configure(HttpSecurity http) throws
        http.authorizeRequests()
         .anyRequest().authenticated()
         .and()
         .oauth2Login();
    }
}
```

Deve-se notar que a última linha **oauth2Login()** é aquela que força, pelo uso de and(), a autenticação do fornecedor.

4. Agora precisamos obter informações do usuário, seja para mapeá-las para permissões dentro de nossa aplicação ou para utilizá-las.

```
      @Autowired
      private OAuth2AuthorizedClientService authorizedClientService;

      @GetMapping("/loginSuccess")
      public String getLoginInfo(Model model, OAuth2AuthenticationToken
      authentication) {
          OAuth2AuthorizedClient client = authorizedClientService
            .loadAuthorizedClient(
              authentication.getAuthorizedClientRegistrationId(),
                authentication.getName());
          String userInfoEndpointUri = client.getClientRegistration()
        .getProviderDetails().getUserInfoEndpoint().getUri();

      if (!StringUtils.isEmpty(userInfoEndpointUri)) {
          RestTemplate restTemplate = new RestTemplate();
          HttpHeaders headers = new HttpHeaders );
                  headers.add(HttpHeaders.AUTHORIZATION, "Bearer " +
      client.getAccessToken()
            .getTokenValue());
          HttpEntity entity = new HttpEntity "", headers);
          ResponseEntity <map>response = restTemplate
                  .exchange(userInfoEndpointUri, HttpMethod.GET, entity,
      Map.class);
          Map userAttributes = response.getBody();
          model.addAttribute "name", userAttributes.get("name"));
      }
}
```

O objeto **OAuth2AuthenticationToken** viaja com os pedidos injetados pelo Spring,

o que já nos traz o **getAuthorizedClientRegistrationId()**, que é suficiente para mapeá-lo em nosso esquema de permissões. Entretanto, se quisermos mais informações, o objeto nos traz o **userInfoEndpointUri,** que a partir da solicitação nos permite trazer as informações "expostas" para mapeá-lo e usá-lo como apropriado.

5. Uma vez terminado o exposto acima, procedemos à configuração do **SecurityContext** (context que contém todas as informações de autenticação). Em nosso caso, como se trata de uma aplicação web, devemos controlar a segurança de cada solicitação recebida. Neste momento, já configuramos o OAuth, mas agora devemos anexar nosso context de segurança a cada solicitação. Considerando que devemos autenticar cada solicitação, o único ponto do sistema onde passam todas as solicitações que eventualmente serão encaminhadas é apenas no **gateway**, e elas são tratadas através de uma cadeia de filtros. Por isso, é lógico que haverá um filtro que também tratará da autenticação. Revendo a explicação anterior de como funciona a autenticação, vemos que em cada solicitação receberemos um ou mais headers que terão as informações de autenticação necessárias para serem processadas.

6. Neste ponto, o **SecurityContext** está configurado. Este context conterá todas as informações do usuário autenticado, informações que serão utilizadas para verificações de segurança. Este context tem informações suficientes para suportar o OAuth contra um aplicativo de desktop ou console. Em nosso caso, um website com solicitações, temos que validar cada uma delas. Se quiséssemos validar a segurança em um único ponto - em cada rota que chega até nós - o ponto onde conseguimos o lugar para modificar é o **Spring Cloud Gateway.** Voltando um pouco, o Spring Cloud Gateway era usado para lidar com uma série de filtros, portanto, devemos configurar um filtro especial que aceite uma solicitação com um certo token.

```
spring:
  application:
    name: gateway
  cloud:
    gateway:
      default-filters:
        - TokenRelay
```


```
routes:
  - id: XXXXX
    uri: XXXXXXXX
    predicates:
      - Path=/XXXXXX/**
    filters:
 - RemoveRequestHeader=Cookie
```

Quando olhamos para o API Gateway, aprendemos como adicionar rotas, predicados e alguns filtros especiais. Agora, vamos adicionar um filtro especial que percorre todas as rotas. Dentro dos **default-filters** adicionamos o filtro **TokenRelay.** Toda a implementação na parte traseira é gerada pela Spring. O Spring Security se encarrega de verificar se o token é válido, mas estamos interessados no fato de que, com as informações deste token, podemos solicitar informações ao fornecedor de segurança.

Se voltarmos ao tópico "Como funciona a autenticação web?", você pode ver que a cada solicitação ao servidor é enviado um token na tag de Autorização e, no lado do servidor, este token é verificado. A implementação da Spring funciona da seguinte forma: recebemos uma solicitação, o Spring Security "captura" e então ele passa por uma série de filtros encadeados que acabam determinando as rotas. Estes filtros são, em última instância, funções que recebem a solicitação e, se esta for afetada pelo filtro, rejeita a operação ou é modificada; caso contrário, ela continua com os outros filtros. O filtro **TokenRelay** leva o token de autenticação que chega na solicitação ao gateway (verificado pelo Spring Security) e o passa para o resto da cadeia de filtros a fim de extrair informações do mesmo, fazer mais verificações de segurança ou encaminhar o token de autenticação para outros serviços.