Certified Tech Developer
The Ultimate Degree

Especialización en Back End I

¿Cómo implementar OAuth?

Conozcamos el paso a paso.

1. Primero debemos registrarnos en la API del proveedor, el cual nos dará una URL (un clientId y un secret). En el caso de Google es Google API Console. Al margen de darles esto, deben proveer una dirección de callback a donde Google debe enviar la información de login. Por ejemplo: http://localhost:8081/login/oauth2/code/google. La ruta **/login/oauth2/code/NombreDelProveedor** es un estándar (la llamada a localhost es para que puedan hacer pruebas locales).

2. En la configuración **/src/main/resources/application.yml** deben poner lo siguiente:

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

   En la práctica los secretos se guardan en un servidor de configuración.

3. En la clase **SecurityConfig** (si cuando generaron el proyecto con Initialzr le pusieron algún módulo de seguridad, ya se las crea, si no deberán crearla). Aplican el siguiente cambio:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter

    @Override
    protected void configure(HttpSecurity http) throws
```

```
        http.authorizeRequests()
          .anyRequest().authenticated()
          .and()
          .oauth2Login();
    }
}
```

Cabe destacar que la última línea **oauth2Login()** es la que fuerza, por el uso del and(), la autenticación del proveedor.

4. Ahora debemos obtener información del usuario, ya sea para mapearlas con permisos dentro de nuestra aplicación o bien para utilizarla.

```
    @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"));
    }
}
```

El objeto **OAuth2AuthenticationToken** viaja con las requests inyectado por Spring, que ya nos trae el **getAuthorizedClientRegistrationId()**, con eso es suficiente para

mapearlo en nuestro esquema de permisos. No obstante, si queremos más información, el objeto nos trae **userInfoEndpointUri** que a partir de request nos permite traer la info “expuesta” para mapearla y utilizarla según el caso.

5. Una vez terminado lo anterior, procedemos a configurar el **SecurityContext** (context que contiene toda la info de autenticación). En nuestro caso, como es una aplicación web, debemos controlar la seguridad en cada request recibido. Ya en este punto tenemos configurado OAuth, pero ahora debemos unir nuestro contexto de seguridad a cada request. Considerando que debemos autenticar cada request, el único punto del sistema donde pasan todas las requests que eventualmente serán ruteadas es justamente el **gateway** y las peticiones se manejan a través de una cadena de filtros. Por lo tanto, es razonable pensar que habrá un filtro que gestione la autenticación también. Repasando la explicación anterior del funcionamiento de la autenticación, vemos que en cada request recibiremos uno o varios headers que tendrán la información necesaria de autenticación para ser procesada.

6. En este punto, se realiza la configuración del **SecurityContext**. Este context contendrá toda la información del usuario autenticado, información que se utilizará para los chequeos de seguridad. Este context tiene información suficiente para dar soporte de OAuth contra una app desktop o de consola. En nuestro caso, una web con requests, tenemos que validar cada una. Si quisiéramos validar la seguridad en un solo punto —en cada ruta que nos llega—, el punto donde nos llega el lugar a modificar es **Spring Cloud Gateway**. Repasando un poco, Spring Cloud Gateway manejaba una serie de filtros, entonces, deberíamos configurar algún filtro especial que acepte un request con un determinado token.

```
spring:
  application:
    name: gateway
  cloud:
    gateway:
      default-filters:
        - TokenRelay
      routes:
        - id: XXXXX
          uri: XXXXXXXX
          predicates:
```


```
            - Path=/XXXXXX/**
          filters:
        - RemoveRequestHeader=Cookie
```

Cuando vimos API Gateway, aprendimos cómo agregar rutas, predicados y algunos filtros especiales. Ahora, vamos a agregar un filtro especial que pasa por todas las rutas. Dentro de **default-filters** agregamos el filtro **TokenRelay**. Toda la implementación por detrás la genera Spring. Spring Security se encarga de verificar que el token sea válido, pero a nosotros nos interesa que con la información de este token podemos pedirle información al proveedor de seguridad.

Si volvemos hacia el tema “¿Cómo funciona la autenticación web?”, podemos observar que en cada request contra el server se envía un token en el tag Authorization y, del lado del server, este Token se verifica. La implementación de Spring funciona de la siguiente manera: nos llega un request, lo “captura” Spring Security y luego este pasa por una serie de filtros encadenados que terminan determinando las rutas. Estos filtros en definitiva son funciones que reciben el request y, si el mismo es afectado por el filtro, rechaza la operación o se modifica el request; caso contrario, se prosigue con los demás filtros. El filtro del **TokenRelay** lo que hace es tomar el token de autenticación que llega en el request al gateway (verificado por Spring Security) y lo pasa al resto de la cadena de filtros para poder extraer información del mismo, hacer más chequeos de seguridad o para forwardear el token de autenticación a otros servicios.