

<http://baeldung.com>

# Spring Security 5 - Acceso OAuth2

Última modificación: 18 de marzo de 2018

por Loredana Crusoveanu (<http://www.baeldung.com/author/loredana-crusoveanu/>)  
(<http://www.baeldung.com/author/loredana-crusoveanu/>)

**Seguridad** (<http://www.baeldung.com/category/security-2/>)

**Primavera** (<http://www.baeldung.com/category/spring/>) +

**OAuth** (<http://www.baeldung.com/tag/oauth/>)

Acabo de anunciar los nuevos módulos de *Spring 5* en REST With Spring:

**>> COMPRUEBA EL CURSO** (</rest-with-spring-course#new-modules>)

## 1. Información general

Spring Security 5 introduce una nueva clase *OAuth2LoginConfigurer* que podemos usar para configurar un servidor de autorizaciones externo.

En este artículo, **exploraremos algunas de las diversas opciones de configuración disponibles para el elemento *oauth2Login()***.

## 2. Dependencias Maven

Además de las dependencias estándar de Spring y Spring Security, también necesitaremos agregar las dependencias *spring-security-oauth2-client* (<https://search.maven.org/#search%7Cga%7C1%7Ca%3A%22spring-security-oauth2-client%22%20AND%20g%3A%22org.springframework.security%22>) y *spring-security-oauth2-jose* (<https://search.maven.org/#search%7Cga%7C1%7Ca%3A%22spring-security-oauth2-jose%22>):

```
1 <dependency>
2   <groupId>org.springframework.security</groupId>
3   <artifactId>spring-security-oauth2-client</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.security</groupId>
7   <artifactId>spring-security-oauth2-jose</artifactId>
8 </dependency>
```

En nuestro ejemplo, las dependencias son administradas por el padre de arranque Spring Boot, versión *2.0.0.M7*, que corresponde a la versión *5.0.0.RELEASE* de los artefactos de Spring Security.

Por ahora, dado que estamos usando una versión clave de Spring Boot, también necesitaremos agregar el repositorio:

```
1 <repositories>
2   <repository>
3     <id>spring-milestones</id>
4     <name>Spring Milestones</name>
5     <url>https://repo.spring.io/milestone (https://repo.spring.io/milestone)</url>
6     <snapshots>
7       <enabled>false</enabled>
8     </snapshots>
9   </repository>
10 </repositories>
```

### 3. Configuración de los clientes

En un proyecto de Spring Boot, todo lo que tenemos que hacer es agregar algunas propiedades estándar para cada cliente que deseamos configurar.

**Establezcamos nuestro proyecto para iniciar sesión con clientes registrados con Google y Facebook como proveedores de autenticación.**

#### 3.1. Obtención de credenciales de cliente

Para obtener las credenciales del cliente para la autenticación de Google OAuth2, dirijase a la consola API de Google (<https://console.developers.google.com/>) : sección "Credenciales".

Aquí crearemos credenciales de tipo "ID de cliente OAuth2" para nuestra aplicación web. Esto da como resultado que Google configure una identificación de cliente y un secreto para nosotros.

También tenemos que configurar un URI de redirección autorizado en la Consola de Google, que es la ruta a la que los usuarios serán redirigidos después de que inicien sesión con éxito en Google.

De forma predeterminada, Spring Boot configura este URI de redirección como `/login/oauth2/client/{clientId}`. Por lo tanto, para Google agregaremos el URI:

```
1 http://localhost:8081/login/oauth2/client/google (http://localhost:8081/login/oauth2/client/google)
```

To obtain the client credentials for authentication with Facebook, we need to register an application on the Facebook for Developers (<https://developers.facebook.com/docs/facebook-login>) website and set up the corresponding URI as a "Valid OAuth redirect URI":

```
1 http://localhost:8081/login/oauth2/client/facebook (http://localhost:8081/login/oauth2/client/facebook)
```

#### 3.3. Security Configuration

Next, we need to add client credentials in the *application.properties* file. **The Spring Security properties are prefixed with "spring.security.oauth2.client.registration" followed by the client name, then the name of the client property:**

```
1 spring.security.oauth2.client.registration.google.client-id=<your client id>
2 spring.security.oauth2.client.registration.google.client-secret=<your client secret>
3
4 spring.security.oauth2.client.registration.facebook.client-id=<your client id>
5 spring.security.oauth2.client.registration.facebook.client-secret=<your client secret>
```

Adding these properties for at least one client will enable the *Oauth2ClientAutoConfiguration* class which sets up all the necessary beans.

La configuración de seguridad web automática es equivalente a definir un elemento simple *oauth2Login()*:

```

1  @Configuration
2  public class SecurityConfig extends WebSecurityConfigurerAdapter {
3
4      @Override
5      protected void configure(HttpSecurity http) throws Exception {
6          http.authorizeRequests()
7              .anyRequest().authenticated()
8              .and()
9              .oauth2Login();
10     }
11 }

```

Aquí, podemos ver que el elemento *oauth2Login()* se usa de manera similar a los elementos ya conocidos *httpBasic()* y *formLogin()*.

**Ahora, cuando intentamos acceder a una URL protegida, la aplicación mostrará una página de inicio de sesión autogenerada con dos clientes:**

## Login with OAuth 2.0

[Facebook](#)

[Google](#)

## 3.4. Otros clientes

Tenga en cuenta que, además de Google y Facebook, el proyecto Spring Security también contiene configuraciones predeterminadas para GitHub y Okta. Estas configuraciones predeterminadas brindan toda la información necesaria para la autenticación, que es lo que nos permite ingresar solo las credenciales del cliente.

Si queremos utilizar un proveedor de autenticación diferente no configurado en Spring Security, necesitaremos definir la configuración completa, con información como URI de autorización y URI de token. Aquí hay (<https://github.com/spring-projects/spring-security/blob/gf6af4f3b8e79a8a45369d99862cd3d96f4083ce/config/src/main/java/org/springframework/security/config/oa>) un vistazo a las configuraciones predeterminadas en Spring Security para tener una idea de las propiedades necesarias.

## 4. Configuración en un proyecto sin arranque

### 4.1. Creando un Bean *ClientRegistrationRepository*

Si no estamos trabajando con una aplicación Spring Boot, necesitaremos definir un bean *ClientRegistrationRepository* que contenga una representación interna de la información del cliente propiedad del servidor de autorización:

```

1  @Configuration
2  @EnableWebSecurity
3  @PropertySource("classpath:application.properties")
4  public class SecurityConfig extends WebSecurityConfigurerAdapter {
5      private static List<String> clients = Arrays.asList("google", "facebook");
6
7      @Bean
8      public ClientRegistrationRepository clientRegistrationRepository() {
9          List<ClientRegistration> registrations = clients.stream()
10              .map(c -> getRegistration(c))
11              .filter(registration -> registration != null)
12              .collect(Collectors.toList());
13
14          return new InMemoryClientRegistrationRepository(registrations);
15     }
16 }

```

Aquí estamos creando *InMemoryClientRegistrationRepository* con una lista de objetos *ClientRegistration*.

### 4.2. Construir objetos de *registro de clientes*

Veamos el método *getRegistration()* que crea estos objetos:

```

1  private static String CLIENT_PROPERTY_KEY
2  = "spring.security.oauth2.client.registration.";
3
4  @Autowired
5  private Environment env;
6
7  private ClientRegistration getRegistration(String client) {
8      String clientId = env.getProperty(
9          CLIENT_PROPERTY_KEY + client + ".client-id");
10
11     if (clientId == null) {
12         return null;
13     }
14
15     String clientSecret = env.getProperty(
16         CLIENT_PROPERTY_KEY + client + ".client-secret");
17
18     if (client.equals("google")) {
19         return CommonOAuth2Provider.GOOGLE.getBuilder(client)
20             .clientId(clientId).clientSecret(clientSecret).build();
21     }
22     if (client.equals("facebook")) {
23         return CommonOAuth2Provider.FACEBOOK.getBuilder(client)
24             .clientId(clientId).clientSecret(clientSecret).build();
25     }
26     return null;
27 }

```

Aquí, estamos leyendo las credenciales del cliente de un archivo *application.properties* similar , y luego usamos la enumeración *CommonOAuth2Provider* ya definida en Spring Security para el resto de las propiedades del cliente para los clientes de Google y Facebook.

Cada instancia de *ClientRegistration* corresponde a un cliente.

### 4.3. Registrar el *ClientRegistrationRepository*

Finalmente, tenemos que crear un bean *OAuth2AuthorizedClientService* basado en el bean *ClientRegistrationRepository* y registrar ambos con el elemento *oauth2Login()*:

```

1  @Override
2  protected void configure(HttpSecurity http) throws Exception {
3      http.authorizeRequests().anyRequest().authenticated()
4          .and()
5          .oauth2Login()
6          .clientRegistrationRepository(clientRegistrationRepository())
7          .authorizedClientService(authorizedClientService());
8  }
9
10 @Bean
11 public OAuth2AuthorizedClientService authorizedClientService() {
12
13     return new InMemoryOAuth2AuthorizedClientService(
14         clientRegistrationRepository());
15 }

```

Como lo demuestra aquí, **podemos utilizar el *clientRegistrationRepository()* método de *oauth2Login()* para registrar un repositorio de registro personalizado.**

También tendremos que definir una página de inicio de sesión personalizada, ya que no se generará automáticamente. Veremos más información sobre esto en la próxima sección.

Continuemos con una mayor personalización de nuestro proceso de inicio de sesión.

## 5. Personalizar *oauth2Login()*

Hay varios elementos que utiliza el proceso OAuth 2 y que podemos personalizar utilizando los métodos *oauth2Login()*.

**Tenga en cuenta que todos estos elementos tienen configuraciones predeterminadas en Spring Boot y no se requiere una configuración explícita.**

Veamos cómo podemos personalizarlos en nuestra configuración.

## 5.1. Página de inicio de sesión personalizada

Aunque Spring Boot genera una página de inicio de sesión predeterminada para nosotros, generalmente queremos definir nuestra propia página personalizada.

**Comencemos configurando una nueva URL de inicio de sesión para el elemento `oauth2Login()` utilizando el método `loginPage()`:**

```
1  @Override
2  protected void configure(HttpSecurity http) throws Exception {
3      http.authorizeRequests()
4          .antMatchers("/oauth_login")
5          .permitAll()
6          .anyRequest()
7          .authenticated()
8          .and()
9          .oauth2Login()
10         .loginPage("/oauth_login");
11 }
```

Aquí, hemos configurado nuestra URL de inicio de sesión para ser `/oauth_login`.

A continuación, definamos un `LoginController` con un método que se asigna a esta URL:

```
1  @Controller
2  public class LoginController {
3
4      private static String authorizationRequestBaseUri
5          = "oauth2/authorization";
6      Map<String, String> oauth2AuthenticationUrls
7          = new HashMap<>();
8
9      @Autowired
10     private ClientRegistrationRepository clientRegistrationRepository;
11
12     @GetMapping("/oauth_login")
13     public String getLoginPage(Model model) {
14         // ...
15
16         return "oauth_login";
17     }
18 }
```

**Este método debe enviar un mapa de los clientes disponibles y sus puntos finales de autorización a la vista**, que obtendremos del bean `ClientRegistrationRepository`:

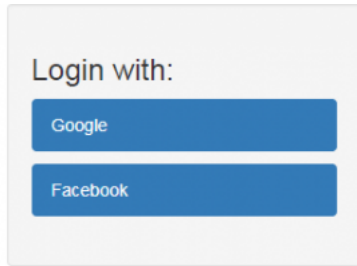
```
1  public String getLoginPage(Model model) {
2      Iterable<ClientRegistration> clientRegistrations = null;
3      ResolvableType type = ResolvableType.forInstance(clientRegistrationRepository)
4          .as(Iterable.class);
5      if (type != ResolvableType.NONE &&
6          ClientRegistration.class.isAssignableFrom(type.resolveGenerics()[0])) {
7          clientRegistrations = (Iterable<ClientRegistration>) clientRegistrationRepository;
8      }
9
10     clientRegistrations.forEach(registration ->
11         oauth2AuthenticationUrls.put(registration.getClientName(),
12             authorizationRequestBaseUri + "/" + registration.getRegistrationId());
13     model.addAttribute("urls", oauth2AuthenticationUrls);
14
15     return "oauth_login";
16 }
```

Finalmente, debemos definir nuestra página `oauth_login.html`:

```
1  <h3>Login with:</h3>
2  <p th:each="url : ${urls}">
3      <a th:text="${url.key}" th:href="${url.value}">Client</a>
4  </p>
```

Esta es una página HTML simple que muestra enlaces para autenticarse con cada cliente.

Después de agregarle un poco de estilo, podemos tener una página de inicio de sesión mucho más agradable:



## 5.2. Comportamiento personalizado de éxito y falla de autenticación

Podemos controlar el comportamiento posterior a la autenticación mediante el uso de diferentes métodos:

- *defaultSuccessUrl()* y *failureUrl()* - para redirigir al usuario a una URL determinada
- *successHandler()* y *failureHandler()* - para ejecutar la lógica personalizada siguiendo el proceso de autenticación

Veamos cómo podemos establecer URL personalizadas para redirigir al usuario a:

```
1 .oauth2Login()
2 .defaultSuccessUrl("/loginSuccess")
3 .failureUrl("/loginFailure");
```

Si el usuario visitó una página segura antes de autenticarse, se le redirigirá a esa página después de iniciar sesión; de lo contrario, serán redirigidos a */loginSuccess*.

Si queremos que el usuario siempre se envíe a la URL */loginSuccess*, independientemente de si estaban en una página segura antes o no, podemos usar el método *defaultSuccessUrl("/loginSuccess", true)*.

Para usar un controlador personalizado, tendríamos que crear una clase que implemente las interfaces *AuthenticationSuccessHandler* o *AuthenticationFailureHandler*, anule los métodos heredados, luego configure los beans con los métodos *successHandler()* y *failureHandler()*.

## 5.3. Punto final de autorización personalizado

El punto final de autorización es el punto final que Spring Security usa para activar una solicitud de autorización al servidor externo.

Primero, **establezcamos nuevas propiedades para el punto final de la autorización** :

```
1 .oauth2Login()
2 .authorizationEndpoint()
3 .baseUri("/oauth2/authorize-client")
4 .authorizationRequestRepository(authorizationRequestRepository());
```

Aquí, hemos modificado el *baseUri* en */oauth2/authorize-client* en lugar de la *autorización /oauth2* predeterminada. También estamos estableciendo explícitamente un bean *authorizationRequestRepository()* que debemos definir:

```
1 @Bean
2 public AuthorizationRequestRepository<OAuth2AuthorizationRequest>
3     authorizationRequestRepository() {
4
5     return new HttpSessionOAuth2AuthorizationRequestRepository();
6 }
```

En nuestro ejemplo, hemos utilizado la implementación provista por Spring para nuestro bean, pero también pudimos proporcionar una personalizada.

## 5.4. Punto final de Token personalizado

El *punto final del token* procesa los tokens de acceso.

Configuremos explícitamente ***tokenEndpoint()*** con la implementación predeterminada del cliente de respuesta:

```
1 .oauth2Login()
2 .tokenEndpoint()
3 .accessTokenResponseClient(accessTokenResponseClient());
```

Y aquí está el bean cliente de respuesta:

```
1 @Bean
2 public OAuth2AccessTokenResponseClient<OAuth2AuthorizationCodeGrantRequest>
3     accessTokenResponseClient() {
4
5     return new NimbusAuthorizationCodeTokenResponseClient();
6 }
```

Esta configuración es la misma que la predeterminada y está utilizando la implementación de Spring, que se basa en el intercambio de un código de autorización con el proveedor.

Por supuesto, también podríamos sustituir un cliente de respuesta personalizado.

## 5.5. Punto final de redireccionamiento personalizado

Este es el punto final para redirigir a después de la autenticación con el proveedor externo.

**Veamos cómo podemos cambiar la *baseUri* para el punto final de redirección:**

```
1 .oauth2Login()
2 .redirectionEndpoint()
3 .baseUri("/oauth2/redirect")
```

El URI predeterminado es *login / oauth2 / client*.

Tenga en cuenta que si lo cambiamos, también debemos actualizar la propiedad *redirectUriTemplate* de cada *registro* de *cliente* y agregar el nuevo URI como un URI de redirección autorizado para cada cliente.

## 5.6. Punto final de información de usuario personalizado

El punto final de información del usuario es la ubicación que podemos aprovechar para obtener información del usuario.

**Podemos personalizar este punto final utilizando el método *userInfoEndpoint()***. Para esto, podemos usar métodos como *userService()* y *customUserType()* para modificar la forma en que se recupera la información del usuario.

## 6. Accediendo a la información del usuario

Una tarea común que podemos querer lograr es encontrar información sobre el usuario que ha iniciado sesión. Para esto, **podemos hacer una solicitud al punto final de la información del usuario**.

Primero, tendremos que obtener el cliente correspondiente al token de usuario actual:

```
1 @Autowired
2 private OAuth2AuthorizedClientService authorizedClientService;
3
4 @GetMapping("/loginSuccess")
5 public String getLoginInfo(Model model, OAuth2AuthenticationToken authentication) {
6     OAuth2AuthorizedClient client = authorizedClientService
7         .loadAuthorizedClient(
8             authentication.getAuthorizedClientRegistrationId(),
9             authentication.getName());
10    //...
11    return "loginSuccess";
12 }
```

A continuación, enviaremos una solicitud al punto final de información del usuario del cliente y recuperaremos el *mapa de UserAttributes*:

```

1 String userInfoEndpointUri = client.getClientRegistration()
2   .getProviderDetails().getUserInfoEndpoint().getUri();
3
4 if (!StringUtils.isEmpty(userInfoEndpointUri)) {
5     RestTemplate restTemplate = new RestTemplate();
6     HttpHeaders headers = new HttpHeaders();
7     headers.add(HttpHeaders.AUTHORIZATION, "Bearer " + client.getAccessToken()
8       .getTokenValue());
9     HttpEntity entity = new HttpEntity("", headers);
10    ResponseEntity<Map>response = restTemplate
11      .exchange(userInfoEndpointUri, HttpMethod.GET, entity, Map.class);
12    Map userAttributes = response.getBody();
13    model.addAttribute("name", userAttributes.get("name"));
14 }

```

Al agregar la propiedad de *nombre* como un atributo de *modelo*, podemos mostrarlo en la vista *loginSuccess* como un mensaje de bienvenida para el usuario:

Welcome, Loredana Crusoveanu!

Además del *nombre*, la *userattributes* *mapa* también contiene propiedades tales como *correo electrónico*, *FAMILY\_NAME*, la *imagen*, la *configuración regional*.

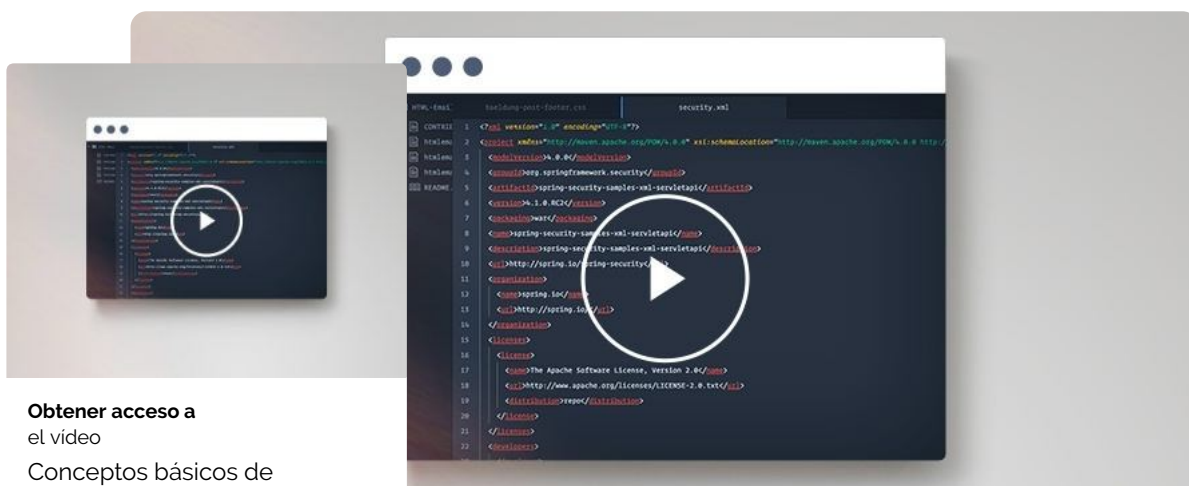
## 7. Conclusión

En este artículo, hemos visto cómo podemos usar el elemento *oauth2Login()* en Spring Security para autenticar con diferentes proveedores como Google y Facebook. También hemos pasado por algunos escenarios comunes de personalización de este proceso.

El código fuente completo de los ejemplos se puede encontrar en GitHub (<https://github.com/eugenp/tutorials/tree/master/spring-5>).

## Acabo de anunciar los nuevos módulos de Spring 5 en REST With Spring:

>> VERIFIQUE LAS LECCIONES (/rest-with-spring-course#new-modules)



Obtener acceso a  
el video  
Conceptos básicos de  
seguridad para una API REST

Email Address

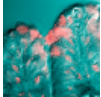
Tener acceso

Learn the basics of securing a REST API with Spring





# Get access to the video lesson!

[Access >>](#)☒ Subscribe ▾[▲ newest](#) [▲ oldest](#) [▲ most voted](#)

Guest

Seb



Say I want to have a users table in my database so I can store information about my users. Could be favorite color or similar. How can I get a unique identifier for the authenticated user regardless of oauth provider?

[+ 0 -](#)[🕒 2 months ago](#) [^](#)Loredana Crusoveanu (<http://www.baeldung.com/author/loredana-crusoveanu/>)

(<http://www.baeldung.com/author/loredana-crusoveanu/>)

Author

That information is included in the userInfo endpoint. Each ClientRegistration has an additional userNameAttributeName property that represents the name of the attribute containing the unique identifier in the response. For Google, the name of this attribute is "sub", for Facebook it's "id". These are auto-configured by Spring Boot.

These properties contain a unique identifier for the user among the provider's user accounts.

For example, here's more info on Google's userInfo endpoint response:

<https://developers.google.com/identity/protocols/OpenIDConnect#obtainuserinfo>

(<https://developers.google.com/identity/protocols/OpenIDConnect#obtainuserinfo>)

[+ 0 -](#)[🕒 2 months ago](#)

Guest

Hans Desmet (<http://www.vdab.be>)

How do you assign (MANAGER,ADMIN,SECRETARY" roles to the logged-in user ?

[+ 0 -](#)[🕒 2 months ago](#) [^](#)Loredana Crusoveanu (<http://www.baeldung.com/author/loredana-crusoveanu/>)

(<http://www.baeldung.com/author/loredana-crusoveanu/>)

Author

That's an interesting question. The user info you get does contain an authorities object as well. If you want to add other authorities of your own, then I think the best way would be to extend the OAuth2LoginAuthenticationFilter and add your own roles at the end.

[+ 0 -](#)[🕒 2 months ago](#) [^](#)

Vincent YSMAL



Guest

Not sure it's the right approach, but I've added some custom OidcUserService where you can add your own GrantedAuthority :

`http.oauth2Login()`

```
.userInfoEndpoint().oidcUserService(new OidcUserService() {
```

```
@Override
```

```
public OidcUser loadUser(OidcUserRequest userRequest) throws OAuth2AuthenticationException {
```

```
DefaultOidcUser oidcUser = (DefaultOidcUser) super.loadUser(userRequest);
```

```
...
```

```
}
```

[+ 0 -](#)[🕒 26 days ago](#)

Jun Huh




Is it possible to add openid connect server to spring?

Guest

+ 0 -

🕒 1 month ago ^



Loredana Crusoveanu (<http://www.baeldung.com/author/loredana-crusoveanu/>)

🔗 🔗


(<http://www.baeldung.com/author/loredana-crusoveanu/>)

Author

Sure, we have an article on openid here: <http://www.baeldung.com/spring-security-openid-connect> (<http://www.baeldung.com/spring-security-openid-connect>)

+ 0 -

🕒 1 month ago



Guest


Turgos

🔗 🔗

Do you have an example of sample Auth Server for Spring Security 5? Another Baeldung Article at page (<http://www.baeldung.com/sso-spring-security-oauth2> (<http://www.baeldung.com/sso-spring-security-oauth2>)) shows both the "Client App" and the "Auth Server", but it is earlier than Spring Security 5, Do you have the similar with Spring Security 5? Thank you in advance.

+ 0 -

🕒 1 month ago ^



Loredana Crusoveanu (<http://www.baeldung.com/author/loredana-crusoveanu/>)

🔗 🔗

(<http://www.baeldung.com/author/loredana-crusoveanu/>)

Author

We don't have a dedicated article with Spring 5, but there shouldn't be too many differences for a standard setup.

There's a resource server with Spring 5 in this repository if it helps: <https://github.com/Baeldung/spring-security-oauth> (<https://github.com/Baeldung/spring-security-oauth>)

La diferencia principal es agregar el Verificador de reclamos en la primavera 5, que se detalla aqui: <http://www.baeldung.com/spring-security-oauth-2-verify-claims> (<http://www.baeldung.com/spring-security-oauth-2-verify-claims>)

+ 0 -

🕒 Hace 1 mes

- PRIMAVERA ([HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/](http://www.baeldung.com/category/spring/))
- DESCANSO ([HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/](http://www.baeldung.com/category/rest/))
- JAVA ([HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/](http://www.baeldung.com/category/java/))
- SEGURIDAD ([HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/](http://www.baeldung.com/category/security-2/))
- PERSISTENCIA ([HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/](http://www.baeldung.com/category/persistence/))
- JACKSON ([HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/](http://www.baeldung.com/category/jackson/))
- HTTPCLIENT ([HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/](http://www.baeldung.com/category/http/))
- KOTLIN ([HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/](http://www.baeldung.com/category/kotlin/))

SERIE

- TUTORIAL "VOLVER A LO BÁSICO" DE JAVA ([HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL](http://www.baeldung.com/java-tutorial))
- JACKSON JSON TUTORIAL ([HTTP://WWW.BAELDUNG.COM/JACKSON](http://www.baeldung.com/jackson))
- TUTORIAL DE HTTPCLIENT 4 ([HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE](http://www.baeldung.com/httpclient-guide))
- REST CON SPRING TUTORIAL ([HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/](http://www.baeldung.com/rest-with-spring-series/))
- TUTORIAL DE SPRING PERSISTENCE ([HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/](http://www.baeldung.com/persistence-with-spring-series/))
- SEGURIDAD CON SPRING ([HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING](http://www.baeldung.com/security-spring))

ACERCA DE

- ACERCA DE BAELDUNG ([HTTP://WWW.BAELDUNG.COM/ABOUT/](http://www.baeldung.com/about/))
- LOS CURSOS ([HTTP://COURSES.BAELDUNG.COM](http://courses.baeldung.com))
- TRABAJO DE CONSULTORÍA ([HTTP://WWW.BAELDUNG.COM/CONSULTING](http://www.baeldung.com/consulting))
- META BAELDUNG ([HTTP://META.BAELDUNG.COM/](http://meta.baeldung.com/))
- EL ARCHIVO COMPLETO ([HTTP://WWW.BAELDUNG.COM/FULL\\_ARCHIVE](http://www.baeldung.com/full_archive))
- ESCRIBIR PARA BAELDUNG ([HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES](http://www.baeldung.com/contribution-guidelines))
- CONTACTO ([HTTP://WWW.BAELDUNG.COM/CONTACT](http://www.baeldung.com/contact))
- INFORMACIÓN DE LA COMPAÑÍA ([HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO](http://www.baeldung.com/baeldung-company-info))
- TÉRMINOS DE SERVICIO ([HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE](http://www.baeldung.com/terms-of-service))
- POLÍTICA DE PRIVACIDAD ([HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY](http://www.baeldung.com/privacy-policy))
- EDITORES ([HTTP://WWW.BAELDUNG.COM/EDITORS](http://www.baeldung.com/editors))
- KIT DE MEDIOS (PDF) ([HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF](https://s3.amazonaws.com/baeldung.com/baeldung+-+media+kit.pdf))