OAuth 2 Developers Guide

Introduction

This is the user guide for the support for `OAuth 2.0` (http://tools.ietf.org/html/draft-ietf-oauth-v2). For OAuth 1.0, everything is different, so see its user guide (oauth1.html).

This user guide is divided into two parts, the first for the OAuth 2.0 provider, the second for the OAuth 2.0 client. For both the provider and the client, the best source of sample code is the integration tests (https://github.com/spring-projects/spring-security-oauth/tree/master/tests) and sample apps (https://github.com/spring-projects/spring-security-oauth/tree/master/samples/oauth2).

OAuth 2.0 Provider

The OAuth 2.0 provider mechanism is responsible for exposing OAuth 2.0 protected resources. The configuration involves establishing the OAuth 2.0 clients that can access its protected resources independently or on behalf of a user. The provider does this by managing and verifying the OAuth 2.0 tokens used to access the protected resources. Where applicable, the provider must also supply an interface for the user to confirm that a client can be granted access to the protected resources (i.e. a confirmation page).

OAuth 2.0 Provider Implementation

The provider role in OAuth 2.0 is actually split between Authorization Service and Resource Service, and while these sometimes reside in the same application, with Spring Security OAuth you have the option to split them across two applications, and also to have multiple Resource Services that share an Authorization Service. The requests for the tokens are handled by Spring MVC controller endpoints, and access to protected resources is handled by standard Spring Security request filters. The following endpoints are required in the Spring Security filter chain in order to implement OAuth 2.0 Authorization Server:

- `AuthorizationEndpoint` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/endpoint/AuthorizationEndpoint.html) is used to service requests for authorization. Default URL: `/oauth/authorize`.
- `TokenEndpoint` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/endpoint/TokenEndpoint.html) is used to service requests for access tokens. Default URL: `/oauth/token`.

The following filter is required to implement an OAuth 2.0 Resource Server:

- The `OAuth2AuthenticationProcessingFilter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/authentication/OAuth2AuthenticationProcessingFilter.html) is used to load the Authentication for the request given an authenticated access token.

For all the OAuth 2.0 provider features, configuration is simplified using special Spring OAuth `@Configuration` adapters. There is also an XML namespace for OAuth configuration, and the schema resides at http://www.springframework.org/schema/security/spring-security-oauth2.xsd (http://www.springframework.org/schema/security/spring-security-oauth2.xsd). The namespace is `http://www.springframework.org/schema/security/oauth2`.

Authorization Server Configuration

As you configure the Authorization Server, you have to consider the grant type that the client is to use to obtain an access token from the end-user (e.g. authorization code, user credentials, refresh token). The configuration of the server is used to provide implementations of the client details service and token services and to enable or disable certain aspects of the mechanism globally. Note, however, that each client can be configured specifically

with permissions to be able to use certain authorization mechanisms and access grants. I.e. just because your provider is configured to support the "client credentials" grant type, doesn't mean that a specific client is authorized to use that grant type.

The `@EnableAuthorizationServer` annotation is used to configure the OAuth 2.0 Authorization Server mechanism, together with any `@Beans` that implement `AuthorizationServerConfigurer` (there is a handy adapter implementation with empty methods). The following features are delegated to separate configurers that are created by Spring and passed into the `AuthorizationServerConfigurer`:

- `ClientDetailsServiceConfigurer`: a configurer that defines the client details service. Client details can be initialized, or you can just refer to an existing store.
- `AuthorizationServerSecurityConfigurer`: defines the security constraints on the token endpoint.
- `AuthorizationServerEndpointsConfigurer`: defines the authorization and token endpoints and the token services.

An important aspect of the provider configuration is the way that an authorization code is supplied to an OAuth client (in the authorization code grant). A authorization code is obtained by the OAuth client by directing the end-user to an authorization page where the user can enter her credentials, resulting in a redirection from the provider authorization server back to the OAuth client with the authorization code. Examples of this are elaborated in the OAuth 2 specification.

In XML there is an `<authorization-server/>` element that is used in a similar way to configure the OAuth 2.0 Authorization Server.

Configuring Client Details

The `ClientDetailsServiceConfigurer` (a callback from your `AuthorizationServerConfigurer`) can be used to define an in-memory or JDBC implementation of the client details service. Important attributes of a client are

- `clientId`: (required) the client id.
- `secret`: (required for trusted clients) the client secret, if any.
- `scope`: The scope to which the client is limited. If scope is undefined or empty (the default) the client is not limited by scope.
- `authorizedGrantTypes`: Grant types that are authorized for the client to use. Default value is empty.
- `authorities`: Authorities that are granted to the client (regular Spring Security authorities).

Client details can be updated in a running application by access the underlying store directly (e.g. database tables in the case of `JdbcClientDetailsService`) or through the `ClientDetailsManager` interface (which both implementations of `ClientDetailsService` also implement).

> NOTE: the schema for the JDBC service is not packaged with the library (because there are too many variations you might like to use in practice), but there is an example you can start from in the test code in github (https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql).

Managing Tokens

The `AuthorizationServerTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/token/AuthorizationServerTokenServices.html) interface defines the operations that are necessary to manage OAuth 2.0 tokens. Note the following:

- When an access token is created, the authentication must be stored so that resources accepting the access token can reference it later.
- The access token is used to load the authentication that was used to authorize its creation.

When creating your `AuthorizationServerTokenServices` implementation, you may want to consider using the `DefaultTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/token/DefaultTokenServices.html) which has many strategies that can be plugged in to change the format and storage of access tokens. By default it creates tokens via random value and handles everything except for the persistence of the tokens which it delegates to a `TokenStore`. The default store is an in-memory implementation (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/token/store/InMemoryTokenStore.html), but there are some other implementations available. Here's a description with some discussion of each of them

- The default `InMemoryTokenStore` is perfectly fine for a single server (i.e. low traffic and no hot swap to a backup server in the case of failure). Most projects can start here, and maybe operate this way in development mode, to make it easy to start a server with no dependencies.

- The `JdbcTokenStore` is the JDBC version (JdbcTokenStore) of the same thing, which stores token data in a relational database. Use the JDBC version if you can share a database between servers, either scaled up instances of the same server if there is only one, or the Authorization and Resources Servers if there are multiple components. To use the `JdbcTokenStore` you need "spring-jdbc" on the classpath.

- The JSON Web Token (JWT) version (`JwtTokenStore`) of the store encodes all the data about the grant into the token itself (so no back end store at all which is a significant advantage). One disadvantage is that you can't easily revoke an access token, so they normally are granted with short expiry and the revocation is handled at the refresh token. Another disadvantage is that the tokens can get quite large if you are storing a lot of user credential information in them. The `JwtTokenStore` is not really a "store" in the sense that it doesn't persist any data, but it plays the same role of translating betweeen token values and authentication information in the `DefaultTokenServices`.

> NOTE: the schema for the JDBC service is not packaged with the library (because there are too many variations you might like to use in practice), but there is an example you can start from in the test code in github (https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql). Be sure to `@EnableTransactionManagement` to prevent clashes between client apps competing for the same rows when tokens are created. Note also that the sample schema has explicit `PRIMARY KEY` declarations - these are also necessary in a concurrent environment.

JWT Tokens

To use JWT tokens you need a `JwtTokenStore` in your Authorization Server. The Resource Server also needs to be able to decode the tokens so the `JwtTokenStore` has a dependency on a `JwtAccessTokenConverter`, and the same implementation is needed by both the Authorization Server and the Resource Server. The tokens are signed by default, and the Resource Server also has to be able to verify the signature, so it either needs the same symmetric (signing) key as the Authorization Server (shared secret, or symmetric key), or it needs the public key (verifier key) that matches the private key (signing key) in the Authorization Server (public-private or asymmetric key). The public key (if available) is exposed by the Authorization Server on the `/oauth/token_key` endpoint, which is secure by default with access rule "denyAll()". You can open it up by injecting a standard SpEL expression into the `AuthorizationServerSecurityConfigurer` (e.g. "permitAll()" is probably adequate since it is a public key).

To use the `JwtTokenStore` you need "spring-security-jwt" on your classpath (you can find it in the same github repository as Spring OAuth but with a different release cycle).

Grant Types

The grant types supported by the `AuthorizationEndpoint` can be configured via the `AuthorizationServerEndpointsConfigurer`. By default all grant types are supported except password (see below for details of how to switch it on). The following properties affect grant types:

- `authenticationManager`: password grants are switched on by injecting an `AuthenticationManager`.
- `userDetailsService`: if you inject a `UserDetailsService` or if one is configured globally anyway (e.g. in a `GlobalAuthenticationManagerConfigurer`) then a refresh token grant will contain a check on the user details, to ensure that the account is still active
- `authorizationCodeServices`: defines the authorization code services (instance of `AuthorizationCodeServices`) for the auth code grant.
- `implicitGrantService`: manages state during the imlpicit grant.
- `tokenGranter`: the `TokenGranter` (taking full control of the granting and ignoring the other properties above)

In XML grant types are included as child elements of the `authorization-server`.

Configuring the Endpoint URLs

The `AuthorizationServerEndpointsConfigurer` has a `pathMapping()` method. It takes two arguments:

- The default (framework implementation) URL path for the endpoint
- The custom path required (starting with a "/")

The URL paths provided by the framework are `/oauth/authorize` (the authorization endpoint), `/oauth/token` (the token endpoint), `/oauth/confirm_access` (user posts approval for grants here), `/oauth/error` (used to render errors in the authorization server), `/oauth/check_token` (used by Resource Servers to decode access tokens), and `/oauth/token_key` (exposes public key for token verification if using JWT tokens).

N.B. the Authorization endpoint `/oauth/authorize` (or its mapped alternative) should be protected using Spring Security so that it is only accessible to authenticated users. For instance using a standard Spring Security `WebSecurityConfigurer`:

```
@Override
 protected void configure(HttpSecurity http) throws Exception {
     http
         .authorizeRequests().antMatchers("/login").permitAll().and()
     // default protection for all resources (including /oauth/authorize)
         .authorizeRequests()
             .anyRequest().hasRole("USER")
     // ... more configuration, e.g. for form login
 }
```

> Note: if your Authorization Server is also a Resource Server then there is another security filter chain with lower priority controlling the API resources. Fo those requests to be protected by access tokens you need their paths *not* to be matched by the ones in the main user-facing filter chain, so be sure to include a request matcher that picks out only non-API resources in the `WebSecurityConfigurer` above.

The token endpoint is protected for you by default by Spring OAuth in the `@Configuration` support using HTTP Basic authentication of the client secret. This is not the case in XML (so it should be protected explicitly).

In XML the `<authorization-server/>` element has some attributes that can be used to change the default endpoint URLs in a similar way. The `/check_token` endpoint has to be explicitly enabled (with the `check-token-enabled` attribute).

Customizing the UI

Most of the Authorization Server endpoints are used primarily by machines, but there are a couple of resource that need a UI and those are the GET for `/oauth/confirm_access` and the HTML response from `/oauth/error`. They are provided using whitelabel implementations in the framework, so most real-world instances of the Authorization Server will want to provide their own so they can control the styling and content. All you need to do is provide a Spring MVC controller with `@RequestMappings` for those endpoints, and the framework defaults will take a lower priority in the dispatcher. In the `/oauth/confirm_access` endpoint you can expect an `AuthorizationRequest` bound to the session carrying all the data needed to seek approval from the user (the default implementation is `WhitelabelApprovalEndpoint` so look there for a starting point to copy). You can grab all the data from that request and render it however you like, and then all the user needs to do is POST back to `/oauth/authorize` with information about approving or denying the grant. The request parameters are passed directly to a `UserApprovalHandler` in the `AuthorizationEndpoint` so you can interpret the data more or less as you please. The default `UserApprovalHandler` depends on whether or not you have supplied an `ApprovalStore` in your `AuthorizationServerEndpointsConfigurer` (in which case it is an `ApprovalStoreUserApprovalHandler`) or not (in which case it is a `TokenStoreUserApprovalHandler`). The standard approval handlers accept the following:

- `TokenStoreUserApprovalHandler`: a simple yes/no decision via `user_oauth_approval` equals to "true" or "false".
- `ApprovalStoreUserApprovalHandler`: a set of `scope.*` parameter keys with "*" equal to the scopes being requested. The value of the parameter can be "true" or "approved" (if the user approved the grant) else the user is deemed to have rejected that scope. A grant is successful if at least one scope is approved.

> NOTE: don't forget to include CSRF protection in your form that you render for the user. Spring Security is expecting a request parameter called "_csrf" by default (and it provides the value in a request attribute). See the Spring Security user guide for more information on that, or look at the whitelabel implementation for guidance.

Enforcing SSL

Plain HTTP is fine for testing but an Authorization Server should only be used over SSL in production. You can run the app in a secure container or behind a proxy and it should work fine if you set the proxy and the container up correctly (which is nothing to do with OAuth2). You might also want to secure the endpoints using Spring Security `requiresChannel()` constraints. For the `/authorize` endpoint is up to you to do that as part of your normal application security. For the `/token` endpoint there is a flag in the `AuthorizationServerEndpointsConfigurer` that you can set using the `sslOnly()` method. In both cases the secure channel setting is optional but will cause Spring Security to redirect to what it thinks is a secure channel if it detects a request on an insecure channel.

Customizing the Error Handling

Error handling in an Authorization Server uses standard Spring MVC features, namely `@ExceptionHandler` methods in the endpoints themselves. Users can also provide a `WebResponseExceptionTranslator` to the endpoints themselves which is the best way to change the content of the responses as opposed to the way they are rendered. The rendering of exceptions delegates to `HttpMesssageConverters` (which can be added to the

MVC configuration) in the case of token endpoint and to the OAuth error view (`/oauth/error`) in the case of teh authorization endpoint. The whitelabel error endpoint is provided for HTML responses, but users probably need to provide a custom implementation (e.g. just add a `@Controller` with `@RequestMapping("/oauth/error")`).

Mapping User Roles to Scopes

It is sometimes useful to limit the scope of tokens not only by the scopes assigned to the client, but also according to the user's own permissions. If you use a `DefaultOAuth2RequestFactory` in your `AuthorizationEndpoint` you can set a flag `checkUserScopes=true` to restrict permitted scopes to only those that match the user's roles. You can also inject an `OAuth2RequestFactory` into the `TokenEndpoint` but that only works (i.e. with password grants) if you also install a `TokenEndpointAuthenticationFilter` - you just need to add that filter after the HTTP `BasicAuthenticationFilter` . Of course, you can also implement your own rules for mapping scopes to roles and install your own version of the `OAuth2RequestFactory` . The `AuthorizationServerEndpointsConfigurer` allows you to inject a custom `OAuth2RequestFactory` so you can use that feature to set up a factory if you use `@EnableAuthorizationServer` .

Resource Server Configuration

A Resource Server (can be the same as the Authorization Server or a separate application) serves resources that are protected by the OAuth2 token. Spring OAuth provides a Spring Security authentication filter that implements this protection. You can switch it on with `@EnableResourceServer` on an `@Configuration` class, and configure it (as necessary) using a `ResourceServerConfigurer` . The following features can be configured:

- `tokenServices` : the bean that defines the token services (instance of `ResourceServerTokenServices`).
- `resourceId` : the id for the resource (optional, but recommended and will be validated by the auth server if present).
- other extension points for the resourecs server (e.g. `tokenExtractor` for extracting the tokens from incoming requests)
- request matchers for protected resources (defaults to all)
- access rules for protected resources (defaults to plain "authenticated")
- other customizations for the protected resources permitted by the `HttpSecurity` configurer in Spring Security

The `@EnableResourceServer` annotation adds a filter of type `OAuth2AuthenticationProcessingFilter` automatically to the Spring Security filter chain.

In XML there is a `<resource-server/>` element with an `id` attribute - this is the bean id for a servlet `Filter` that can then be added manually to the standard Spring Security chain.

Your `ResourceServerTokenServices` is the other half of a contract with the Authorization Server. If the Resource Server and Authorization Server are in the same application and you use `DefaultTokenServices` then you don't have to think too hard about this because it implements all the necessary interfaces so it is automatically consistent. If your Resource Server is a separate application then you have to make sure you match the capabilities of the Authorization Server and provide a `ResourceServerTokenServices` that knows how to decode the tokens correctly. As with the Authorization Server, you can often use the `DefaultTokenServices` and the choices are mostly expressed through the `TokenStore` (backend storage or local encoding). An alternative is the `RemoteTokenServices` which is a Spring OAuth features (not part of the spec) allowing Resource Servers to decode tokens through an HTTP resource on the Authorization Server (`/oauth/check_token`). `RemoteTokenServices` are convenient if there is not a huge volume of traffic in the Resource Servers (every request has to be verified with the Authorization Server), or if you can afford to cache the results. To use the `/oauth/check_token` endpoint you need to expose it by changing its access rule (default is "denyAll()") in the `AuthorizationServerSecurityConfigurer` , e.g.

```
@Override
public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {
        oauthServer.tokenKeyAccess("isAnonymous() || hasAuthority('ROLE_TRUSTED_CLIENT')").checkTokenAccess(
                        "hasAuthority('ROLE_TRUSTED_CLIENT')");
}
```

In this example we are configuring both the `/oauth/check_token` endpoint and the `/oauth/token_key` endpoint (so trusted resources can obtain the public key for JWT verification). These two endpoints are protected by HTTP Basic authentication using client credentials.

Configuring An OAuth-Aware Expression Handler

You may want to take advantage of Spring Security's expression-based access control (http://docs.spring.io/spring-security/site/docs/3.2.5.RELEASE/reference/htmlsingle/#el-access). An expression handler will be registered by default in the `@EnableResourceServer` setup. The expressions include *#oauth2.clientHasRole*, *#oauth2.clientHasAnyRole*, and *#oath2.denyClient* which can be used to provide access based on the role of the oauth client (see `OAuth2SecurityExpressionMethods` for a comprehensive list). In XML you can register a oauth-aware expression handler with the `expression-handler` element of the regular `<http/>` security configuration.

OAuth 2.0 Client

The OAuth 2.0 client mechanism is responsible for access the OAuth 2.0 protected resources of other servers. The configuration involves establishing the relevant protected resources to which users might have access. The client may also need to be supplied with mechanisms for storing authorization codes and access tokens for users.

Protected Resource Configuration

Protected resources (or "remote resources") can be defined using bean definitions of type `OAuth2ProtectedResourceDetails` (/spring-security-oauth2/src/main/java/org/springframework/security/oauth2/client/resource/OAuth2ProtectedResourceDetails.java). A protected resource has the following properties:

- `id` : The id of the resource. The id is only used by the client to lookup the resource; it's never used in the OAuth protocol. It's also used as the id of the bean.
- `clientId` : The OAuth client id. This is the id by which the OAuth provider identifies your client.
- `clientSecret` : The secret associated with the resource. By default, no secret is empty.
- `accessTokenUri` : The URI of the provider OAuth endpoint that provides the access token.
- `scope` : Comma-separted list of strings specifying the scope of the access to the resource. By default, no scope will be specified.
- `clientAuthenticationScheme` : The scheme used by your client to authenticate to the access token endpoint. Suggested values: "http_basic" and "form". Default: "http_basic". See section 2.1 of the OAuth 2 spec.

Different grant types have different concrete implementations of `OAuth2ProtectedResourceDetails` (e.g. `ClientCredentialsResource` for "client_credentials" grant type). For grant types that require user authorization there is a further property:

- `userAuthorizationUri` : The uri to which the user will be redirected if the user is ever needed to authorize access to the resource. Note that this is not always required, depending on which OAuth 2 profiles are supported.

In XML there is a `<resource/>` element that can be used to create a bean of type `OAuth2ProtectedResourceDetails`. It has attributes matching all the properties above.

Client Configuration

For the OAuth 2.0 client, configuration is simplified using `@EnableOAuth2Client`. This does 2 things:

- Creates a filter bean (with ID `oauth2ClientContextFilter`) to store the current request and context. In the case of needing to authenticate during a request it manages the redirection to and from the OAuth authentication uri.

- Creates a bean of type `AccessTokenRequest` in request scope. This can be used by authorization code (or implicit) grant clients to keep state related to individual users from colliding.

The filter has to be wired into the application (e.g. using a Servlet initializer or `web.xml` configuration for a `DelegatingFilterProxy` with the same name).

The `AccessTokenRequest` can be used in an `OAuth2RestTemplate` like this:

```
@Autowired
private OAuth2ClientContext oauth2Context;

@Bean
public OAuth2RestTemplate sparklrRestTemplate() {
        return new OAuth2RestTemplate(sparklr(), oauth2Context);
}
```

The OAuth2ClientContext is placed (for you) in session scope to keep the state for different users separate. Without that you would have to manage the equivalent data structure yourself on the server, mapping incoming requests to users, and associating each user with a separate instance of the `OAuth2ClientContext`.

In XML there is a `<client/>` element with an `id` attribute - this is the bean id for a servlet `Filter` that must be mapped as in the `@Configuration` case to a `DelegatingFilterProxy` (with the same name).

Accessing Protected Resources

Once you've supplied all the configuration for the resources, you can now access those resources. The suggested method for accessing those resources is by using the `RestTemplate` introduced in Spring 3 (http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html). OAuth for Spring Security has provided an extension of RestTemplate (/spring-security-oauth2/src/main/java/org/springframework/security/oauth2/client/OAuth2RestTemplate.java) that only needs to be supplied an instance of `OAuth2ProtectedResourceDetails` (/spring-security-oauth2/src/main/java/org/springframework/security/oauth2/client/resource/OAuth2ProtectedResourceDetails.java). To use it with user-tokens (authorization code grants) you should consider using the `@EnableOAuth2Client` configuration (or the XML equivalent `<oauth:rest-template/>`) which creates some request and session scoped context objects so that requests for different users do not collide at runtime.

As a general rule, a web application should not use password grants, so avoid using `ResourceOwnerPasswordResourceDetails` if you can in favour of `AuthorizationCodeResourceDetails`. If you desparately need password grants to work from a Java client, then use the same mechanism to configure your `OAuth2RestTemplate` and add the credentials to the `AccessTokenRequest` (which is a `Map` and is ephemeral) not the `ResourceOwnerPasswordResourceDetails` (which is shared between all access tokens).

Persisting Tokens in a Client

A client does not *need* to persist tokens, but it can be nice for users to not be required to approve a new token grant every time the client app is restarted. The `ClientTokenServices` (/spring-security-oauth2/src/main/java/org/springframework/security/oauth2/client/token/ClientTokenServices.java) interface defines the operations that are necessary to persist OAuth 2.0 tokens for specific users. There is a JDBC implementation provided, but you can if you prefer implement your own service for storing the access tokens and associated authentication instances in a persistent database. If you want to use this feature you need provide a specially configured `TokenProvider` to the `OAuth2RestTemplate` e.g.

```
@Bean
@Scope(value = "session", proxyMode = ScopedProxyMode.INTERFACES)
public OAuth2RestOperations restTemplate() {
        OAuth2RestTemplate template = new OAuth2RestTemplate(resource(), new DefaultOAuth2ClientContext(accessTokenRequest));
        AccessTokenProviderChain provider = new AccessTokenProviderChain(Arrays.asList(new
AuthorizationCodeAccessTokenProvider()));
        provider.setClientTokenServices(clientTokenServices());
        return template;
}
```

Customizations for Clients of External OAuth2 Providers

Some external OAuth2 providers (e.g. Facebook (http://developers.facebook.com/docs/authentication)) do not quite implement the specification correctly, or else they are just stuck on an older version of the spec than Spring Security OAuth. To use those providers in your client application you might need to adapt various parts of the client-side infrastructure.

To use Facebook as an example, there is a Facebook feature in the `tonr2` application (you need to change the configuration to add your own, valid, client id and secret - they are easy to generate on the Facebook website).

Facebook token responses also contain a non-compliant JSON entry for the expiry time of the token (they use `expires` instead of `expires_in`), so if you want to use the expiry time in your application you will have to decode it manually using a custom `OAuth2SerializationService`.