JAX-RS OAuth2

JAX-RS: OAuth2

Introduction

New:

- JCache and JPA2 OAuthDataProviders (as well as Ehcache 2.x prior to CXF 3.3.0) can represent access tokens in JWT
- Client Certificate Authentication and Token Binding is supported
- DynamicRegistrationService is enhanced

CXF provides an implementation of OAuth 2.0. See also the JAX-RS OAuth page for information about OAuth 1.0.

Authorization Code, Implicit, Client Credentials, Resource Owner Password Credentials, Refresh Token, SAML2 Assertions and JWT assertion grants are currently supported.

Custom grant handlers can be registered.

OAuth2 is a protocol which offers a complex yet elegant solution toward helping end users (resource owners) authorize third-party providers to access their resources.

The OAuth2 flow which is closely related to the original OAuth 1.0 3-leg flow is called Authorization Code and involves 3 parties: the end user, the third party service (client) and the resource server which is protected by OAuth2 filters. Typically a client offers a service that an end user requests and which requires the former to access one or more protected resources on behalf of this user which are located at the resource server. For example, the client may need to access the end user's photos in order to print them and post to the user or read and possibly update a user's calendar in order to make a booking.

In order to make this happen, the third-party service application/client needs to register itself with the OAuth2 server. This happens out-of-band and after the registration the client gets back a client key and secret pair. Typically the client is expected to provide the name and description of the application, the application logo URI, one or more redirect URIs, and other information that may help the OAuth2 authorization server to identify this client to the end user at authorization time.

From then on, the authorization code flow works like this:
1. End User requests the third-party service using a browser.

2. The client redirects the end user to OAuth2 Authorization Service, adding its client id, the state, redirect URI and the optional scope to the target URI. The state parameter represents the current end user's request, redirect URI - where the authorization code is expected to be returned to, and the scope is the list of opaque permissions that the client needs in order to access the protected resources.

3. The Authorization Service will retrieve the information about the client using its client id, build an HTML form and return it to the end user. The form will ask the user if the given third-party application can be allowed to access specified resources on behalf of this user.

4. If the user approves, then the Authorization Service will generate an authorization code and redirect the user back to the redirect uri provided by the client, also adding a state parameter to the redirect URI.

5. The client requests an access token from OAuth2 Access Token Service by providing an authorization code grant.

6. After getting an access token, the service finally proceeds with accessing the current user's resources and completes the user's request.

As you can see the flow can be complex yet it is very effective. A number of issues may need to be taken care along the way such as managing expired tokens, making sure that the OAuth2 security layer is functioning properly and not interfering when the end user itself is trying to access its own resources, etc.

Please check the specification as well as other resources available on the web for more information you may need to know about OAuth2.

CXF JAX-RS gives the best effort to making this process as simple as possible and requiring only a minimum effort on behalf of OAuth2 server developers. It also offers utility code for greatly simplifying the way third-party applications can interact with the OAuth2 service endpoints.

Maven dependencies

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-security-oauth2</artifactId>
  <version>3.1.7</version>
</dependency>
```

Client Registration

Client Registration is typically done out of band, with dynamic client registration also being possible.
The client registration service will offer an HTML form where the clients will enter their details, see the Client bean for currently supported properties.

See this JAX-RS service implementation for one possible approach.

Developing OAuth2 Servers

OAuth2 server is the core piece of the complete OAuth2-based solution. Typically it contains 3 services for:
1. Authorizing request tokens by asking the end users to let clients access some of their resources and returning the grants back to the client (Authorization Service)
2. Exchanging the token grants for access tokens (Access Token Service)

3. Validating access tokens

CXF offers several JAX-RS service implementations that can be used to create OAuth2 servers quickly: AuthorizationCodeGrantService and ImplicitGrantService for managing the redirection-based flows, as well as AccessTokenService for exchanging the grants for new tokens.

All of these services rely on the custom OAuthDataProvider which persists the access tokens and converts the opaque scope values to the information that can be presented to the users. Additionally, AuthorizationCodeDataProvider is an OAuthDataProvider which can keep temporary information about the authorization code grants which needs to be removed after the tokens are requested in exchange.

Note that some grants that do not require redirection-based support, such as Client Credentials or SAML2 or JWT assertion grants, and instead may require only an Access Token Service be operational.

If your OAuth2 server does support either Authorization Code or Implicit flow then either AuthorizationCodeGrantService and ImplicitGrantService need to be registered. If both services need to be supported then simply register two of them, but note each service will have its own @Path segment, "/authorize" and "/authorize-implicit". If you'd like both services listening on the same path then use AuthorizationService and inject the AuthorizationCodeGrantService and ImplicitGrantService beans into it.

If no AuthorizationCode redirection flow is supported then implementing OAuthDataProvider is sufficient.

Writing your own AuthorizationCodeDataProvider or OAuthDataProvider implementation is what is needed to get the OAuth2 server up and running. In many cases all you need to do is to persist or remove the Authorization Code Grant data, use one of the available utility classes to create a new access token and also persist it or remove the expired one, and finally convert the optional opaque scope values (if any are supported) to a more view-able information.

CXF ships several default provider implementations, see the section on writing the providers below.

Authorization Service

The main responsibility of OAuth2 Authorization Service is to present an end user with a form asking the user to allow or deny the client to access specified resources owned by the user. CXF offers AuthorizationCodeGrantService and ImplicitGrantService for accepting redirection requests, challenging end users with the authorization forms, handling the end user decisions and returning the results back to the clients.

One of the differences between Authorization Code and Implicit flows is that in the latter case the grant is the actual access token which is returned as the URI fragment value to the client script running in the browser. The way the end user is asked to authorize the client request is similar between the two flows. In this section we will assume that the Authorization Code flow is being used.

A third-party client redirects the current user to AuthorizationCodeGrantService, for example, here is how a redirection may happen:

```
Response-Code: 303
Headers: {Location=[http://localhost:8080/services/social/authorize?client_id=123456789&scope=updateCalendar-
7&response_type=code
&redirect_uri=http%3A//localhost%3A8080/services/reservations/reserve/complete&state=1],
Date=[Thu, 12 Apr 2012 12:26:21 GMT], Content-Length=[0]}
```

The client application asks the current user (the browser) to go to a new address provided by the Location header and the follow-up request to AuthorizationCodeGrantService will look like this:

```
Address: http://localhost:8080/services/social/authorize?client_id=123456789&scope=updateCalendar-
7&response_type=code
&redirect_uri=http%3A//localhost%3A8080/services/reservations/reserve/complete&state=1
Http-Method: GET
Headers: {
Accept=[text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8],
Authorization=[Basic YmFycnlAc29jaWFsLmNvbToxMjM0],
Cookie=[JSESSIONID=suj2wyl54c4g],
Referer=[http://localhost:8080/services/forms/reservation.jsp]
...
}
```

Note that the end user needs to authenticate. The Request URI includes the client_id, custom scope value, response_type set to 'code', the current request state and the redirect uri. Note the scope is optional - the Authorization Service will usually allocate a default scope; however even if the client does include an additional custom scope the end user may still not approve it. The redirect uri is also optional, assuming one or more redirect URIs were provided when the client was registered with the Authorization Service.

The AuthorizationCodeGrantService will report a warning if no secure HTTPS transport is used:

```
12-Apr-2012 13:26:21 org.apache.cxf.rs.security.oauth2.services.AbstractOAuthService checkTransportSecurity
WARNING: Unsecure HTTP, Transport Layer Security is recommended
```

It can also be configured to reject requests made over insecure HTTP transport.

The AuthorizationCodeGrantService will retrieve the information about the client application to populate an instance of OAuthAuthorizationData bean and return it. OAuthAuthorizationData contains application name and URI properties, optional list of Permissions and other properties which can be either presented to the user or kept in the hidden form fields in order to uniquely identify the actual authorization request when the end user returns the decision.

One important OAuthAuthorizationData property is "authenticityToken". It is used for validating that the current session has not been hijacked - AuthorizationCodeGrantService generates a random key, stores it in a Servlet HTTPSession instance and expects the returned authenticityToken value to match it - this is a recommended approach and it also implies that the authenticityToken value is hidden from a user, for example, it's kept in a 'hidden' form field. See also "User Session Authenticity" on how a SessionAuthenticityTokenProvider can help.

A number of properties which have been submitted to the authorization endpoint with the original user redirect need to be made available after the user has been challenged with the authorization consent form, when the user makes an authorization decision. These are properties such as 'clientId', 'state', 'redirectUri', and other properties, see this class which is extended by OAuthAuthorizationData. One simple approach is to have a view handler prepare an authorization form with OAuthAuthorizationData to have these properties hidden in the form. Another option is to secure them in a session thus making the view creation process much simpler, see "User Session Authenticity" for one example.

The helper "replyTo" property is an absolute URI identifying the AuthorizationCodeGrantService handler which processes the user's decision and can be used by view handlers when building the forms or by other OAuthAuthorizationData handlers.

So the populated OAuthAuthorizationData is finally returned. Note that it's a JAXB XMLRootElement-annotated bean and can be processed by registered JAXB or JSON providers given that AuthorizationCodeGrantService supports producing "application/xml" and "application/json" (See the OAuth Without Browser section below for more). But in this case we have the end user working with a browser so an HTML form is what is expected back.

The AuthorizationCodeGrantService supports producing "text/html" and simply relies on a registered RequestDispatcherProvider to set the OAuthAuthorizationData bean as an HttpServletRequest attribute and redirect the response to a view handler (can be JSP or some other servlet) to actually build the form and return it to the user. See the section below on other alternatives on how a view can be created.

Assuming RequestDispatcherProvider is used, the following example log shows the initial response from AuthorizationCodeGrantService:

```
12-Apr-2012 13:26:21 org.apache.cxf.jaxrs.provider.RequestDispatcherProvider logRedirection
INFO: Setting an instance of "org.apache.cxf.rs.security.oauth2.common.OAuthAuthorizationData" as
HttpServletRequest
attribute "data" and redirecting the response to "/forms/oauthAuthorize.jsp".
```

Note that a "/forms/oauthAuthorize.jsp" view handler will create an HTML view - this is a custom JSP handler and whatever HTML view is required can be created there, using the OAuthAuthorizationData bean for building the view. Most likely you will want to present a form asking the user to allow or deny the client accessing some of this user's resources. If OAuthAuthorizationData has a list of Permissions set then adding the information about the permissions is needed.

Next the user makes a decision and selects a button allowing or denying the client accessing the resources. The form data are submitted to AuthorizationCodeGrantService:

```
Address: http://localhost:8080/services/social/authorize/decision
Encoding: ISO-8859-1
Http-Method: POST
Content-Type: application/x-www-form-urlencoded
Headers: {
Authorization=[Basic YmFycnlAc29jaWFsLmNvbToxMjM0],
Content-Type=[application/x-www-form-urlencoded],
...
}
--------------------------------------
12-Apr-2012 15:36:29 org.apache.cxf.jaxrs.utils.FormUtils logRequestParametersIfNeeded
INFO: updateCalendar-7_status=allow&readCalendar_status=allow&scope=updateCalendar-7+readCalendar
&redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Fservices%2Freservations%2Freserve%2Fcomplete
&session_authenticity_token=4f0005d9-565f-4309-8ffb-c13c72139ebe&oauthDecision=allow&state=1&client_id=123456789
```

The AuthorizationCodeGrantService will use the 'session_authenticity_token' to first check that the session is valid and if so will process the user's decision.

If the decision is "allow" then it will check the status of the submitted individual scope values. It relies on the "scopename_status" convention (e.g., "updateCalendar-7_status=allow") in name/value pairs, if the form offers the user a chance to selectively enable individual scopes. If none such pairs are returned then it means the user has approved all the default and any additional scopes.

Next it will ask the OAuthDataProvider to generate an authorization code grant and return it alongside with the state if any by redirecting the current user back to the redirect URI:

```
Response-Code: 303
Headers: {
 Location=[http://localhost:8080/services/reservations/reserve/complete?state=1
&code=5c993144b910bccd5977131f7d2629ab],
 Date=[Thu, 12 Apr 2012 14:36:29 GMT],
 Content-Length=[0]}
```

which leads to a browser redirecting the user:

```
Address: http://localhost:8080/services/reservations/reserve/complete?
state=1&code=5c993144b910bccd5977131f7d2629ab
Http-Method: GET
Headers: {
Accept=[text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8],
Authorization=[Basic YmFycnlAcmVzdGF1cmFudC5jb206NTY3OA==],
Cookie=[JSESSIONID=1c289vha0cxfe],
}
```

If a user decision was set to "deny" then the error will be returned to the client.

Assuming the decision was "allow", the client has now received back the authorization code grant and is ready to exchange it for a new access token.

How to create Authorization View

Typically one can use RequestDispatcherProvider to redirect to a view handler like JSP. Overriding RedirectionBasedService.startAuthorization by delegating to the superclass and then converting the Response to HTML or writing a custom MessageBodyWriter that will do the conversion are other two options.

Yet another option is to register XSLTJaxbProvider which would convert OAuthAuthorizationData to HTML or JAXBProvider set with a reference to an XSLT stylesheet.

EndUser Name in Authorization Form

You may want to display a resource owner/end user name in the authorization form this user will be facing, you can get org.apache.cxf.rs.security.oauth2.provider.ResourceOwnerNameProvider registered with either AuthorizationCodeGrantService or ImplicitGrantService. org.apache.cxf.rs.security.oauth2.provider.DefaultResourceOwnerNameProvider, if registered, will return an actual login name, the custom implementations may choose to return a complete user name instead, etc.

Public Clients (Devices)

The Client can be 'public' if it has been registered as a public client with no client secret if the service itself has a "canSupportPublicClients" property enabled. The same property will also have to be enabled on AccessTokenService (described in the next section) for a public client without a secret be able to exchange a code grant for an access token.

OOB Response

If a public client has not registered a redirect URI with the Authorization service then the authorization code can be returned out-of-band (OOB), see OOBAuthorizationResponse bean. By default, it is returned directly to the end user, unless a custom OOBResponseDeliverer is registered with AuthorizationCodeGrantService which may deliver it to the client via some custom back channel.

Having OOB responses supported is useful when a public client (typically a device which can not keep client secrets and/or where no redirect URI is supported) needs to get a code grant. What will happen is that a device owner will send a request to the Authorization Service which may look like this:

```
GET
http://localhost:8080/services/social/authorize?client_id=mobileClient&response_type=code
```

Assuming the 'mobileClient' has been registered as public one with no secret and the service has been set up to support such clients, the end user will get a chance to authorize this client the same way it can confidential clients, and after this user gets back a code (delivered directly in the response HTML page by default) the user will enter the code securely into the device which will then replace it for a time-scoped access token by contacting AccessTokenService.

PKCE support

CXF supports RFC-7636: Proof Key for Code Exchange by OAuth Public Clients (PKCE). If you are using the authorization code grant with public clients, it is recommended to use PKCE to avoid attacks which exploit the lack of a binding between the authorization code request and the token request. This binding is achieved for confidential clients by including the client_id in both requests, however with public clients we do not have a registered client_id.

The public (mobile) client generates a 'code_verifier' value and includes a related 'code_challenge' and optional 'code_challenge_method' to the authorization service. The authorization service will save the code_challenge value, with the help of the registered AuthorizationCodeDataProvider into an instance of ServerAuthorizationCodeGrant. The client will next request a token providing the 'code_verifier' - which will be compared by AuthorizationCodeGrantHandler with the original 'code_challenge' value. By default, the 'code_challenge' is expected to be equal to the original 'code_verifier', but the grant handler can be registered with a custom org.apache.cxf.rs.security.oauth2.grants.code.CodeVerifierTransformer. CXF ships a DigestCodeVerifier which implements a transformation mentioned in the extension.

From CXF 3.5.1/3.4.6, the AuthorizationCodeGrantHandler has a "requireCodeVerifier" property, which when set to "true" requires that all clients must specify a code verifier. This aligns with the OAuth 2.1 specification.

Form Post Response Mode

Form Post Response Mode was originally introduced for OpenId Connect but has been generally recommended recently as a possibly safer option of returning OAuth2 Authorization Service responses to clients. Starting from CXF 3.1.9, if a client sends a "response_mode=form_post" parameter during the original redirect, CXF AuthorizationCodeService will return an OOBAuthorizationResponse with its 'redirectUri' property set - which a JSP/etc handler will convert to an HTML form which will re-post the data to the client callback address.

AccessTokenService

The role of AccessTokenService is to exchange a token grant for a new access token which will be used by the client to access the end user's resources. Here is an example request log:

```
Address: http://localhost:8080/services/oauth/token
Http-Method: POST

Headers: {
Accept=[application/json],
Authorization=[Basic MTIzNDU2Nzg5Ojk4NzY1NDMyMQ==],
Content-Type=[application/x-www-form-urlencoded]
}
Payload:

grant_type=authorization_code&code=5c993144b910bccd5977131f7d2629ab
&redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Fservices%2Freservations%2Freserve%2Fcomplete
```

This request contains a client_id and client_secret (Authorization header), the grant_type, the grant value (code) plus the redirect URI the authorization grant was returned to which is needed for the additional validation. Note that the alternative client authentication methods are also possible, in this case the token service will expect a mapping between the client credentials and the client_id representing the client registration available.

After validating the request, the service will find a matching AccessTokenGrantHandler and request to create a ServerAccessToken which is a server-side representation of the access token.
The grant handlers, such as AuthorizationCodeGrantHandler may delegate the creation of the actual access token to data providers, which will create access tokens with the help of utility classes shipped with CXF or depend on other 3rd party token libraries.

The data providers do not strictly required to persist the data such as access tokens, instead the token key may act as an encrypted bag capturing all the relevant information.

Note that AccessTokenService may not need to have AccessTokenGrantHandler injected - if it finds out that the data provider is AuthorizationCodeDataProvider then it will create AuthorizationCodeGrantHandler itself. This will work well unless AuthorizationCodeGrantHandler itself needs to be customized and thus directly injected into AccessTokenService.

Now that the token has been created, it is mapped by the service to a client access token representation and is returned back as a JSON payload:

```
Response-Code: 200
Content-Type: application/json
Headers: {
 Cache-Control=[no-store],
 Pragma=[no-cache],
 Date=[Thu, 12 Apr 2012 14:36:29 GMT]
}

Payload:

{"access_token":"5b5c8e677413277c4bb8b740d522b378", "token_type":"bearer"}
```

The client will use this access token to access the current user's resources in order to complete the original user's request, for example, the request to access a user's calendar may look like this:

```
Address: http://localhost:8080/services/thirdPartyAccess/calendar
Http-Method: GET
Headers:
{
  Authorization=[Bearer 5b5c8e677413277c4bb8b740d522b378],
  Accept=[application/xml]
}
```

Note that the access token key is passed as the Bearer scheme value. Other token types such as MAC ones, etc, can be represented differently.

Access Token Types

As mentioned above, AccessTokenService can work with whatever token is created by a given data provider. This section provides more information on how CXF may help with supporting Bearer and other token types.

Bearer

The following code fragment shows how a BearerAccessToken utility class can be used to create Bearer tokens:

```
import org.apache.cxf.rs.security.oauth2.common.AccessTokenRegistration;
import org.apache.cxf.rs.security.oauth2.common.ServerAccessToken;
import org.apache.cxf.rs.security.oauth2.tokens.bearer.BearerAccessToken;

public class CustomOAuthDataProvider implements AuthorizationCodeDataProvider {

    public ServerAccessToken createAccessToken(AccessTokenRegistration reg)
                throws OAuthServiceException {

                ServerAccessToken token = new BearerAccessToken(reg.getClient(), 3600L);

                List<String> scope = reg.getApprovedScope().isEmpty() ? reg.getRequestedScope()
                                                                        : reg.getApprovedScope();
                token.setScopes(convertScopeToPermissions(reg.getClient(), scope));
                token.setSubject(reg.getSubject());
                token.setGrantType(reg.getGrantType());

        // persist or encrypt and return

                return token;
   }
   // other methods are not shown
}
```

CustomOAuthDataProvider will also be asked by OAuthRequestFilter to validate the incoming Bearer tokens given that they typically act as database key or key alias, if no Bearer token validator is registered.

Note that all the default providers shipped with CXF create and persist Bearer access tokens themselves.

HAWK

Starting from CXF 3.0.0-milestone2 the Hawk scheme is supported instead of MAC (described in the next section). The way it is supported is identical to the way MAC scheme is supported in earlier CXF versions. The only differences are: 'Hawk' replaces 'Mac' in the Authorization header, the Hawk token returned by the server will have 'secret' and 'algorithm' parameters instead of 'mac_key' and 'mac_algorithm' parameters.

MAC

The text below applies to CXF up to 3.0.0-milestone2. Starting from 3.0.0-milestone2 MAC scheme is not supported, see above about the Hawk scheme. See also OAuth2 Proof Of Possession Tokens which will be supported in CXF in the future.

CXF 2.6.2 supports MAC tokens as specified in the latest MAC Access Authentication draft created by Eran Hammer and others. MAC tokens offer an option for clients to demonstrate they 'hold' the token secret issued to them by AccessTokenService.
It is recommended that AccessTokenService endpoint issuing MAC tokens enforces a two-way TLS for an extra protection of the MAC token data returned to clients.

The following code fragment shows how a MacAccessToken utility class can be used to create MAC tokens:

```
import org.apache.cxf.rs.security.oauth2.common.AccessTokenRegistration;
import org.apache.cxf.rs.security.oauth2.common.ServerAccessToken;
import org.apache.cxf.rs.security.oauth2.tokens.mac.HmacAlgorithm;
import org.apache.cxf.rs.security.oauth2.tokens.mac.MacAccessToken;

public class CustomOAuthDataProvider implements AuthorizationCodeDataProvider {

    public ServerAccessToken createAccessToken(AccessTokenRegistration reg)
                throws OAuthServiceException {

                // generate
                ServerAccessToken token = new MacAccessToken(reg.getClient(),
                                                             HmacAlgorithm.HmacSHA1,
                                                             3600L);

                // set other token fields as shown in the Bearer section

                // persist as needed and then return

                return token;
   }
   // other methods are not shown
}
```

One can expect the following response:

```
Response-Code: 200
Content-Type: application/json
Headers: {
 Cache-Control=[no-store],
 Pragma=[no-cache],
 Date=[Thu, 12 Apr 2012 14:36:29 GMT]
}

Payload:

{"access_token":"5b5c8e677413277c4bb8b740d522b378", "token_type":"mac", "mac_key"="1234568",
"mac_algorithm"="hmac-sha-1"}
```

Note that 'access_token' is the MAC key identifier.

MacAccessTokenValidator has to be registered with OAuthRequestFilter for validating the incoming MAC tokens. This validator can get a reference to custom NonceVerifier with CXF possibly shipping a default implementation in the future.

The client can use CXF OAuthClientUtils to create Authorization MAC headers. All is needed is to provide references to ClientAccessToken representing the MAC token issued by AccessTokenService and HttpRequestProperties capturing the information about the current request URI:

```
String requestURI = "http://localhost:8080/calendar";
WebClient wc = WebClient.create(requestURI);

// represents client registration
OAuthClientUtils.Consumer consumer = getConsumer();
// the token issued by AccessTokenService
ClientAccessToken token = getToken();

HttpRequestProperties httpProps = new HttpRequestProperties(wc, "GET");
String authHeader = OAuthClientUtils.createAuthorizationHeader(consumer, token, httpProps);
wc.header("Authorization", authHeader);

Calendar calendar = wc.get(Calendar.class);
```

This code will result in something like:

```
GET /calendar HTTP/1.1
Host: localhost
Accept: application/xml
Authorization: MAC id="5b5c8e677413277c4bb8b740d522b378",
                   nonce="di3hvdf8",
                   mac="W7bdMZbv9UWOTadASIQHagZyirA="
                   ts="12345678"
```

where 'ts' attribute is used to pass a timestamp value.

Encrypted Tokens

Note: consider using JWE Encryption with JWT access tokens (see next section).

Typically, the tokens are persisted in the storage. The alternative approach is to completely encrypt the token state and return the encrypted representation back to a client: the processing time to do with the encryption and decryption might increase but the server wins on avoiding the DB / storage lookups.

CXF 3.0.0-milestone2 introduces the utility support for encrypting the state of BearerAccessToken and RefreshToken.

The tokens can be encrypted and decrypted with symmetric (secret) keys or certificates (public and private keys) and the combination of certificates and secret keys.

ModelEncryptionSupport can be used to encrypt the tokens using the custom serialization format.

Note that ServerAuthorizationGrant and Client can also be encrypted.

The simplest strategy is to encrypt and decrypt the tokens with the symmetric/secret keys. Every new token can be encrypted with a unique secret key or all of them can be encrypted with a single secret key. The utilities provide few methods for creating secret keys with the default and advanced properties, in addition there are many examples around on how to create the keys with the specific properties.

For example, see org.apache.cxf.rs.security.oauth2.grants.code.DefaultEncryptingCodeDataProvider and org.apache.cxf.rs.security.oauth2.provider.DefaultEncryptingOAuthDataProvider which are shipped starting from CXF 3.0.2.

Here is a typical code demonstrating how the encryption/decryption works:

```
SecretKey key = CryptoUtils.getSecretKey();

// create a new token, encrypt its state and return

ServerAccessToken token = new BearerAccessToken(client, 3600L);

String encryptedToken = ModelEncryptionSupport.encryptAccessToken(token, key);

token.setTokenKey(encryptedToken);

return token;

// decrypt a token given a token key

ModelEncryptionSupport.decryptAccessToken(this, encryptedToken, key);
```

JWT Tokens

Starting from CXF 3.1.8 some of CXF OAuthDataProvider implementations (JCache and JPA2 based, as well as EhCache 2.x prior to CXF 3.3.0) support Access Token representations in JWT. This means that ServerAccessTokens created by data providers are converted to a sequence of JSON JWT claims and then JWS signed and/or JWE encrypted.

Custom data providers can extend AbstractOAuthDataProvider to depend on the code which converts ServerAccessTokens to JWT and use JwtTokenUtils to convert JOSE token representations back to ServerAccessToken.

For example, here is how one can configure one of CXF data providers to use JWT:

```
<bean id="oauthProvider" class="org.apache.cxf.rs.security.oauth2.grants.code.DefaultEHCacheCodeDataProvider">
       <property name="useJwtFormatForAccessTokens" value="true"/>
</bean>
```

Additionally, in order to sign and/or encrypt, this provider can be injected with an instance of OAuthJoseJwtProducer or AccessTokenService endpoint where this provider is registered can be configured as follows:

```
<jaxrs:server id="oauthServer1" address="https://localhost:${testutil.ports.jaxrs-oauth2-serviceJwt}/services">
    <jaxrs:serviceBeans>
        <ref bean="tokenService"/>
    </jaxrs:serviceBeans>
    <!-- Sign -->
    <jaxrs:properties>
        <entry key="rs.security.signature.properties" value="org/apache/cxf/systest/jaxrs/security/alice.rs.
properties"/>
        <entry key="rs.security.signature.key.password.provider" value-ref="keyPasswordProvider"/>
    </jaxrs:properties>
</jaxrs:server>
```

Note that in this case Ehcache, JCache and JPA2 providers will still persist the complete ServerAccessToken representations - once JOSE sequence is created it becomes a new tokenId of the current ServerAccessToken, with the original tokenId becoming a JWT 'jti' claim.

The main advantage is that such tokens can be introspected locally at the resource server side, thus avoiding the remote token validation calls.

One can configure the providers (JCache only at the moment, as well as EhCache 2.x prior to CXF 3.3.0) to persist access tokens only as these newly created JOSE sequences:

```
<bean id="oauthProvider" class="org.apache.cxf.systest.jaxrs.security.oauth2.common.OAuthDataProviderImpl">
       <property name="useJwtFormatForAccessTokens" value="true"/>
       <property name="storeJwtTokenKeyOnly" value="true"/>
</bean>
```

Only a JOSE sequence representing a given ServerAccessToken will be persisted. The providers will recreate ServerAccessToken from this sequence when the token is needed by the runtime, while requiring much less storage to keep such tokens. In this case, if it is preferred, one can further optimize it not to even store these secure string token representations - however the major downside is that it will be impossible to manage such tokens from the administration consoles due to no record of such tokens will be available in the storage.

Resource server (RS) will need to make a decision how to validate this JWT token. It can continue validating it remotely with AccessTokenValidationService or TokenIntrsopectionService (see below for more info about these services) or if RS has an access to the keys used to sign/encrypt JWT then it can use a local JWT validation, example:

```
<bean id="jwtTokenValidator" class="org.apache.cxf.rs.security.oauth2.filters.JwtAccessTokenValidator"/>
<bean id="oAuthFilterLocalValidation" class="org.apache.cxf.rs.security.oauth2.filters.OAuthRequestFilter">
    <property name="tokenValidator" ref="jwtTokenValidator"/>
</bean>

<jaxrs:server
    depends-on="tls-config"
    address="https://localhost:${testutil.ports.jaxrs-oauth2-filtersJwt}/securedLocalValidation">
    <jaxrs:serviceBeans>
        <ref bean="serviceBean"/>
    </jaxrs:serviceBeans>
    <jaxrs:providers>
        <ref bean="oAuthFilterLocalValidation"/>
    </jaxrs:providers>
    <jaxrs:properties>
         <entry key="rs.security.signature.in.properties" value="org/apache/cxf/systest/jaxrs/security/alice.rs.
properties"/>
    </jaxrs:properties>
</jaxrs:server>
```

When to use JWT ? The pros are: might be easier to align with some newer OAuth2 related specifications, might be possible to avoid a remote validation call, possible OAuth2 server storage optimization. Cons: the extra cost of validating (or decrypting), access token value reported to and used by clients becomes larger. If JWS only is used - care should be taken to avoid putting some sensitive JWT claims given that JWS payload can be introspected.

See JAX-RS JOSE wiki page for more information on how to sign and encrypt JSON Web Tokens. Specifically, if you need to create JWT values in your custom providers, then have a look at this section: one can delegate to or extend **JoseJwtConsumer** or **JoseJwtProducer**. Addtionally org.apache.cxf.rs.security.oauth2.provider.**OAuthJoseJwtConsumer** (and **OAuthJoseJwtProducer**) can help in cases where OAuth2 Client secret is used as a key for HMAC based signatures or encryptions, while **OAuthServerJoseJwtConsumer** (and **OAuthServerJoseJwtProducer**) can also use OAuth2 Client certificates.

Custom tokens

If needed, users can use their own custom token types, with the only restriction that the custom token type implementations have to extend org.apache.cxf.rs.security.oauth2.common.ServerAccessToken.

Simple Tokens and Audience

Starting from CXF 2.7.7 an audience parameter is supported during the client token requests.

OAuthJSONProvider

org.apache.cxf.rs.security.oauth2.provider.OAuthJSONProvider is a JAX-RS MessageBodyWriter which supports returning ClientAccessToken and OAuthError representations to the client in a JSON format required by OAuth2 spec. It is also a JAX-RS MessageBodyReader that is used by client OAuthClientUtils (see below) to read the responses from AccessTokenService.

Register it as a provider with a JAXRS AccessTokenService endpoint.

Alternatively, if you prefer, a custom MessageBodyWriter implementation can be registered instead.

Access Token Validation Service

AccessTokenValidatorService

The AccessTokenValidatorService is a CXF specific OAuth2 service for accepting the remote access token validation requests. OAuthRequestFilter needs to be injected with AccessTokenValidatorClient which will ask AccessTokenValidatorService to return the information relevant to the current access token, before setting up a security context.

TokenIntrospectionService

The TokenIntrospectionService is a standard OAuth2 service for accepting the remote access token introspection requests. See RFC 7662. OAuthRequestFilter needs to be injected with AccessTokenIntrospectionClient.

TokenRevocationService

TokenRevocationService is a simple OAuth2 service supporting the clients wishing to revoke the access or refresh tokens they own themselves, please see OAuth2 Token Revocation Draft for more information.

TokenRevocationService and AccessTokenService share the same code which enforces that the clients have been correctly authenticated.

Note, OAuthDataProvider implementations processing a revocation request should simply ignore the invalid tokens as recommended by the specification which will let TokenRevocationService return HTTP 200 which is done to minimize a possible attack surface (specifically for bad clients not to see if their requests failed or succeeded) and throw the exceptions only if the token revocation feature is not currently supported.

DynamicRegistrationService

This service is available starting from CXF 3.1.8. It supports the dynamic client registration and management. At the moment some of the advanced registration properties are not yet processed and linked to the way the core OAuth2 services operate but the service will be enhanced as needed going forward.

AuthorizationMetadataService

This service is available starting from CXF 3.1.8. It supports OAuth2 server configuration queries at ".well-known/oauth-authorization-server".

Supported Grants

The following subsections briefly describe how the well-known grant types can be supported on the server side. Please also check the "Client Side Support" section on how to use the related AccessTokenGrant implementations to request the access tokens.

Authorization Code

As described above, AuthorizationCodeGrantService service and AuthorizationCodeDataProvider data provider can support a redirection-based Authorization Code flow.

The code that the client receives in the end of the redirection process will need to be exchanged for a new access token with AccessTokenService. CXF-based clients can use a helper AuthorizationCodeGrant bean to request a new access token with OAuthClientUtils.

Implicit

Implicit grant is supported the same way Authorization Code grant is except that no code is created, a token is issued immediately and returned to the client running within a web browser.

ImplicitGrantService service asks OAuthDataProvider data provider to issue a new token after a user has approved it.

Note the only difference is the use of ImplicitGrantService instead of AuthorizationCodeGrantService.

Also note that when an Implicit grant client (running within a browser) replaces the code grant for a new access token and tries to access the end user's resource, Cross Origin Resource Sharing (CORS) support will most likely need to be enabled on the end user's resource server.
The simplest approach is to register a CXF CORS filter, right before OAuth2 filter (see on it below).

Starting from CXF 2.7.5 it is possible to request ImplicitGrantService to return a registered Client id to the browser-hosted client. This is recommended so that the client can verify that the token is meant to be delivered to this client.

Client Credentials

Register ClientCredentialsGrantHandler handler with AccessTokenService for this grant be supported.

CXF-based clients can use a helper ClientCredentialsGrant bean to request a new access token with OAuthClientUtils.

Resource Owner Password Credentials

Register ResourceOwnerGrantHandler handler with AccessTokenService for this grant be supported.

CXF-based clients can use a helper ResourceOwnerGrant bean to request a new access token with OAuthClientUtils.

Refresh Token

The client can issue a refresh token grant if the current access token it owns has expired or been revoked and the refresh token was issued alongside with the access token which is now invalid and get the new, 'refreshed' access token. This can allow the client to avoid seeking a new authorization approval from the end user.

Register RefreshTokenGrantHandler handler with AccessTokenService for this grant be supported. Note this grant handler is only useful for refreshing the existing access token, so one or more of the other grant handlers (Authorization Code, Implicit, etc) will also have to be registered with AccessTokenService.

CXF-based clients can use a helper RefreshTokenGrant bean to request a new access token with OAuthClientUtils.

SAML and JWT Assertions

SAML2 assertions and JWT assertions can be used as token grants.

JWT assertion grants are supported in this package. JwtBearerAuthHandler can be used as a generic client authentication filter (where the client authenticated with JWT token as opposed to with a username:password pair, etc).

Please also see JAXRS OAuth2 Assertions section for more information.

Custom Grants

If you need to customize the way the well-known grant requests are handled then consider extending one of the grant handlers listed in the previous sub-sections.

Alternatively create a custom AccessTokenGrantHandler and register it with AccessTokenService. Additionally, consider providing a related AccessTokenGrant implementation for making it easy for the client code to request a new access token with this custom grant.

Redirection Flow Filters

AuthorizationRequestFilter implementations can be registered with AuthorizationCodeGrantService or ImplicitGrantService in order to pre-process code requests. For example, JwtRequestCodeFilter can be used to process JWS-signed or JWE-encrypted code requests.

AuthorizationCodeResponseFilter implementations can be registered with AuthorizationCodeService in order to post-process code responses.

AccessTokenResponse Filters

AccessTokenResponseFilter implementations can be registered with AccessTokenService in order to post-process access token responses. For example, OIDC IdToken can be added to a response with a IdTokenResponseFilter.

PreAuthorized access tokens

When working with the flows which require the end users/resource owners explicitly authorizing clients (for example, as in the case of redirection-based flows), using pre-authorized access tokens is one option to minimize the need for the end-user intervention.
OAuthDataProvider is always checked first if the pre-authorized access token for a given Client exists and if yes then it will be returned immediately, without starting the authorization process involving the end user (as required by some flows).

Consider providing a user interface which will let the end users/resource owners to pre-authorize specific clients early. Note, a CXF service for supporting the users pre-authorizing the clients or revoking the tokens for some of the clients may be introduced in the future.

Also note that using a refresh token grant may further help with minimizing the end user involvement, in cases when the current access token has expired.

Pre-registered scopes

Clients can register custom scopes they will be expected to use and then avoid specifying the scopes when requesting the code grants or access tokens. Alternatively it makes it easier to support so called wild-card scopes. For example, a client pre-registers a scope "update" and actually uses an "update-7" scope: Redirection-based services and access token grants can be configured to do a partial scope match, in this case, validate that "update-7" starts from "update"

Writing OAuthDataProvider

Using CXF OAuth service implementations will help a lot with setting up an OAuth server. As you can see from the above sections, these services rely on a custom OAuthDataProvider implementation.

The main task of OAuthDataProvider is to persist and generate access tokens. Additionally, as noted above, AuthorizationCodeDataProvider needs to persist and remove the code grant registrations. The way it's done is really application-specific. Consider starting with a basic memory based implementation and then move on to keeping the data in some DB.

Finally OAuthDataProvider may need to convert opaque scope values such as "readCalendar" into a list of OAuthPermissions. AuthorizationCodeGrantService and OAuth2 security filters will depend on it (assuming scopes are used in the first place).

Default Providers

CXF 3.1.7 ships JPA2 (JPAOAuthDataProvider and JPACodeDataProvider), Ehcache 2.x (DefaultEHCacheOAuthDataProvider and DefaultEHCacheCodeDataProvider) and JCache (JCacheOAuthDataProvider and JCacheCodeDataProvider) provider implementations which take care of all the persistence tasks: saving or removing registered clients, tokens and code grants. These providers can be easily customized.

Custom implementations can also extend AbstractOAuthDataProvider or AbstractCodeDataProvider and only implement their abstract persistence related methods or further customize some of their code.

OAuth Server JAX-RS endpoints

With CXF offering OAuth service implementations and a custom OAuthDataProvider provider in place, it is time to deploy the OAuth2 server.
Most likely, you'd want to deploy AccessTokenService as an independent JAX-RS endpoint, for example:

```
<!-- implements OAuthDataProvider -->
<bean id="oauthProvider" class="oauth.manager.OAuthManager"/>

<bean id="accessTokenService" class="org.apache.cxf.rs.security.oauth2.services.AccessTokenService">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<jaxrs:server id="oauthServer" address="/oauth">
   <jaxrs:serviceBeans>
      <ref bean="accessTokenService"/>
  </jaxrs:serviceBeans>
</jaxrs:server>
```

AccessTokenService listens on a relative "/token" path. Given that jaxrs:server/@adress is "/oauth" and assuming a context name is "/services", the absolute address of AccessTokenService would be something like "http://localhost:8080/services/oauth/token".

If the remote token validation is supported then have AccessTokenValidatorService added too:

```
<!-- implements OAuthDataProvider -->
<bean id="oauthProvider" class="oauth.manager.OAuthManager"/>

<bean id="accessTokenService" class="org.apache.cxf.rs.security.oauth2.services.AccessTokenService">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>
<bean id="accessTokenValidateService" class="org.apache.cxf.rs.security.oauth2.services.
AccessTokenValidatorService">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<jaxrs:server id="oauthServer" address="/oauth">
   <jaxrs:serviceBeans>
      <ref bean="accessTokenService"/>
      <ref bean="accessTokenValidateService"/>
  </jaxrs:serviceBeans>
</jaxrs:server>
```

The absolute address of AccessTokenValidateService would be something like "http://localhost:8080/services/oauth/validate".

AuthorizationCodeGrantService is easier to put where the application endpoints are. It can be put alongside AccessTokenService, but ideally an SSO based authentication solution will be also be deployed, for the end user to avoid signing in separately several times (see more in it below). Here is an example of AuthorizationCodeGrantService and ImplicitGrantService being collocated with the application endpoint:

```
<bean id="authorizationService" class="org.apache.cxf.rs.security.oauth2.services.AuthorizationCodeGrantService"
>
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<bean id="implicitService" class="org.apache.cxf.rs.security.oauth2.services.ImplicitGrantService">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<bean id="jaxrsService" class="org.myapp.MyService"/>

<jaxrs:server id="appServer" address="/myapp">
   <jaxrs:serviceBeans>
      <ref bean="jaxrsService"/>
      <ref bean="authorizationService"/>
      <ref bean="implicitService"/>
   </jaxrs:serviceBeans>
</jaxrs:server>
```

AuthorizationCodeGrantService listens on a relative "/authorize" path so in this case its absolute address will be something like "http://localhost:8080/services/myapp/authorize". This address and that of AccessTokenService will be used by third-party clients.

ImplictGrantService listens on a relative "/authorize-implicit" path

AuthorizationCode and Implicit Services on the same relative path

As has already been mentioned in the previous section, AuthorizationCodeGrantService and ImplictGrantService listen on two different relative paths: "/authorize" and "/authorize-implicit". Having both services available at different addresses may not always be preferred though. If preferred, one can use AuthorizationService 'container' service:

```
<bean id="authorizationService" class="org.apache.cxf.rs.security.oauth2.services.AuthorizationCodeGrantService"
>
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<bean id="implicitService" class="org.apache.cxf.rs.security.oauth2.services.ImplicitGrantService">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<util:list id="servicesList">
  <ref bean="authorizationService"/>
  <ref bean="implicitService"/>
</util:list>

<bean id="oauth2Service" class="org.apache.cxf.rs.security.oauth2.services.AuthorizationService">
    <property name="services" ref="servicesList"/>
</bean>

<bean id="jaxrsService" class="org.myapp.MyService"/>

<jaxrs:server id="appServer" address="/myapp">
   <jaxrs:serviceBeans>
      <ref bean="jaxrsService"/>
      <ref bean="oauth2Service"/>
   </jaxrs:serviceBeans>
</jaxrs:server>
```

See this application context for another example.

Third Party Client Authentication

When a client requests a token from Access Token Service, it needs to get authenticated. Providing its client_id and client secret as part of Basic Authorization scheme or posting them directly as form parameters are typical options, however other authentication schemes can easily be supported if required.

For example, using client certificates or assertions like SAML2 Bearer or JWT is all acceptable - the only additional requirement in this case is that a given security filter processing a specific authentication scheme maps the client credentials to an actual client_id - CXF Access Token Service will check a "client_id" property on the current message context as the last resort. Note that org.apache.cxf.rs.security.oauth2.provider.ClientIdProvider can be registered with AccessTokenService to facilitate the mapping between an authenticated client and its id expected by the data provider if the container or filter based authentication can not set a "client_id" contextual property.

If a Basic authentication scheme is used and neither the container or filter has authenticated the client AccessTokenService will request a Client from the data provider and compare the Client's secret against the password found in the Basic scheme data. org.apache.cxf.rs.security.oauth2.provider.ClientSecretVerifier is available starting from CXF 3.0.3 to support Clients saving only password hashes. Its org.apache.cxf.rs.security.oauth2.provider.ClientSecretHashVerifier (calculates a SHA-256 password hash and compares it with the Client's secret) or custom implementations can be registered with AccessTokenService.

Please see JAXRS OAuth2 Assertions section for more information on how it may work.

Client Certificate Authentication

If a 2-way TLS is used to authenticate a client, and the Client has a Base64 encoded representations of its X509Certificates available in its "applicationCertificates" property, then the AccessTokenService will do the additional comparison of these certificates against the ones available in the current TLS session.

Also, OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens is completely supported since CXF 3.1.12. After the client authenticates to the token service, the token service will insert a digest of the client's certificate in the issued (JWT based) access token. On the resource server side, the OAuthRequestFilter will check the digest of the client certificate against the value stored in the token claim, and will throw a 401 exception if they do not match.

User Session Authenticity

Redirection-based Authorization Code and Implicit flows depend on end users signing in if needed during the initial redirection, challenged with the client authorization form and returning their decision. By default, CXF will enforce the user session authenticity by keeping the session state in a servlet container's HTTPSession. If the alternative storage is preferred then you can register a new SessionAuthenticityTokenProvider with either AuthorizationCodeGrantService or ImplicitGrantService beans.

CXF ships JoseSessionTokenProvider which uses CXF JOSE to create a compact JWS and/or JWE sequence capturing all the data which need to be available when the user returns an authorization form decision and this secure sequence becomes a session token.

Keeping the state in the session

Note that SessionAuthenticityTokenProvider has been further updated in CXF 3.1.0 to support signing and/or encrypting some of the redirection properties that would otherwise have to be kept as HTML form hidden fields (see "Authorization Service" section).

CXF ships JoseSessionTokenProvider which uses CXF JOSE that can be used as a SessionAuthenticityTokenProvider which JWS-signs and/or JWE-encrypts the properties and saves the result in the session. The HTML authorization forms will only have to have an "authenticityToken" property which the provider will use to match the session signed/encryped data and decrypt and/or validate the session data.

Multiple Factor Verification

Note that SessionAuthenticityTokenProvider has been updated in CXF 3.0.2 to accept request parameters and a reference to the authenticated user. This allows for introducing a multiple factor session verification: when the provider created a session property it can for example sent a message to a user's mobile phone expect the authorization consent form return the sent value.

The other minor enhancement is that RedirectionBasedGrantService will check the authorization content form for the name of the form property that contains a session authentication property, using a "session_authenticity_token_param_name" property name. This allows for the 'rotation' of hidden form properties containing the actual session authenticity values.

Customizing End User Subject initialization

By default, redirection based authorization services will the the current CXF SecurityContext to initialize a subject representing the authenticated resource owner/end user. If the customization if needed: custom CXF filter can be used to create UserSubject and set it on the message or org.apache.cxf.rs.security.oauth2.provider.SubjectCreator interface implementation can be registered with either AuthorizationCodeGrantService or ImplicitGrantService.

Protecting resources with OAuth filters

OAuthRequestFilter request handler can be used to protect the resource server when processing the requests from the third-party clients. Add it as a jaxrs:provider to the endpoint which deals with the clients requesting the resources.

When checking a request like this:

```
Address: http://localhost:8080/services/thirdPartyAccess/calendar
Http-Method: GET
Headers:
{
  Authorization=[Bearer 5b5c8e677413277c4bb8b740d522b378],
  Accept=[application/xml]
}
```

the filter will do the following:

1. Retrieve a ServerAccessToken by delegating to a matching registered AccessTokenValidator. AccessTokenValidator is expected to check the validity of the incoming token parameters and possibly delegate to OAuthDataProvider to find the token representation - this is what the filter will default to if no matching AccessTokenValidator is found and the Authorization scheme is 'Bearer'.

2. Check the token has not expired

3. AccessToken may have a list of OAuthPermission. For every permission it will:

- If it has a uri property set then the current request URI will be checked against it
- If it has an httpVerb property set then the current HTTP verb will be checked against it

If an allPermissionsMatch property is set then the filter will check that all the token permissions have been met.

If a requestScopes property is set then the filter will check that all of the scopes are 'covered' by one or more token permissions.

4. Finally, it will create a CXF SecurityContext using this list of OAuthPermissions, the UserSubject representing the client or the end user who authorized the grant used to obtain this token.

This SecurityContext will not necessarily be important for some of OAuth2 applications. Most of the security checks will be done by OAuth2 filters and security filters protecting the main application path the end users themselves use. Only if you would like to share the same JAX-RS resource code and access URIs between end users and clients then it can become handy. More on it below.

Here is one example of how OAuthRequestFilter can be configured:

```
<bean id="oauthProvider" class="oauth.manager.OAuthManager"/>
<bean id="oauthFiler" class="org.apache.cxf.rs.security.oauth2.filters.OAuthRequestFilter">
  <property name="dataProvider" ref="oauthProvider"/>
</bean>

<bean id="myApp" class="org.myapp.MyApp"/>

<jaxrs:server id="fromThirdPartyToMyApp" address="/thirdparty-to-myapp">
   <jaxrs:serviceBeans>
      <ref bean="myApp"/>
  </jaxrs:serviceBeans>
  <jaxrs:providers>
      <ref bean="oauthFilter"/>
  </jaxrs:providers>

</jaxrs:server>
```

It will rely on an instance of OAuthDataProvider to get the information about the current access token and will validate it.
This option works OK for when it is easy to get the same OAuthDataProvider shared between this filter, as well as Authorization and AccessToken services. OAuthDataProvider can also be implemented such that it manages the information in the distributed manner so the above configuration option may scale well for more sophisticated deployments.

When one has Authorization and AccessToken service not collocated with the application endpoints, the following may work better:

```
<bean id="tvServiceClientFactory" class="org.apache.cxf.jaxrs.client.JAXRSClientFactoryBean">
    <property name="address" value="http://localhost:${http.port}/services/oauth/validate"/>
    <property name="headers">
        <map>
            <entry key="Accept" value="application/xml"/>
            <entry key="Content-Type" value="application/x-www-form-urlencoded"/>
        </map>
    </property>
</bean>

<bean id="tvServiceClient" factory-bean="tvServiceClientFactory" factory-method="createWebClient"/>

<bean id="tokenValidator" class="org.apache.cxf.rs.security.oauth2.filters.AccessTokenValidatorClient">
    <property name="tokenValidatorClient" ref="tvServiceClient"/>
</bean>

<bean id="oauthFiler" class="org.apache.cxf.rs.security.oauth2.filters.OAuthRequestFilter">
    <property name="tokenValidator" ref="tokenValidator"/>
</bean>

<bean id="myApp" class="org.myapp.MyApp"/>

<jaxrs:server id="fromThirdPartyToMyApp" address="/thirdparty-to-myapp">
   <jaxrs:serviceBeans>
      <ref bean="myApp"/>
  </jaxrs:serviceBeans>
  <jaxrs:providers>
      <ref bean="oauthFilter"/>
  </jaxrs:providers>
</jaxrs:server>
```

OAuth2 tokens and SOAP endpoints

If you use HTTP Authorization header or WS-Security Binary token to pass OAuth2 tokens to SOAP endpoints then OAuthRequestInterceptor can be used to validate such tokens. It is OAuthRequestFilter running as CXF interceptor which will work OOB for tokens passed with Authorization header and it can be easily extended to support WS-Security binary tokens

Scope-based access control

OAuthRequestFilter can be configured to do a lot of security checks as described above.

Additionally, starting from CXF 3.1.5 it is also possible to control which service methods can be invoked with a new Scopes annotation and OAuthScopesFilter (it needs to be registered alongside OAuthRequestFilter).

For example:

```
@Path("calendar")
public class CalendarResource {


   @PUT
   @Path("{id}")
   @Scopes("update-calendar")
   @ConfidentialClient
   public void updateCalendar(@PathParam("id") long id, Calendar c) {
       // update the calendar for a user identified by 'id'
   }
}
```

In this example a client will only be able to invoke the updateCalendar method if its access token contains an "update-calendar" scope and

it is a ConfidentialClient. As mentioned earlier, OAuthRequestFilter may be configured with the 'requestScopes' property but using the Scopes annotation can offer a more typed and fine-grained

access control.

How to get the user login name

When one writes a custom server application which needs to participate in OAuth2 flows, the major question which needs to be addressed is how one can access a user login name that was used during the end-user authorizing the third-party client. This username will help to uniquely identify the resources that the 3rd party client is now attempting to access. The following code shows one way of how this can be done:

```
import org.apache.cxf.rs.security.oauth2.utils.OAuthContextUtils;

@Path("/userResource")
public class ThirdPartyAccessService {

    @Context
    private MessageContext mc;

    @GET
    public UserResource getUserResource() {

        String endUserName = OAuthContextUtils.resolveUserName(mc);

        return findUserResource(endUserName)
    }

    private UserResource findUserResource(String endUserName) {
        // find and return UserResource
    }
}
```

The above shows a fragment of the JAX-RS service managing the access to user resources from authorized 3rd-party clients (see the Design Considerations section for more information).

The injected MessageContext provides an access to OAuthContext which has been set by OAuth2 filters described in the previous section. OAuthContext will act as a container of the information which can be useful to the custom application code which do not need to deal with the OAuth2 internals. OAuthContextUtils provides a number of utility methods for retrieving and working with OAuthContext.

Note that starting from CXF 2.7.6 it is also possible to inject OAuthContext as JAX-RS context:

```
import org.apache.cxf.rs.security.oauth2.common.OAuthContext;

@Path("/userResource")
public class ThirdPartyAccessService {

    @Context
    private OAuthContext context;

    @GET
    public UserResource getUserResource() {
        //....
    }
}
```

org.apache.cxf.rs.security.oauth2.provider.OAuthContextProvider will have to be registered as jaxrs:provider for it to work.

Client-side support

When developing a third party application which needs to participate in OAuth2 flows one has to deal with redirecting users to OAuth2 AuthorizationCodeGrantService, interacting with AccessTokenService in order to exchange code grants for access tokens as well as correctly building Authorization OAuth2 headers when accessing the end users' resources.

Advanced OAuth2 client applications

In a number of cases an OAuth2 client application supporting the code flow needs to have an OAuth2-specific code written directly. Such clients qualify as advanced given that writing such a code requires thel understanding of OAuth2 specifics. That said,

JAX-RS makes it straightforward to support the redirection, while OAuthClientUtils class makes it possible to encapsulate most of the complexity away from the client application code, so ultimately the code required to support is typically not that complex at all, while at the same it offers the most flexibility.

For example, the following custom code can be used by the third-party application:

```
public class OAuthClientManager {

        private WebClient accessTokenService;
    private String authorizationServiceURI;
    private Consumer consumer;

    // inject properties, register the client application...

        public URI getAuthorizationServiceURI(ReservationRequest request,
                                                      URI redirectUri,
                                                      /* state */String reservationRequestKey) {
            String scope = OAuthConstants.UPDATE_CALENDAR_SCOPE + request.getHour();
            return OAuthClientUtils.getAuthorizationURI(authorizationServiceURI,
                                                            consumer.getKey(),
                                                            redirectUri.toString(),
                                                            reservationRequestKey,
                                                            scope);
        }
        public ClientAccessToken getAccessToken(AuthorizationCodeGrant codeGrant) {
            try {
                return OAuthClientUtils.getAccessToken(accessTokenService, consumer, codeGrant);
            } catch (OAuthServiceException ex) {
                return null;
            }
        }

        public String createAuthorizationHeader(ClientAccessToken token) {
                return OAuthClientUtils.createAuthorizationHeader(consumer, token);
        }
}
```

The reason such a simple wrapper can be introduced is to minimize the exposure to OAuth2 of the main application code to the bare minimum, this is why in this example OAuthServiceExceptions are caught, presumably logged and null values are returned which will indicate to the main code that the request failed. Obviously, OAuthClientUtils can be used directly as well.

Note that in the above example, an instance of AuthorizationCodeGrant is passed as the last parameter to OAuthClientUtils.getAccessToken() method, alongside the references to the AccessTokenService client and OAuthClientUtils.Consumer bean keeping the client id and secret.

CXF provides the utility grant beans for all the grants it supports, see the information on grants above. Please use the appropriate grant bean relevant to your application when requesting an access token or create a custom AccessTokenGrant bean implementation.

For example, consider a case where a client who already owns an authorized access token and accessing the end user resource gets HTTP 401 error back and the client also owns a refresh token. Here is one possible way to handle it:

```
import javax.ws.rs.NotAuthorizedException;
import javax.ws.rs.core.HttpHeaders;

import org.apache.cxf.rs.security.oauth2.client.OAuthClientUtils;
import org.apache.cxf.rs.security.oauth2.client.OAuthClientUtils.Consumer;
import org.apache.cxf.rs.security.oauth2.grants.code.AuthorizationCodeGrant;
import org.apache.cxf.rs.security.oauth2.grants.refresh.RefreshTokenGrant;
import org.apache.cxf.rs.security.oauth2.common.ClientAccessToken;

// the pseudo-code for getting the access token
Consumer consumer = ...
AuthorizationCodeGrant codeGrant = ...

ClientAccessToken accessToken = OAuthClientUtils.getAccessToken(codeGrant, consumer);

WebClient endUserResourceClient = WebClient.create(endUserServerAddress);

endUserResourceClient.header(HttpHeaders.AUTHORIZATION,
                             OAuthClientUtils.createAuthorizationHeader(accessToken));
try {
   return endUserResourceClient.get();
} catch (NotAuthorizedException ex) {
    String refreshToken = accessToken.getRefreshToken();
    if (refreshToken != null) {
        // retry once

        // refresh the token
        accessToken = OAuthClientUtils.getAccessToken(new RefreshTokenGrant(refreshToken), consumer);

        // reset Authorization header
        endUserResourceClient.replaceHeader(HttpHeaders.AUTHORIZATION,
                             OAuthClientUtils.createAuthorizationHeader(accessToken));

        // try to access the end user resource again
        return endUserResourceClient.get();

    } else {
        throw ex;
    }

}
```

The client code directly dealing with OAuth2 specifics can be the most flelxible option: the client which has both access and refresh tokens can check the current access token expiry time and if it is known to have expiried then it can proactively

refresh the tokens, avoiding doing a futile HTTP request that is bound to return 401. Or/and indeed it can take care of JAX-RS NotAuthorizedException (401) and refresh the tokens. Sophisticated clients might want to check which scopes have been approved for a given access token and dynamically decide if a given HTTP service call can be made or not. Clients can also proactively revoke the tokens using a token revocation mechanism.

OAuth2 client applications with code-grant filters

The code in the previous section shows the client application code directly supporting OAuth2 dynamics (redirection, the access token acquisition). Starting from CXF 3.0.6 a simpler option is possible with the help of ClientCodeRequestFilter. This filter manages the initial redirection, exchanging code grants for tokens, persisting the request state, and then making the token and the state available to the application code, for example, the following code does not deal itself with the redirection or interacting with OAuth2 servers:

```
import org.apache.cxf.jaxrs.client.WebClient;
import org.apache.cxf.rs.security.oauth2.client.ClientTokenContext;

@Path("reserve")
public class ReservationService {

private WebClient socialService;
private WebClient restaurantService;

@GET
@Path("table")
@Produces("text/html")
public Response reserve(@Context ClientTokenContext context) {
    // Check if token is available
    if (context.getToken() == null) {
       return redirectToFailureHandler(NO_CODE_GRANT);
    }
    // Prepare Authorization header
    socialService.authorization(context.getToken());

    // Get the state that was captured by the filter before redirecting the user to OAuth2 server
    ReservationRequest request = context.getState(ReservationRequest.class);

    // Work with the service on behalf of a user
    Calendar c = null;
    try {
      c = socialService.get(Calendar.class);
    } catch (RuntimeException ex) {
      return redirectToFailureHandler(CALENDAR_ACCESS_PROBLEM);
    }

    CalendarEntry entry = c.getEntry(request.getHour());
    if (entry.getEventDescription() == null || entry.getEventDescription().trim().isEmpty()) {
        String address = restaurantService.post(new Form().param("name", request.getReserveName())
                                         .param("phone", request.getContactPhone())
                                         .param("hour", Integer.toString(request.getHour())),
                                          String.class);
        if (address == null) {
            return redirectToFailureHandler(NO_RESERVATION);
        }

        // update the user's calendar
        Response response = socialService.form(new Form().param("hour", Integer.toString(request.getHour()))
                                         .param("description", "Table reserved at " + address));
        boolean calendarUpdated = response.getStatus() == 200 || response.getStatus() == 204;

        return Response.ok(new ReservationConfirmation(address, request.getHour(), calendarUpdated))
                           .build();
        } else {
            return redirectToFailureHandler(CALENDAR_BUSY);
        }
    }
}
```

The filter is configured as follows:

```
<beans>

<jaxrs:server id="reservationsServer" address="/reservations">
    <jaxrs:serviceBeans>
       <ref bean="restaurantReserveService"/>
    </jaxrs:serviceBeans>
    <jaxrs:providers>
       <!-- other providers -->

       <bean class="oauth2.thirdparty.CustomClientTokenContextProvider"/>
       <bean class="org.apache.cxf.rs.security.oauth2.client.ClientCodeRequestFilter">
           <property name="authorizationServiceUri" value="http://localhost:8080/services/authorize"/>
           <property name="accessTokenServiceClient" ref="atServiceClient"/>
           <property name="startUri" value="reserve/table"/>
           <property name="clientCodeStateManager" ref="codeManager"/>
           <property name="consumer" ref="consumer"/>
       </bean>
    </jaxrs:providers>
</jaxrs:server>

<bean id="codeManager" class="oauth2.thirdparty.ClientCodeStateManagerImpl"/>

<!-- the consumer pre-registered with OAuth2 servers -->
<bean id="consumer" class="org.apache.cxf.rs.security.oauth2.client.Consumer">
    <property name="key" value="123456789"/>
    <property name="secret" value="987654321"/>
</bean>

<!-- WebClient for communicating with OAuth2 AccessTokenService -->
<jaxrs-client:client id="atServiceClient" serviceClass="org.apache.cxf.jaxrs.client.WebClient"
    address="http://localhost:8080/services/oauth2Token/token">
    <jaxrs-client:headers>
       <entry key="Accept" value="application/json"/>
    </jaxrs-client:headers>
</jaxrs-client:client>

</beans>
```

ClientCodeRequestFilter redirects to 'authorizationServiceUri' when a 'startUri' is matched.

In the above example the filter uses a custom 'clientCodeStateManager' (org.apache.cxf.rs.security.oauth2.client.ClientCodeStateManager implementation) for keeping the original request state before redirecting for it to be available later on to the application code - this is optional and is only needed if the redirection request depends on the request parameters (example, Oauth2 scope values are dynamic such as updateCalendar-7 where '7' is the hour) . By default, CXF ships some state managers out of the box, at the moment these are MemoryClientCodeStateManager and JoseClientCodeStateManager implementations, the latter signs and/or encrypts the request state and saves it in the HTTP session.

After the user is redirected back the filter exchanges the code for a new access token and sets this token and the original redirection state in ClientTokenContext. Note CllientTokenContext makes the original request state available to the application as MultivaluedMap.

CXF also ships a default ClientTokenContext implementation, a simple org.apache.cxf.rs.security.oauth2.client.ClientTokenContextImpl bean. Finally CXF ships org.apache.cxf.rs.security.oauth2.client.ClientTokenContextProvider to ensure ClientTokenContext can be available as JAX-RS context. Note though that the above configuration registers a custom ClientTokenContext provider ('oauth2.thirdparty.CustomClientTokenContextProvider') instead - this is optional and is only needed if it is preferred for the application code to access the state in a type safe way (example, via ReservationRequest type as shown above), such a provider can be implemented like this:

```
import javax.ws.rs.core.MultivaluedMap;
import org.apache.cxf.jaxrs.ext.ContextProvider;
import org.apache.cxf.message.Message;
import org.apache.cxf.rs.security.oauth2.client.ClientTokenContext;
import org.apache.cxf.rs.security.oauth2.client.ClientTokenContextProvider;
import org.apache.cxf.rs.security.oauth2.common.ClientAccessToken;

public class CustomClientTokenContextProvider extends ClientTokenContextProvider {
    @Override
    public ClientTokenContext createContext(Message m) {
        return new WrapClientTokenContext(super.createContext(m));
    }
    private static class WrapClientTokenContext implements ClientTokenContext {
        private ClientTokenContext ctx;
        public WrapClientTokenContext(ClientTokenContext ctx) {
            this.ctx = ctx;
        }
        @Override
        public MultivaluedMap<String, String> getState() {
            return ctx.getState();
        }
        @Override
        public <T> T getState(Class<T> cls) {
            if (ReservationRequest.class == cls) {
                MultivaluedMap<String, String> state = getState();
                return (T)new ReservationRequest(state.getFirst("name"),
                                              state.getFirst("phone"),
                                              Integer.parseInt(state.getFirst("hour")));
            }
            return ctx.getState(cls);
        }
        @Override
        public ClientAccessToken getToken() {
            return ctx.getToken();
        }
    }
}
```

ClientCodeRequestFilter can also be configured with org.apache.cxf.rs.security.oauth2.client.ClientTokenContextManager ('clientTokenContextManager' property) to support a case where the same user is returning with the same request and the access token granted previously has not expired yet. At the moment CXF ships only MemoryClientTokenContextManager with the JOSE-aware provider to follow. If the manager is registered and the context (and the access token) for the current user is found then the request goes ahead without the redirection. In this case the filter can pro-actively refresh the token if it has already expired or close to being expired ('expiryThreshold' property, example, if it is set to 5 secs and the filter sees that the token will expire in less than 5 sec then it will try to refresh).

ClientCodeRequestFilter can also be configured with a 'scopes' property when static OAuth2 scopes are used for all the users (note 'scope' is one of redirect parameters typically sent to OAuth2 servers)

Note a similar filter support is availble for client applications depending on OIDC RP authentication.

OAuth2 client authenticators for non-dynamic clients

Not all clients that may need to access an OAuth2-protected application server can be modified. Furthermore, not all OAuth2 clients can participate in advanced flows such as an authorization code flow and need to be initialized with access and refresh tokens.

CXF HTTPConduit HttpAuthSupplier supporting access and refresh tokens is shipped starting from CXF 3.0.5 .

org.apache.cxf.rs.security.oauth2.client.BearerAuthSupplier supports creating HTTP Authorization header from bearer access tokens, refreshing them proactively or in response to 401 failures and recreating HTTP Authorization from the refreshed token.

It is not possible to refresh a token from a JAX-RS ClientRequestFilter because such a filter does not handle HTTP responses so it can not detect 401 (returned by a server if the access token has expired), while HTTPConduit HttpAuthSupplier gets a chance to react to 401 and retry.

Here is a configuration example:

```
<beans>
<bean id="consumer" class="org.apache.cxf.rs.security.oauth2.client.Consumer">
   <property name="clientId" value="1"/>
   <property name="clientSecret" value="2"/>
</bean>
<bean id="bearerAuthSupplier" class="org.apache.cxf.rs.security.oauth2.client.BearerAuthSupplier">
   <!-- access token -->
   <property name="accessToken" value="12345678"/>
   <!-- refresh token and the info needed to use it to refersh the expired access token proactively or in
response to 401 -->
   <property name="refreshToken" value="87654321"/>
   <!--
       Set this property for the authenticator to check the access token expiry date and refresh the token
proactively.
       Note that this property can also become effective after the first token refresh as it is not known in
advance when
       the injected access token will expire
   -->
   <property name="refreshEarly" value="true"/>
   <!-- client OAuth2 id and secret - needed to use a refresh token grant -->
   <property name="consumer" ref="consumer"/>
   <!-- address of OAuth2 token service that supports a refresh token grant
   <property name="accessTokenServiceUri" value="https://server/oauth2/accessToken"/>
</bean>
<conduit name="*.http-conduit" xmlns="http://cxf.apache.org/transports/http/configuration">
  <authSupplier>
     <ref bean="bearerAuthSupplier"/>
  </authSupplier>
</conduit>
</beans>
```

At the moment only BearerAuthSupplier supporting bearer access tokens is available; authenticators supporting other well known token types will be provided in the future.

org.apache.cxf.rs.security.oauth2.client.CodeAuthSupplier is also shipped. It is similar to BearerAuthSupplier except that it is initailized with an authorization code grant obtained out of band, uses this grant to get the tokens and then delegates to BearerAuthSupplier. Example:

```
<beans>
<bean id="consumer" class="org.apache.cxf.rs.security.oauth2.client.Consumer">
   <property name="clientId" value="1"/>
   <property name="clientSecret" value="2"/>
</bean>
<bean id="codeAuthSupplier" class="org.apache.cxf.rs.security.oauth2.client.CodeAuthSupplier">
   <!-- authorization code -->
   <property name="code" value="12345678"/>

   <!-- Set this property for the authenticator to check the access token expiry date and refresh the token
proactively -->
   <property name="refreshEarly" value="true"/>
   <!-- client OAuth2 id and secret - needed to use a refresh token grant -->
   <property name="consumer" ref="consumer"/>
   <!-- address of OAuth2 token service that supports a refresh token grant
   <property name="accessTokenServiceUri" value="https://server/oauth2/accessToken"/>
</bean>
<conduit name="*.http-conduit" xmlns="http://cxf.apache.org/transports/http/configuration">
  <authSupplier>
     <ref bean="codeAuthSupplier"/>
  </authSupplier>
</conduit>
</beans>
```

Additionally, a basic JAX-RS 2.0 ClientRequestFilter, org.apache.cxf.rs.security.oauth2.client.BearerClientFilter, is shipped and is initialized with an "accessToken" property only. It might be used in cases where only a non-expiring access token is available.

Using a token that expires within ClientRequestFilter does not work as explained above. However BearerClientFilter might be enhanced to support the pro-active refreshment of access token in the future.

OAuth2 without the Explicit Authorization

Client Credentials is one of OAuth2 grants that does not require the explicit authorization and is currently supported by CXF.

OAuth Without a Browser

When an end user is accessing the 3rd party application and is authorizing it later on, it's usually expected that the user is relying on a browser.
However, supporting other types of end users is easy enough. Writing the client code that processes the redirection requests from the 3rd party application and AuthorizationCodeGrantService is simple with JAX-RS and additionally CXF can be configured to do auto-redirects on the client side.

Also note that AuthorizationCodeGrantService can return XML or JSON OAuthAuthorizationData representations. That makes it easy for a client code to get OAuthAuthorizationData and offer a pop-up window or get the input from the command-line. Authorizing the third-party application might even be automated in this case - which can lead to a complete 3-leg OAuth flow implemented without a human user being involved.

Reporting error details

This section lists all the error properties that can be returned to the client application. CXF OAuth2 services will always report a required 'error' property but will omit the optional error properties by default (for example, in case of access token grant handlers throwing OAuthServiceException initialized with OAuth Error which may have the optional properties set).
When reporting the optional error properties is actually needed then setting a 'writeCustomErrors' property to 'true' will help:

```
<bean id="oauthProvider" class="oauth2.manager.OAuthManager"/>

<bean id="accessTokenService" class="org.apache.cxf.rs.security.oauth2.services.AccessTokenService">
    <property name="dataProvider" ref="oauthProvider"/>
    <property name="writeCustomErrors" value="true"/>
</bean>
```

OAuth2 and JOSE

See JAX-RS JOSE wiki page for the information about JOSE and how it is supported in CXF.

CXF OAuth2 depends on its JOSE implementation which is referred to the sections above. Particularly:

- JoseSessionTokenProvider can be used as a custom SessionAuthenticityTokenProvider with AuthorizationCodeGrantService or ImplicitService
- JoseClientCodeStateManager can be used with ClientCodeRequestFilter in confidential client web applications.
- JWT assertion client and server grant handlers are shipped in this package.
- JwtRequestCodeFilter and JwetRequestCodeGrant are provided to support securing authorization code and implicit flow request properties.
- Initial utility code for representing JWT as access tokens is provided and to be extended further.
- A number of utiliy classes for working with JWT are available: OAuthJoseJwtConsumer, OAuthJoseJwtProducer and OAuthServerJoseJwtProducer. These classes extend JOSE producer and consumer helpers and support the use of OAuth2 Client secrets and public certificates in JWS or JWE operations.

OAuth2 and OIDC

CXF ships OIDC RP and IDP service code which depends on its OAuth2 and JOSE implementations. See this page for more information.

Design considerations

This section will talk about various design considerations one need to take into account when deploying OAuth-based solutions.

Controlling the Access to Resource Server

One of the most important issues one need to resolve is how to partition a URI space of the resource server application.

We have two different parties trying to access it, the end users which access the resource server to get to the resources which they own and 3rd party clients which have been authorized by the end users to access some of their resources.

In the former case the way the authentication is managed is completely up to the resource server application: basic authentication, two-way TLS, OpenId (more on it below), you name it.

In the latter case an OAuth filter must enforce that the 3rd party client has been registered using the provided client key and that it has a valid access token which represents the end user's approval.

Letting both parties access the resource server via the same URI(s) complicates the life for the security filters but all the parties are only aware of the single resource server URI which all of them will use.

Providing different access points to end users and clients may significantly simplify the authentication process - the possible downside is that multiple access points need to be maintained by the resource server.

Both options are discussed next.

Sharing the same access path between end users and clients

The first problem which needs to be addressed is how to distinguish end users from third-party clients and get both parties authenticated as required. Perhaps the simplest option is to extend a CXF OAuth2 filter (JAX-RS or servlet one), check Authorization header, if it is OAuth2 then delegate to the superclass, alternatively - proceed with authenticating the end users:

```
public class SecurityFilter extends org.apache.cxf.rs.security.oauth2.filters.OAuthRequestFilter {
   @Context
   private HttpHeaders headers;

   public Response handleRequest(ClassResourceInfo cri, Message message) {
       String header = headers.getRequestHeaders().getFirst("Authorization");
       if (header.startsWith("Bearer ")) {
           return super.handleRequest(cri, message);
       } else {
           // authenticate the end user
       }
   }

}
```

The next issue is how to enforce that the end users can only access the resources they've been authorized to access.
For example, consider the following JAX-RS resource class:

```
@Path("calendar")
public class CalendarResource {

   @GET
   @Path("{id}")
   public Calendar getPublicCalendar(@PathParam("id") long id) {
       // return the calendar for a user identified by 'id'
   }

   @GET
   @Path("{id}/private")
   public Calendar getPrivateCalendar(@PathParam("id") long id) {
       // return the calendar for a user identified by 'id'
   }

   @PUT
   @Path("{id}")
   public void updateCalendar(@PathParam("id") long id, Calendar c) {
       // update the calendar for a user identified by 'id'
   }
}
```

Let's assume that the 3rd party client has been allowed to read the public user Calendars at "/calendar/{id}" only, how to make sure that the client won't try to:
1. update the calendar available at the same path
2. read the private Calendars available at "/calendar/{id}/private"

As noted above, OAuthPermission has an optional URIs property. Thus one way to solve the problem with the private calendar is to add, say, a uri "/calendar/{id}" or "/calendar/1" (etc) property to OAuthPermission (representing a scope like "readCalendar") and the OAuth filter will make sure no subresources beyond "/calendar/{id}" can be accessed. Note, adding a "*" at the end of a given URI property, for example, "/a*" will let the client access "/a", "/a/b", etc.

Solving the problem with preventing the update can be easily solved by adding an httpVerb property to a given OAuthPermission.

One more option is to rely on the role-based access control and have @RolesAllowed allocated such that only users in roles like "client" or "enduser" can invoke the getCalendar() method and let only those in the "enduser" role access getPrivateCalendar() and updateCalendar(). OAuthPermission can help here too as described in the section on using OAuth fiters.

Providing different access points to end users and clients

Rather than letting both the end users and 3rd party clients use the same URI such as "http://myapp.com/service/calendars/{id}", one may want to introduce two URIs, one for end users and one for third-party clients, for example, "http://myapp.com/service/calendars/{id}" - for endusers, "http://myapp.com/partners/calendars/{id}" - for the 3rd party clients and deploy 2 jaxrs endpoints, where one is protected by the security filter checking the end users, and the one - by OAuth filters.

Additionally the endpoint managing the 3rd party clients will deploy a resource which will offer a resticted URI space support. For example, if the application will only allow 3rd party clients to read calendars then this resource will only have a method supporting @GET and "/calendar/{id}".

Single Sign On

When dealing with authenticating the end users, having an SSO solution in place is very handy. This is because the end user interacts with both the third-party and its resource server web applications and is also redirected from the client application to the resource server and back again. Additionally, the end user may need to authenticate with Authorization service if it is not collocated with the application endpoints. OpenID or say a WebBrowser SSO profile can help.

CXF 2.6.1 provides an initial support for a SAML2 SSO SP profile. This will make it easier to minimize a number of sign ins to a single attempt and run OAuth2 Authorization servers separately from the application endpoints.

CXF 3.1.7 offers OpenId Connect RP support.