



Platform, Services, and Utilities

Generated on: 2024-12-03 11:15:13 GMT+0000

SAP Commerce | 2205

Public

Original content: https://help.sap.com/docs/SAP_COMMERCE/d0224eca81e249cb821f2cdf45a82ace?locale=en-US&state=PRODUCTION&version=2205

Warning

This document has been generated from the SAP Help Portal and is an incomplete version of the official SAP product documentation. The information included in custom documentation may not reflect the arrangement of topics in the SAP Help Portal, and may be missing important aspects and/or correlations to other topics. For this reason, it is not for productive use.

For more information, please visit the <https://help.sap.com/docs/disclaimer>.

OAuth2

The `oauth2` core extension has replaced the `webservicescommons/oauthauthorizationserver` extension. It exposes the HTTP endpoint as an authorization server. It doesn't introduce any new significant functionalities.

Enabling and Configuring oauth2

To enable the authorization server, add the `oauth2` extension entry into the `localextensions.xml` file:

```
<extension name="oauth2" />
```

By default, the web root is `authorizationserver/` but you can change it using the `oauth2.webroot=...` property.

To configure the `oauth2` extension, in the `project.properties` file use the following properties:

Property Name	Description	Type	Default Value
<code>webservicescommons.required.channel</code>	Channel protocol for oauth/* endpoints	http/https	https
<code>oauthauthorizationserver.tokenServices.reuseRefreshToken</code>	Specifies if a new refresh token should be created during refreshing an access token	boolean	false
<code>oauth2.supportRefreshToken</code>	Enables the refresh token	boolean	true
<code>oauth2.refreshTokenValiditySeconds</code>	Refresh token time-to-live	seconds	2592000 (30 days)
<code>oauth2.cleanupAccessToken.maxRows</code>	Max rows fetched on cleaning up the obsolete access token table	int	100
<code>oauth2.accessTokenValiditySeconds</code>	Access token time-to-live	seconds	43200 (12 hours)
<code>oauth2.optimize.accesstoken.save.enabled</code>	Optimizes the performance of the <code>oauth2</code> extension by ensuring that the <code>OAuthAccessTokenModel</code> object isn't updated when the <code>saveAccessToken</code> method is called and there are no changes to the <code>OAuth2Authentication</code> object.	boolean	true
<code>oauth2.accesstoken.save.retry</code>	Ensures that the system retries to save access tokens in situations when an attempt to save such tokens ends with the	boolean	true

Property Name	Description	Type	Default Value
	ModelSavingException exception being thrown. Such situations are a result of duplicate access token IDs being created by two or more threads trying to create the same access token in HybrisOAuthTokenStore .		
<code>oauth2.authorizationcode.stored.as.sha.signature</code>	Enables storing authorization codes as an SHA signature.	boolean	true
<code>oauth2.authorizationcode.length</code>	Allows you to configure the length of generated authorization codes.	int	30
<code>oauthauthorizationserver.tokenServices.refreshWithLock</code>	<p>Prevents the DuplicateKeyException exception caused by authenticationIdIdx index unique constraint violation. With selective thread locking implemented for HSQLDB and database row locking implemented for databases other than HSQLDB, refresh tokens cannot longer be consumed more than once by concurrent requests.</p> <p>For databases other than HSQLDB, the property also introduces database row locking for refresh tokens before refresh tokens are removed, which prevents potential deadlocks that could occur when access tokens are refreshed and refresh tokens are revoked at the same time.</p>	boolean	true

In the [System/OAuth](#) tab in Backoffice, you can manage OAuth clients and access tokens.

Deserialised Access Tokens after Spring Security Upgrade


Since all access tokens are serialized in the database, they are prone to deserialization when Spring Security is upgraded. Remove all your access tokens whenever Spring Security is upgraded.

For more information on upgraded libraries, see [Additional Information](#).

Authorization Server

The authorization server exposes two endpoints:

- /oauth/authorize
- /oauth/token

For details on usage and interfaces, see the RFC specification at <https://tools.ietf.org/html/rfc6749#section-1.3.3>  .

Response Encoding

Response content type might be in a JSON format, for example:

/oauth/token - proper credentials - json

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{"access_token":"a084da1c-72af-4cf7-a461-986de3f3070f","token_type":"bearer","expires_in":34191,"scope":null}
```

or in XML, for example:

/oauth/token - proper credentials - xml

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<oauth><access_token>a084da1c-72af-4cf7-a461-986de3f3070f</access_token><expires_in>33966</expires_in><scope></scope></oauth>
```

The same applies to errors:

/oauth/token - bad credentials - json

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="hybris", error="invalid_client", error_description="Bad client credentials"
Content-Type: application/json;charset=UTF-8

{"error":"invalid_client","error_description":"Bad client credentials"}
```

/oauth/token - bad credentials - xml

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="hybris", error="invalid_client", error_description="Bad client credentials"
Content-Type: application/xml;charset=UTF-8

<oauth><error_description>Bad client credentials</error_description><error>invalid_client</error><scope></scope></oauth>
```

Resource Server

There are two ways to validate a client token on custom web services. You can use the `ywebservices` template to generate a restful extension with a configuration that is already set up for integrating the resource server present in Platform.

You can also prepare your configuration manually:

Manual setup

```
<!-- load oauth2 definitions for resource server -->
<import resource="classpath*:oauth2-resource-spring.xml"/>
```

```

<!-- specify the filter change, the 'id' must be the same on web.xml filter -->
<bean id="oauth2SecurityFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterChain"
    <constructor-arg>
        <list>
            <ref bean="springSecurityFilterChain"/>
        </list>
    </constructor-arg>
</bean>

<!-- spring security. the pattern will be the one specific of your application -->
<http pattern="/rest/**" create-session="never"
    entry-point-ref="oauthAuthenticationEntryPoint"
    access-decision-manager-ref="accessDecisionManager"
    xmlns="http://www.springframework.org/schema/security">
    <anonymous enabled="false"/>
    <intercept-url pattern="/rest/**" access="IS_AUTHENTICATED_FULLY" method="GET"/>
    <custom-filter ref="resourceServerFilter" before="PRE_AUTH_FILTER"/>
    <access-denied-handler ref="oauthAccessDeniedHandler"/>
</http>

```

Preventing Brute Force Attacks

Using the `oauth2.maxAuthenticationAttempts` property, you can set the maximum number of failed client authentication attempts. If the max number of failed authentication attempts is reached, the `oauth2` client is disabled. The property isn't enabled by default. You can set it in `local.properties`, as shown in the example:

```
oauth2.maxAuthenticationAttempts=10
```

The `oauth2.maxAuthenticationAttempts` property defines common configuration for all OAuth clients.

Once a client is disabled, it's not possible to access resources using an access token previously issued for this client. The client can also no longer be used in OAuth flows, which means that it's impossible to issue a new access token or refresh an existing one by using this client in the request.

For disabled clients, each subsequent authentication attempt results in a 401 HTTP error response similar to the following:

```
{"error":"unauthorized","error_description":"Authentication for clientId:'e2eTests' is disabled"}
```

You can re-enable disabled clients in the **Clients** tab under **OAuth** in Backoffice. Once your client is re-enabled, access tokens that are issued for this client can be used again if they haven't already expired.

Preventing Brute Force Attacks for Public Clients

The `oauth2.maxAuthenticationAttempts` configuration property is also applied for OAuth public clients. Public clients are not required to provide any password parameters in their authentication requests.

Once a public client is disabled, it's not possible to access resources using an access token previously issued for this client. The client can also no longer be used in OAuth flows, which means that it's impossible to issue a new access token or refresh an existing one by using this client in the request.

i Note

Providing non-empty passwords in authentication requests for public clients causes authentication errors.

For information on prevention against user login brute force attacks, see [User Account](#).

Disabling Client-side SSL Certificate Checks and Hostname Verification

If you want to switch to HTTPS for the token endpoint and other RESTful resources that our API offers, but still do not want to use an official server certificate, you need to disable the client-side checks. The following code shows you how this task can be accomplished.

i Note

Keep in mind that this is to be used only during development and once a real certificate replaces the non-official certificate that ships with the SAP Commerce development package, you need to remove this code.

The `DummyHostnameVerifier` disables hostname checks and simply verifies all hostnames:

```
package de.hybris.platform.ycommercewebservices.test.groovy.webservicetests;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;

public class DummyHostnameVerifier implements HostnameVerifier {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        return true;
    }
}
```

The `DummyTrustManager` overrides all checks and trusts all clients, even if the certificate cannot be verified:

```
package de.hybris.platform.ycommercewebservices.test.groovy.webservicetests;

import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

import javax.net.ssl.X509TrustManager;

public class DummyTrustManager implements X509TrustManager
{
    @Override
    public void checkClientTrusted(final X509Certificate[] arg0, final String arg1) throws CertificateException
    {
        //TODO unimplemented
    }

    @Override
    public void checkServerTrusted(final X509Certificate[] arg0, final String arg1) throws CertificateException
    {
        //TODO unimplemented
    }

    @Override
    public X509Certificate[] getAcceptedIssuers()
    {
        return null;
    }
}
```

Before you can create `<HttpsURLConnection>`, you need to modify your SSL context to use instances of the above classes.

<Applying DummyHostnameVerifier and DummyTrustManager>

```
def trustManager = new DummyTrustManager()
def hostnameVerifier = new DummyHostnameVerifier();
def sc = javax.net.ssl.SSLContext.getInstance("SSL");
sc.init(null, [trustManager] as X509TrustManager[], new java.security.SecureRandom());
javax.net.ssl.HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
javax.net.ssl.HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier);
```

Roles

OAuth 2.0 defines the following roles:

- **Resource Owner:** An entity that can grant access to a protected resource. When the resource owner is a person, then it is called: end-user.
- **Resource Server:** The server that hosts the protected resources, capable of accepting and responding to protected resource requests using the access tokens.
- **Client:** An application making protected resource requests on behalf of the resource owner and with its authorization. The term client does not imply any particular implementation characteristics (for example, whether the application runs on a server, desktop, or any other particular device).
- **Authorization Server:** The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The authorization server is defined in the [OAuth2](#) and the example resource server is configured in [ycommercewebservices Extension](#) and [webservices Extension](#).

Flows

OAuth 2.0 comes with four flows. SAP Commerce supports all of them:

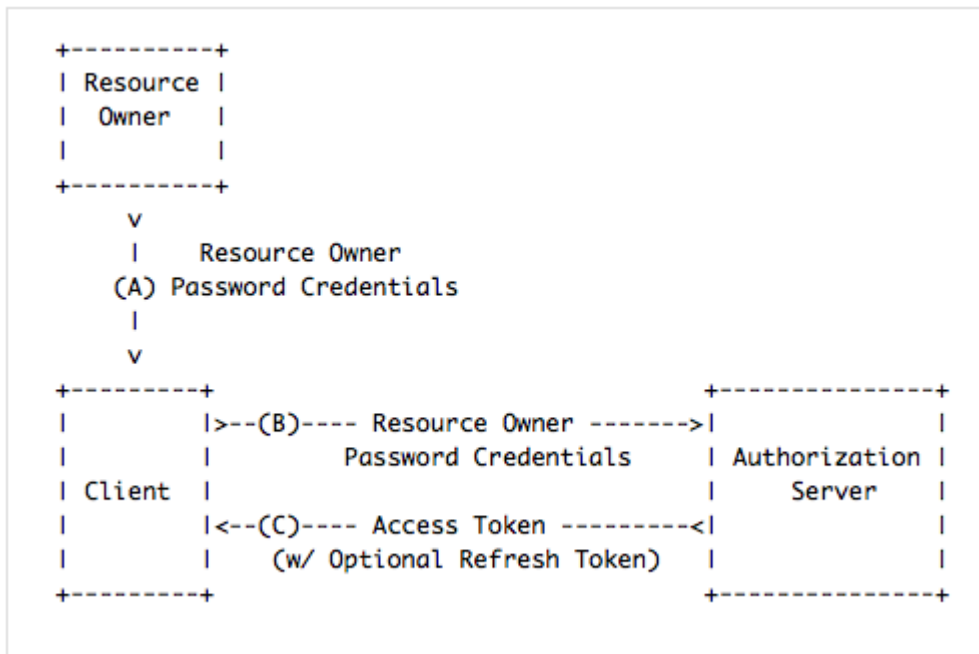
- As the **Resource Owner Password Flow** holds a user's `<username>` and `<password>`, it is less secure and not for third-party applications.
- The **Authorization Code Flow** is best to use if the web application can keep the `<client_secret>`.
- The **Implicit Flow** does not need any authorization tokens. As such, it is easier but less secure. JavaScript running within a browser is less trusted, and the refresh tokens are not issued. This is suited for the client-side web applications that need temporary access.
- The **Client Credentials Flow** gives the client access to the resources it owns.

Depending on your client application, you need to choose the appropriate flow. You also need to disable the other flows that you do not use.

Resource Owner Password Flow

This flow is somewhat similar to the basic authentication flow but it has a few benefits. It is typically used for trusted mobile applications, such as mobile Android or iOS applications. The flow includes sending the user's `<username>` and `<password>` to the token endpoint in exchange for an `<access_token>`. Replying with a refresh token is optional. The mobile `<client>` otherwise has to keep the username and password for long-lived access.

The flow needs a `<username>` and `<password>` for the `<access_token>`. However, keep in mind that the API provider provides access tokens combined with a `<refresh_token>`. The client therefore does not need to save the username and password, but it only has to pass this information on. The `<access_token>` and `<refresh_token>` need to be persisted locally, which is better than storing user credentials. The following diagram describes this flow.



Detailed description of the presented flow:

1. **Step (A):** The *<client>* receives the *<username>* and *<password>*. In this step, the user enters this information directly into the client application. Note that users must have a way to identify the application as being the official application they can trust.
2. **Step (B):** Next, the client application makes a request to the Authorization Server, for example the `/oauth/token` endpoint. There are two ways the *<client_id>* and *<client_secret>* can be sent along: either in a regular basic authentication request header, or as a part of the parameters passed in the request payload (that is, the request body). See the list of parameters to be passed:
 - *<client_id>* and *<client_secret>*: Either passed as parameters or as a basic authentication header. Basic authentication means that *<client_id>* and *<client_secret>* are treated as username and password, concatenated using a colon (:) and then *<Base64>* encoded. This value is then used as a part of the authorization request header, for example: `Authorization: Basic <Base64-encoded username:password>`
 - *<username>* and *<password>*: Credentials for the resource owner, the user's real credentials.

i Note

If One-Time Password for Customer Login is enabled and the token is issued for a Customer type user, then *<Token Id>* value is expected as *<username>* and *<Token Code>* value as *<password>*. For more information about One-Time Password for Customer Login, see [One-Time Password](#).

- *<grant_type>*: Needs to be set to *<password>* for this flow.

3. **Step (C):** The authentication server returns the *<access_token>* with an optional *<refresh_token>*.

The following code sample describes this flow.

```

import com.google.appengine.api.urlfetch.*
import groovy.json.*
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'

if (!params.username)
{
    out << "No username given..."
    return
  
```



```

}

if (!params.password)
{
    out << "No password given..."
    return
}

def username = URLEncoder.encode(params.username, 'UTF-8')
def password = URLEncoder.encode(params.password, 'UTF-8')

HTTPResponse res
URL tokenURL = "http://localhost:9001/authorization/oauth/token".toURL()

if (params.basic)
{
    //direct exchange of username and password for access token
    def headers = [Authorization:"Basic ${"${client_id}:${client_secret}".toString().bytes.encodeBase64()}"]
    res = tokenURL.post(deadline: 30, headers: headers, payload:"grant_type=password&username=${username}&password=${password}")
}
else
{
    //direct exchange of username and password for access token
    res = tokenURL.post(deadline: 30, payload:"client_id=${client_id}&client_secret=${client_secret}&grant_type=password&username=${username}&password=${password}")
}

out << res.text

```

The above code is written in Groovy and uses the Gaelyk Web Framework. This is, however, only for demonstration purposes. Typically, the code would not be a part of the server-side web application, but it serves as an example of how the single request against the authorization server is performed. First, `<client_id>` and `<client_secret>` need to be known for the code issuing the request. In this case, `<username>` and `<password>` are passed as parameters. In a real scenario, the credentials would be entered into a form through the application user interface.

Depending on the basic parameter, we either pass the username and password in a HTTP Post request as the authorization header or as a part of the parameters in the body of the request. The above code simply prints the response, which shall be a JSON-formatted document like in the example below.

```

{"access_token":"f2288c2b-1d08-4aac-a210-d62c0901f915","token_type":"bearer","refresh_token":"3f945b1c-1d08-4aac-a210-d62c0901f915","expires_in":3600}

```

Using CURL For Resource Owner Password Flow

During the development phase, you may want to quickly try out a RESTful request. If you run this request against an OAuth 2.0 protected resource, you need an `<access_token>`. Unfortunately, OAuth 2.0, just like basic authentication, is not supported by the browser. The easiest option is to use cURL, the command-line utility for HTTP requests in combination with the resource owner password flow.

The token endpoint needs to be HTTPS in the production environment, but during the development phase the following example should work fine.

```

curl -X POST -d "client_id=mobile_android&client_secret=secret&grant_type=password&username=demo&password=secret" http://localhost:9001/authorization/oauth/token

```

This will result in a JSON response with the `<access_token>`.

```

{
    "access_token": "a503faf9-45b5-4fec-8334-337284a66ea4",
    "token_type": "bearer",
    "refresh_token": "486adfdde-757b-4d37-81d7-446c2ec4bd91",
    "expires_in": 43199
}

```

Next, if you want to access a protected resource, you have to pass the authorization header. The following code sample shows how to access the *<current user>* resource.

```
curl --header "Authorization: Bearer a503faf9-45b5-4fec-8334-337284a66ea4" http://localhost:9001/r
```

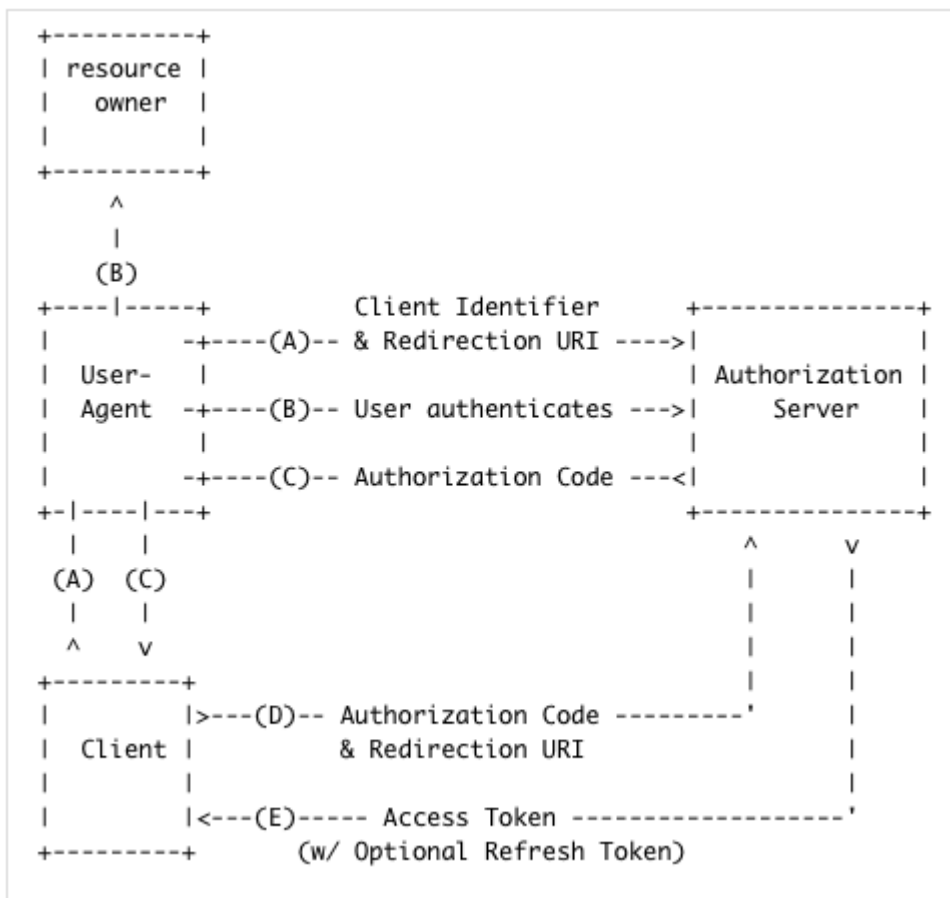
The response is presented in the next sample below.

```
{
    "uid": "demo",
    "name": "demo",
    "firstName": "Klaus",
    "lastName": "Demokunde"
    ...
}
```

Authorization Code Flow Or Server-side Flow

The flow includes sending the client user by redirecting to the provider's login and authorization page. Next, it redirects back to the web application and passes an authorization code in the URL parameters. Then, it can be exchanged for *<access_token>* which needs to be passed in the HTTP request headers to obtain access to the user's data. With the *<access_token>* and information about its expiry, there is also a *<refresh_token>* issued. In the next step, a *<refresh_token>* can be exchanged against a new *<access_token>* for a long-lived access.

The following flow diagram describes this flow.



Detailed description of the presented flow:

- Step (A):** Redirect the user to the authorization server (typical endpoint should be `/oauth/authorize`). All communication should occur through HTTPS so that *<client_id>* contained in the URL is safe. The following parameters have to be included:

- `<response_type>`: The value needs to be `<code>`, as we use the authorization code flow.
- `<client_id>`: For SAP Commerce web services, the OAuth 2.0 client has to be manually set up in the Spring Security OAuth2 XML configuration.
- `<client_secret>`: The `<secret>` for the `<client_id>`, and it needs to be on the server-side so that users cannot see it.
- `<redirect_uri>`: Optional, but recommended. When a user is redirected to the authorization endpoint, this value needs to be passed, and it has to match the server configuration settings.
- `<scopes>`: Used to allow different access levels.
- `<state>`: Optional, but recommended. Used to overcome CSRF (cross-site request forgery) attacks.

The following code sample uses the described parameters.

```
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'

def scopes = [
    'customer'
]

def state = new Random(System.currentTimeMillis()).nextInt().toString()
request.getSession(true).setAttribute('state', state)

redirect "http://localhost:9001/authorizationserver/oauth/authorize?client_id=${client_
```

2. **Step (B):** The user authenticates himself and is granted authorization. This can be a two-step process: if the user is not logged in, then he has to log in first and then provide access to the `<client>`.

3. **Step (C):** The authorization server redirects back to the web application. The parameters passed in this redirect to the application are:

- **code:** After the `<grant_type>` is set to `<code>`, the authorization code is accessible. Later, `<code>` is used to request the `<access_token>`.
- **state:** The `<state>` specified in the first redirect to the authorization server. It should be checked with the session if the value is the same.

4. **Step (D):** The web application verifies the redirect and exchanges the authorization code for an access token. The following code snippet is an example of how it works.

```
import com.google.appengine.api.urlfetch.*
import groovy.json.*
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'

if (!params.code)
{
    out << "User denied access."
    return
}

if (!params.state)
{
    out << "State parameter missing, somethign is wrong..."
    return
}

if (!session)
{
```

```

        out << "No session, how weird!"
        return

    }

    if (params.state != session.state)
    {
        out << "State does not match! (${params.state} != ${session.state})"
        return
    }

    def code = params.code

    //exchange code for real oauth token
    URL tokenURL = "http://localhost:9001/authorizationserver/oauth/token".toURL()
    HTTPResponse res = tokenURL.post(deadline: 30, payload:"code=${code}&client_id=${client_id}&c

    out << res.text

```

The above code first verifies that the `<code>` and `<state>` parameters are there and then checks whether the current state is equal to the session's state. Then the final request is made to obtain the `<access_token>`. The parameters passed into the URL are:

- `<code>`: The obtained code.
- `<grant_type>`: Tells the token endpoint we want the access token. This is now set to `<authorization_code>`.
- `<redirect_uri>`: Used for consistency.
- `<client_id>` and `<client_secret>`: Used for identifying the client application.

5. **Step (E)**: Eventually, access token is passed on to the client. The output looks like in the following sample.

```

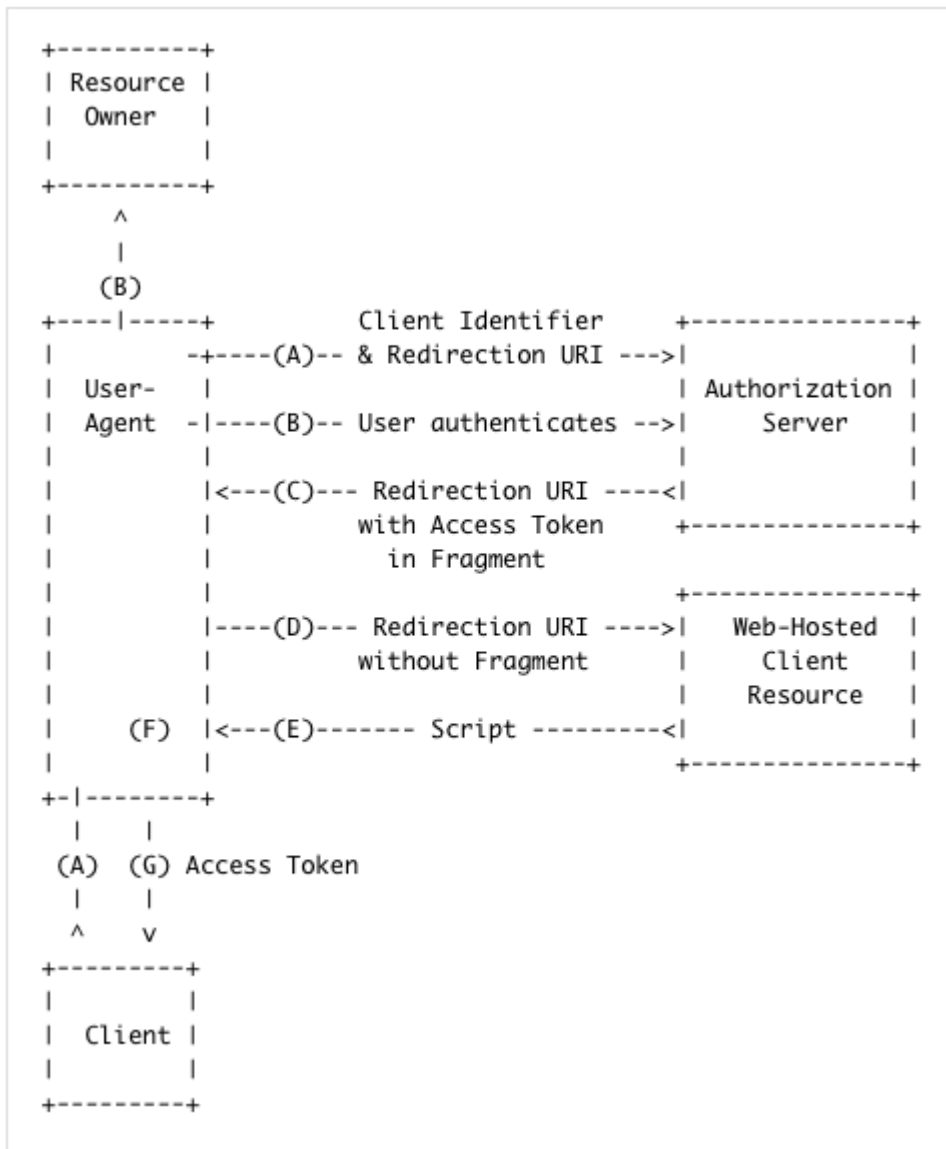
{"access_token": "865b3ecf-54d3-413c-951e-fae0b00c906f", "token_type": "bearer", "refresh_token":

```

Implicit Flow Or Client-Side Flow

In the Implicit Flow (also called, Implicit Grant Flow), the required access token is sent back to the client application without the need of an authorization request token. This makes the flow easier, but also less secure. As the client application, which is typically JavaScript running within a browser, is less trusted, no refresh tokens for long-lived access are returned. You should use this flow for the client-side web applications (JavaScript clients) that need temporary access (for example, a few hours) to the user's data. Returning an access token to the JavaScript clients also means that your browser-based application needs to take special care. Take into account the XSS (Cross-Site Scripting) attacks that could leak the access token to other systems.

The following flow-diagram describes this flow.



Detailed description of the presented flow:

1. **Step (A):** The user information is sent to the authorization server. Following parameters are sent by URL:

- *<response_type>*: Tells the authorization server to respond directly with an access token. It needs to be set to *<token>*.
- *<client_id>*: Identifies which client is requesting access for a user.
- *<redirect_uri>*: Strongly recommended and needs to match the settings for the *<client_id>*.
- *<scope>*: Optional, but currently not required for the SAP Commerce OCC Web Services, as there is a single *<customer>* scope.
- *<state>*: Optional, but recommended to mitigate XSS (Cross Site Scripting) attacks.

The following code sample uses the described parameters.

```

import java.net.URLEncoder

def client_id = 'client-side'
def redirect_uri = 'http://localhost:8080/oauth2_implicit_callback'

def scopes = [
    'customer'
]

def state = new Random(System.currentTimeMillis()).nextInt().toString()
  
```

```
session.state = state
```

```
redirect "http://localhost:9001/authorizationserver/oauth/authorize?client_id=${client_:
```

2. **Step (B):** Now, the user may log in and then be presented an authorization screen where they are asked to grant access to the specified scope.

3. **Step (C):** The authorization server redirects back to the client web application. The `<access_token>` is part of this redirect URI, but in the `<#hash>` fragment of the URI. This makes it invisible for some server-side code, like the J2EE, which does not allow access to the hash fragment (as it is only intended for the client). At this point, nothing can be done for the redirection request, but as it hits the web application again at the `<redirect_uri>` you need to make sure the request will work.

For the purpose of a demo client application, a controller endpoint is used that forwards directly to the HTML page. The remaining logic (which is, getting access to the `<access_token>` and verifying the state parameter) all happens in JavaScript. In the demo application we chose to extract the state from the user's session for which we need to route the redirect request through the controller.

```
request.setAttribute('state', session.state)
session.removeAttribute('state')
forward '/WEB-INF/pages/oauth2_implicit_callback.gtpl'
```

4. **Step (D):** Now, the user-agent (browser) follows the redirect and the client-side web application server responds with an HTML page that includes JavaScript to parse the hash fragment.

5. **Step (E):** The following sample shows the code of the returned HTML page.

```
<!doctype html>
<html>
  <head>
    <title>OAuth2 Implicit Callback Page</title>
  </head>
  <body>

    <div id="oauthParams" data-state=${request.getAttribute('state')}></div>

    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script src="/js/oauth.js"></script>

  </body>
</html>
```

Elements to note:

- **state:** The state that was into the user's session in the controller is extracted and passed on by the J2EE `HttpServletRequest` as an attribute. This state is passed on to a `<data-*>` field of the `<oauthParams>` HTML `<div>` element. Later, it can be extracted and verified with the state arrived in the redirection URI.
- **scripts:** In the sample, two script sources are included. One is the CDN-hosted jQuery, the second is our own oauth JavaScript.

6. **<Step (F)>** Now, at the user-agent end, JavaScript will access the `<#hash>` fragment to extract the access token and compare the state value. The JavaScript resource looks like in the following sample.

```
$(document).ready(function(){
  var oauthParams = {},
      queryString = location.hash.substring(1),
      regex = /([OAuth 2.0 (PSI 2 Update)^&=]+)=([OAuth 2.0 (PSI 2 Update)^&]*)/g,
      m,
      html='
  ';

  var state = $('#oauthParams').data('state');
```

```

while (m = regex.exec(queryString))
{
    oauthParams[decodeURIComponent(m[1])] = decodeURIComponent(m[2]);
}

console.log(oauthParams);

for (pos in oauthParams)
    html += pos + '=' + oauthParams[pos] + '
';

if (oauthParams.state == state)
    html += 'State matches, use the access_token!';
else
    html += 'State does NOT match! Issue!';

$('#oauthParams').html(html);
});

```

The OAuth 2.0 parameters are part of the hash fragment, that is, the part of a URL after the `<#>` sign.

7. **Step (G):** Finally, the `<access_token>` is passed on to the client.

Client Credentials Flow

The purpose of this flow is giving the client access to the resources it owns. It identifies the client and the OCC Web Services using the client credentials flow to restrict access to various calls.

The flow to obtain a client credentials token is very simple: you need to pass the `<client_id>` and `<client_secret>` parameters to the token endpoint. Keep in mind that in the production environment the token endpoint needs to use HTTPS. The following cURL sample provides an example.

```
curl -X POST -d "client_id=mobile_android&client_secret=secret&grant_type=client_credentials" http:
```

The authorization server responds with the access token if the credentials are valid.

```

{
  "access_token": "b4dc1410-6b15-4a7c-bd22-1abealf0fa3a",
  "token_type": "bearer",
  "expires_in": 42921
}

```

The token identifies, authenticates, and authorizes the client.

Refreshing an Expired Access Token

There is a need to refresh the issued `<access_token>`. Without refreshing these tokens, a `<client>` must remember user credentials, which is a less secure approach. Providing access tokens and refreshing the tokens means that the client only has to remember the tokens. The following code describes how to issue refresh tokens.

```

import com.google.appengine.api.urlfetch.*
import groovy.json.*
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'

if (!params.token)
{
    out << "No refresh token given..."
    return
}

```

```

}

def token = params.token

//exchange code for real oauth token
URL tokenURL = "http://localhost:9001/authorizationserver/oauth/token".toURL()
HTTPResponse res = tokenURL.post(deadline: 30, payload:"refresh_token=${token}&client_id=${client_id}")

out << res.text

```

It is a single request and response. In real life, the token endpoint would only be accessible by HTTPS. Parameters that must be sent to the /oauth/token endpoint to refresh a token are:

- *<client_id>* and *<client_secret>*: As the refresh request is sent from a server, keep the *<client_secret>* really secret. Both parameters need to be sent along.
- *<redirect_uri>*: Required, although not used here. The server responds with a JSON file with the token information directly. If you have a *<redirect_uri>* configured for the client, you need to pass this on for the refresh request.
- *<grant_type>*: It needs to be *<refresh_token>*, indicating the exchange of a refresh token for a new *<access_token>* and also a refresh token for the next time.
- *<refresh_token>*: For the client; the refresh token passed as the token request parameter and then into the refresh token request.

The server responds with a JSON response like in the following example.

```

{"access_token": "b70aeeb9-9095-4899-9480-2dbf432746ac", "token_type": "bearer", "refresh_token": "61f0c"}

```

Configuring OAuth Clients

Follow the steps to configure OAuth clients using the Backoffice Administration Cockpit or ImpEx.

Context

OAuth client is an application making protected resource requests on behalf of the resource owner and with its authorization. Each client application, such as mobile android client, mobile iOS client, which should have access to resources need to be registered in OAuth authorization server.

Adding OAuth Clients

Context

Follow the steps listed below to add a new OAuth client.

Procedure

1. Navigate to the OAuth Clients node ( **System** >  **OAuth** >  **OAuth Clients** >).

On the left you see a list of already created OAuth Clients. Here you can easily manage the clients.

2. In order to add a new OAuth client click the **OAuth Client Details** field.

The editor appears.

3. In the **Essential** section, provide *<Client ID>* and *<Client Secret >*(an encoded password).

Create New OAuth Client Details

Essential Basic Scope Token Validity

Client ID: ?

DemoClient

Client Secret: ?

.....

.....

Cancel Next Done

4. In the Basic section, provide the following values.

- **Authorities:** Authorities - List of authorities (roles) that are granted to the OAuth client such as ROLE_CLIENT, ROLE_TRUSTED_CLIENT
- **Authorized Grant Types:** Grant types available for client for example: refresh_token, password, authorization_code, client_credentials Grant types decide, which getting token flow can be used by this client. Grant types supported by the authorization server can be configured in <authorization-server> element.
- **Resource Ids :** The resource identifiers to which this client can be granted access.
- **Redirect Uri :** Redirect Uri allowed for the client in Authorization Code Flow and Implicit Flow

To enter a particular value, use the **plus** sign and confirm with **Add**.

Create New OAuth Client Details

Essential Basic Scope Token Validity

Authorities: ?

1 Values

+ ✎ ⌵

ROLE_CLIENT

Cancel

Add

☐ Add to top

Cancel Next

5. In the **Scope** section, provide the following values:

- **Scopes:** List of scopes to which the client is limited.
- **Auto Approve Scopes:** Scopes the Client does not need the User approval for.

Create New OAuth Client Details

Essential Basic Scope Token Validity

Scopes: ?

0 Values

+ basic Cancel Add ☐ Add to top

Back Cancel Next Done

6. In the **Token Validity** section, provide the following values:

- **Access Token Validity Seconds:** The access token validity period in seconds.
- **Refresh Token Validity Seconds:** The refresh token validity period in seconds.

Create New OAuth Client Details

Essential Basic Scope Token Validity

Access Token validity time: ?




2000

Refresh Token validity time: ?

3000

Back Cancel Done

7. A new client is added to the list.

4 (OAuth Client Details)		
<div>   OAuth Client Details  </div>		
Client ID	Authorities	Client Grant Types
demoClient	[ROLE_CLIENT]	[client_credentials]
trusted_client	[ROLE_TRUSTED_CLIENT]	[refresh_token, password, client_credentials, authorization_code]
mobile_android	[ROLE_CLIENT]	[refresh_token, password, client_credentials, authorization_code]
client-side	[ROLE_CLIENT]	[implicit, client_credentials]

Editing OAuth Clients

Context

Follow the steps listed below to edit an already existing OAuth client.

Procedure

1. Navigate to **System > OAuth > OAuth Clients**.

On the left you see a list of already created OAuth Clients. Here you can easily manage the clients.

2. Select the client you want to edit.
3. Click **Save** after selected values.

Removing OAuth Clients

Context

Follow the steps listed below to remove an already existing OAuth client.

Procedure

1. Navigate to **System > OAuth > OAuth Clients**.

On the left you see a list of already created OAuth Clients. Here you can easily manage the clients.

2. Select the client you want to remove.
3. Click the **bin** icon to remove the row. A dialog is displayed for you to confirm the action.

Defining OAuth Clients in an ImpEx File

OAuth clients data are stored in the database using the `OAuthClientDetailsModel`, so it can be also created using an ImpEx file.

Procedure

1. Define the OAuth client in the `projectdata.impex` file.

```
INSERT_UPDATE OAuthClientDetails;clientId[unique=true]      ;resourceIds      ;scope      ;a
                ;client-side      ;hybris      ;basic      ;i
                ;mobile_android      ;hybris      ;basic      ;a
```

2. **i Note**

When defining the clients remember to assign either the `ROLE_CLIENT` or `ROLE_TRUSTED_CLIENT` to them, because these roles allow client access to `ycommercewebservices`. Be careful with the `ROLE_TRUSTED_CLIENT` because it has got specific, extended rights.

If you want to add the `ROLE_TRUSTED_CLIENT`, you will also need to define it in the `projectdata.impex` file.

```
INSERT_UPDATE OAuthClientDetails;clientId[unique=true]      ;resourceIds      ;scope      ;a
                ;trusted_client      ;hybris      ;extended      ;a
```

Platform as an OpenId Connect Identity Provider

With OpenId Connect, software developers don't have to bother with managing or storing passwords to authenticate end users. You can use this technology to authenticate client application users to access external systems by using Platform as the identity provider.

To use this function, you have to configure Platform as an ID provider, and register it in the external system.

The example below uses the OpenID Connect Implicit Grant flow. First, your application requests Platform for an ID token. Platform returns the requested ID token, and the application sends it to the external system. In return, the external system sends back the access token that allows you to access specific resources from the external system.

Configuring Platform as an ID Provider

OpenId Connect is an identity layer based on the OAuth 2.0 authorization framework. For that reason, you have to configure the OAuth Client role. For details, see [OAuth 2.0](#).

Here is an example of how you can configure the client role. The client used in all examples has id `client-side`.

exampleOpenIDClientDetails configuration impex

```
INSERT_UPDATE OpenIDClientDetails;clientId[unique=true]      ;resourceIds      ;scope      ;authori
                ;client-side      ;hybris      ;basic,email,profile,c
```

Here is an example of how you can configure the external system scopes.

```
insert_update OpenIDExternalScopes;code[unique=true];clientDetailsId(clientId)[allownull=true,force
                ;editor      ;client-side
                ;reader      ;client-side
                ;admin      ;client-side
```

KeyStore

To use the OpenID feature, your SAP Commerce instance must have a Java KeyStore repository containing both public and private keys. Platform is shipped with a demo keystore but it is only for development use. You can generate a new keystore using the Java keytool. We recommend using the RSA algorithm for encryption.

For example, this command generates a 2048-bit RSA keystore.

```
> keytool -genkey -keyalg RSA -alias alias -keystore keystore.jks -keysize 2048
```

The password is stored in the properties.

Properties

Property	Definition
oauth2.idTokenValiditySeconds=43200	Id_token validity in seconds.
oauth2.client-side.kid=test1	The id (alias) of the public/private key used by a particular client for signing id tokens.
oauth2.client-side.keystore.location=/security/keystore.jks	Location of the KeyStore for a particular client id.
oauth2.client-side.keystore.password=nimda123	Password for the given KeyStore.
oauth2.algorithm=RS256	The algorithm used in the KeyStore; the same algorithm should be used for signing id tokens; the default implementation is RS256 and shouldn't be changed without having an additional implementation of another algorithm.

Configuring the ID Provider in an External System

You have to configure Platform as an ID provider in your external system. For that reason, refer to the documentation of your external system.

Implicit Grant Flow Requests

First, request an id_token from the external provider (Platform).

```
GET https://EC_IDP_URL/authorize/?
  response_type=id_token token
  &client_id=EXTERNAL_CLIENT_ID
  &redirect_uri=REDIRECT_URI_TO_WEB_PAGE
  &scope=SCOPE
  &nonce=NONCE
  &state=WEB_STATE
```

It should redirect you to the external page and finally redirect to the original page. The id_token should be in the URL of the web page.

Example URL:

```
GET https://localhost:9002/authorizationserver/oauth/authorize?response_type=token id_token&client_
```

The redirected URL contains the id_token value.

`http://MY_APPLICATION/ #access_token=cfc4af6d-b38b-4152-a923-9af8de6c7f33%26token_type=bearer%26st:`

The encoded `id_token` is:

`eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6InRlc3QxIn0.eyJzdWIiOiJlZGl0b3IiLCJzY29wZSI6WyJvcGVuaWw`

After decoding the `id_token`, you get:

```
HEADER:ALGORITHM & TOKEN TYPE
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "test1"
}
PAYLOAD:DATA
{
  "sub": "editor",
  "scope": [
    "openid",
    "hybris.product_read_unpublished"
  ],
  "iss": "ec",
  "state": "3",
  "exp": 1501849757,
  "nonce": "5",
  "iat": 1501846157
}
```

Now exchange the `id_token` for the `access_token`:

```
GET https://external_system.io/oauth2/v1/authorize/?
&client_id=EXTERNAL_SYSTEM_CLIENT_ID
&hybris_id_provider=ID_PROVIDER_REGISTERED_IN_EXTERNAL_SYSTEM
&id_token_hint=ID_TOKEN_FROM_EXTERNAL_PROVIDER
&nonce=NONCE
&redirect_uri=REDIRECT_URI_TO_WEB_PAGE
&response_type=token
&state=WEB_STATE
```

It should first redirect you to the external system OAuth, then to the external provider (Platform), and finally to the original page. The `access_token` should be in the URL of the web page.

Exchange the `id_token` for the `access_token`.

```
GET https://external_system.io/oauth2/v1/authorize/?client_id=ZwXU5ktg3TEYdZnDjrf45CZjyc09KYd4&id_1
```

The redirected URL contains this `access_token` value.

`http://MY_APPLICATION/#token_type=Bearer&access_token=022-de34d0e4-0557-4c18-b8ef-7c8544e8d6b9&exp:`

OpenID Connect Discovery

OpenID Connect Discovery enables clients to verify the identity of the end user based on the authentication performed by SAP Commerce.

OpenID Connect Discovery is based on two endpoints provided by SAP Commerce.

The Discovery endpoint is available at `https://{server}:9002/authorizationserver/.well-known/openid-configuration`. The following is an example response from the Discovery endpoint:

```
{
  "issuer" : "https://myserver.sap.corp:9002/authorizationserver",
  "authorization_endpoint" : "https://myserver.sap.corp:9002/authorizationserver/oa
  "token_endpoint" : "https://myserver.sap.corp:9002/authorizationserver/oauth/tok
  "jwks_uri" : "https://myserver.sap.corp:9002/authorizationserver/.well-known/jwks
  "token_endpoint_auth_methods_supported" : [ "client_secret_post", "client_secret_
  "subject_types_supported" : [ "public" ],
  "response_types_supported" : [ "code", "code id_token", "id_token", "token id_to
  "scopes_supported" : [ "openid", "email", "groups" ],
  "id_token_signing_alg_values_supported" : [ "RS256" ]
}
```

The JWKS endpoint is available at `https://{server}:9002/authorizationserver/.well-known/jwks.json`. The following is an example response from the JWKS endpoint:

```
{
  "keys" : [ {
    "kty" : "RSA",
    "use" : "sig",
    "kid" : "kyma",
    "alg" : "RS256",
    "n" : "ALyXGDrQ3QZcugzjcJDPsVsK70RTNHxnJIEt0ANM_tPxzf5wd7lFaZ7xjbl2rZaNICxT7MqfY-euvX1PAsXkby8l
    "e" : "AQAB"
  } ]
}
```

The following is an example configuration:

Property	Example Value	Info
<code>oauth2.kyma.algorithm</code>	RS256	Defaults to RS256
<code>oauth2.kyma.responseTypes</code>	code,code id_token,id_token,token id_token	Defaults to code,code id_token,id_token,token id_token
<code>oauth2.kyma.kid</code>	kyma	kid stands for key id (alias) for the kyma keytore
<code>oauth2.kyma.keystore.location</code>	/security/keystore.jks	Kyma keystore location
<code>oauth2.kyma.keystore.password</code>	nimda123	
<code>oauth2.kyma.public.address</code>	http://www.example.com	Replaces <code>myserver.sap.corp</code> from the example Discovery endpoint response with chosen address

To enable configuration per client, you can access endpoints per configuration/key:


- `https://{server}:9002/authorizationserver/.well-known/openid-configuration?client_id={id}`
- `https://{server}:9002/authorizationserver/.well-known/ jwks.json ?client_id={id}`

For example:

- `https://{server}:9002/authorizationserver/.well-known/openid-configuration?client_id=kyma`
- `https://{server}:9002/authorizationserver/.well-known/ jwks.json ?client_id=kyma`

Token Revocation

You can use the `/revoke` endpoint in the `oauth2` extension to revoke issued tokens.

This endpoint conforms with the RFC-7009 requirements described at <https://tools.ietf.org/html/rfc7009> .

You don't need any additional authorization to revoke tokens - you are authorized with a token you are about to revoke.

The example tokens used in this topic are:

- access token `d7689e7c-957e-46ea-949b-c39afb1c9935`
- refresh token `00d275c4-0daa-4876-8a3b-4aa731f3e13a`

To revoke the access token, call the `/revoke` endpoint with a POST method, and pass the access token in the `token` parameter:

```
-X POST -H https://localhost:9002/authorizationserver/oauth/revoke -d "token=d7689e7c-957e-46ea-949b-c39afb1c9935"
```

The response you get is:

```
HTTP/1.1 200
Content-Length: 0
```

You can revoke an access or a refresh token by passing it with a hint indicating the token type. To do it, add to the request the `token_type_hint` parameter set to `access_token` or `refresh_token` respectively:

```
curl -X POST -H https://localhost:9002/authorizationserver/oauth/revoke -d "token=00d275c4-0daa-4876-8a3b-4aa731f3e13a&token_type_hint=refresh_token"
```

The response you get is:

```
HTTP/1.1 200
Content-Length: 0
```

A request with `token_type_hint` revokes a token of the specified type. If you pass, for example, an access token and set the hint to `refresh_token`, then the token doesn't get revoked.

Setting `token_type_hint` is optional. If you don't specify it, then Platform first tries to revoke your token as an access token. If your token isn't an access token, Platform tries to revoke it as a refresh token.

If you specify an unsupported token type, then you get the following error message:

```
"unsupported_token_type: The authorization server does not support the revocation of the presented token type. That is, the client tried to revoke an access token on a server not supporting this feature"
```

If you specify an invalid token in the `token` parameter, you don't receive any error message. The request returns the 200 OK status response.

Configuring customTokenGranter for External IdPs

Enhance user experience by enabling additional login methods through external IdPs.

Context

provides an alias `customTokenGranter` bean as part of the Spring Security OAuth Authorization Server. The baseline implementation, `DefaultCustomTokenGranter`, accepts the “custom” grant type and, in the default configuration, throws an exception stating that there is no implementation. Replace it with your own implementation of the `TokenGranter` interface to integrate external IdPs with SAP Commerce Cloud, composable storefront or any of your custom storefronts. By using external IdPs, you can expand the list of available login methods, allowing your customers to log into your application through already authenticated accounts without the need of going through the process of creating new ones from scratch. For more information on `TokenGranter`, see [TokenGranter](#) ➔ in Spring documentation.

i Note

You can only use one `customTokenGranter` per instance. To integrate multiple IdPs with your storefront, ensure that your custom implementation of `customTokenGranter` supports such configuration.

Procedure

1. Create your custom implementation of `customTokenGranter` to override the existing configuration of the `TokenGranter` interface:

```
<alias name="specificCustomTokenGranter" alias="customTokenGranter" />
<bean id="specificCustomTokenGranter" ... />
```

2. In your custom implementation of `customTokenGranter`, define what information the client needs to pass into the token request. This includes custom or specific fields needed by a specific IdP.
3. Ensure that the external IdP client passes the required information that is defined in `customTokenGranter` as part of the token request into the `oauth2` extension. The IdP client needs to use the “custom” grant type that routes its requests to the `customTokenGranter` implementation.
4. Perform any necessary post-validation processing, such as logging, auditing, or updating user sessions, before returning the OAuth token to the client.
5. Once the OAuth token is generated and returned to the client, it can be used for subsequent requests to access protected resources. Ensure that your OAuth server validates these tokens for each request to ensure secure access to resources.