RESO Web API Security v1.0.3

Section 1 - RESO Security Requirement 4
Section 2 - Intro to OpenID Connect 5
2.1.1 Terminology 7
2.1 - OpenID Connect Relying Party 7
2.1.1 OpenID Connect Relying Party Libraries 8
2.1.2 Discover Endpoints 8
2.1.3 Authorization Code Flow 9
2.1.3.1 Step 1 - Authorize 10
2.1.3.2 Step 2 - Callback 11
2.1.3.3 Step 3 - DATA! 14
2.1.3.4 Step 4 - Refresh 15
2.1.4 Implicit Flow 17
2.1.5 Hybrid Flow 18
2.2 - OpenID Connect RETS Server Provider 19
2.2.1 OpenID Connect Provider Libraries 20
2.2.2 Discovery service 20
2.2.3 Register New Relying Parties 21
2.2.4 Authorize Endpoint 22
2.2.5 Token Endpoint 23
2.2.6 UserInfo Endpoint 24
2.2.7 Verify Access Tokens 25
2.2.8 Refreshing an Access Token 25
2.2.8.1 An expired access token returns HTTP 401 26
2.2.8.2 Relying Party makes a request to the RETS Server Provider's token endpoint 26
2.2.8.3 Relying Party saves the access and refresh tokens 26
2.2.9 Implicit Flow 26
2.2.10 Hybrid Flow 27
2.2.11 Extra Security Measures 29
Section 3 - FAQ 29
Section 4 - Authors 30
Section 5 - Revision List 31
Section 6 - Appendices 32
6.1 Use Case Diagrams 33
6.1.1 SP (Service Provider) to SP/IdP (Identity Provider) 33
6.1.2 SP to IdP to SP Typical three-way authorization 34
6.1.3 SP to SP/IdP Transparent three-way authorization 35
6.1.4 SP to SP/IdP Transparent, recurring "on behalf of" authorization 37
6.1.5 2-legged Client-Server Auth 38
6.1.6 4-legged Federated Identities 38
6.2 Resources and Links 43
6.2.1 Help Guides and Introductions 43
6.2.2 Library Demos and Examples 43
6.2.3 Identity-as-a-Service Providers 44

RESO Web API Security v1.0.3

Section 1 - RESO Security Requirement

Section 2 - Intro to OpenID Connect

- 2.1.1 Terminology
- 2.1 - OpenID Connect Relying Party
  - 2.1.1 OpenID Connect Relying Party Libraries
  - 2.1.2 Discover Endpoints
  - 2.1.3 Authorization Code Flow
    - 2.1.3.1 Step 1 - Authorize
    - 2.1.3.2 Step 2 - Callback
    - 2.1.3.3 Step 3 - DATA!
    - 2.1.3.4 Step 4 - Refresh
  - 2.1.4 Implicit Flow
  - 2.1.5 Hybrid Flow
- 2.2 - OpenID Connect RETS Server Provider
  - 2.2.1 OpenID Connect Provider Libraries
  - 2.2.2 Discovery service
  - 2.2.3 Register New Relying Parties
  - 2.2.4 Authorize Endpoint
  - 2.2.5 Token Endpoint
  - 2.2.6 UserInfo Endpoint
  - 2.2.7 Verify Access Tokens
  - 2.2.8 Refreshing an Access Token
    - 2.2.8.1 An expired access token returns HTTP 401
    - 2.2.8.2 Relying Party makes a request to the RETS Server Provider's token endpoint
    - 2.2.8.3 Relying Party saves the access and refresh tokens
  - 2.2.9 Implicit Flow
  - 2.2.10 Hybrid Flow
  - 2.2.11 Extra Security Measures

Section 3 - FAQ

Section 4 - Authors

Section 5 - Revision List

Section 6 - Appendices

- 6.1 Use Case Diagrams
  - 6.1.1 SP (Service Provider) to SP/IdP (Identity Provider)
  - 6.1.2 SP to IdP to SP Typical three-way authorization
  - 6.1.3 SP to SP/IdP Transparent three-way authorization
  - 6.1.4 SP to SP/IdP Transparent, recurring "on behalf of" authorization
  - 6.1.5 2-legged Client-Server Auth
  - 6.1.6 4-legged Federated Identities
- 6.2 Resources and Links
  - 6.2.1 Help Guides and Introductions
  - 6.2.2 Library Demos and Examples
  - 6.2.3 Identity-as-a-Service Providers

Section 1 - RESO Security Requirement

A compliant RESO Web API Server v1.0.3 **MUST** support **token based authentication** with an HTTP `Authorization` header of "Bearer <token>" where the format of <token> is defined by the compliant RESO Web API Server v1.0.3.

A compliant RESO Web API Server v1.0.3 **MUST** support Section 2.1 of RFC 6750 OAuth2 Bearer Token Usage as the authentication method for server-to-server based communication.

A compliant RESO Web API Server v1.0.3 **MUST** support Authorization Code Flow of the OpenID Connect Protocol Suite as the authentication method for user based communication.

A compliant RESO Web API Server v1.0.3 **MAY** use any of the additional OpenID Connect Protocol Suite of standard specifications.

For data warehousing and replicating data between servers, a compliant RESO Web API Server v1.0.3 **MAY** use the OAuth2 Client Credentials Grant specified in Section 4.4 of RFC 6749 The OAuth 2.0 Authorization Framework.

Recommendations

If a RESO Web API Server intends to use a real-time 3-legged "on behalf of" architecture, the official RESO recommended standard is the OpenID Connect Protocol Suite. The RESO Web API Security document provides a help guide for implementing OpenID Connect.

If a RESO Web API Server is used for replicating data between servers, the recommended approach is to manually distribute access tokens to the peer servers as described in Use Case 6.1.1.

Section 2 - Intro to OpenID Connect

TL;DR OpenID Connect:

OpenID Connect **is** OAuth2, wrapped up in a standardized, worldwide protocol. It adds a few new features in the interest of security, identity, and mobile apps. It's backwards compatible with the RESO Web API Security v1.0.1 standard, and provides an easy transition path from plain OAuth2.

> **Disclaimer**
> The RESO Web API Security v1.0.3 document **does not alter the OpenID Connect standard**.
>
> This document is just a guide to assist the learning process. Check out the Resources section for more excellent articles on OpenID Connect.

> **Words Are Hard**
> Read about them in 2.1.1 Terminology

OpenID Connect defines three authentication flows. A RETS Provider does not need to implement all three. At the minimum, either Implicit or Authorization Code could be implemented based on your architecture.

Implicit

The Implicit flow allows a client web browser or native mobile app to talk directly to a RETS Provider.

Authorization Code

The Authorization Code flow behaves just like the typical OAuth2 3-legged authentication.

Hybrid

The Hybrid flow combines both Implicit and Authorization Code. This can be used to solve many use cases, but the most relevant is for use with a native mobile app. With a single log in, both a native mobile app and a Relying Party website can acquire access to a RETS Provider's API.

Discovery Service

OpenID Connect's interoperability power comes from the Discovery metadata document. This is a simple (static) JSON response that advertises the features and URIs of a RETS Server Provider. OpenID Connect client libraries automatically pick up the metadata and know how to behave with a given RETS Server Provider. This removes the need to document endpoints as in the previous API Security v1.0.1. It also gives the RETS Server Provider the freedom to choose features without impacting interoperability.

The Discovery service is optional in the OpenID Connect specifications. However, it's very simple to implement, and gives OpenID Connect its powerful interoperability.

ID Tokens

OpenID Connect adds an ID Token on top of OAuth2's access and refresh tokens. ID Tokens are JSON Web Tokens, and represent an identity with additional profile information (Claims) about a Member. This is the only added requirement OpenID Connect has on top of the OAuth2 RFC.

UserInfo Endpoint

OpenID Connect adds only one API resource with database-backed content. It is protected by an access token, and returns a JSON structure of Claims (profile information) about the member. It's very similar to the Web API Member resource, except it only returns data about the current user session. The UserInfo endpoint is optional, and the primary use is to transport large amounts of Claim data that shouldn't fit in an ID Token.

Continue with the overview: Which role are you writing?

- Section 1.1 - OpenID Connect Relying Party
- Section 1.2 - OpenID Connect RETS Server Provider

2.1.1 Terminology

OpenID Connect Vocabulary

A few of the terms in this document are changing to reflect the terms in the OpenID Connect specifications. Nothing radical is changing though.

Client Browser – The Member's web browser or mobile native application.

RETS Relying Party – Previously called the API Consumer, or Service Provider. This is the server-side application that consumes MLS Data and acts as a middle-man between the Client Browser and the RETS Server Provider. The OpenID Connect specifications usually abbreviate this as **RP**.

RETS Server Provider – Previously called just RETS Server in Security v1.0.1. This is the Identity Provider for the Site/MLS. Since OpenID Connect calls the server a Provider, we'll add this on to avoid confusion. The OpenID Connect specifications usually abbreviate this as **OP**.

RETS API Server – The Site/MLS's server running the RESO Web API with OData V4 resources. With OpenID Connect federated identities, the API Server might live on a separate domain from the OpenID Connect Server Provider.

Implicit Flow – A Client Browser talks directly to the RETS Server Provider with this method

Authorization Code Flow – This is the standard 3-legged OAuth2 style authentication.

Hybrid Flow – A combination of both Implicit and Hybrid flows.

ID Token – A JSON Web Token that represents a Member's identity

UserInfo Endpoint – An API endpoint on the RETS Server Provider that returns Claims about the current user

Discovery service – An API endpoint on the RETS Server Provider that returns a static JSON structure which describes the Provider's supported features and URI endpoints.

Claim – Identifying profile information about a user. (IE: Name, Email, Phone, Address, etc)

2.1 - OpenID Connect Relying Party

The Relying Party accepts client browsers and uses an access token to retrieve data from the RETS API Server. This access token is an obfuscated transient string, and represents the API access grant for an MLS Member or VOW consumer. A Relying Party is typically a standard MVC web application and is strictly a server-side middle-man between a Client Browser and the RETS Server Provider and RETS API Server. This middle-man approach adds extra security features when compared to the older RETS 1.x client-server model.

A Relying Party must first apply for a registration process from the Site/MLS. A RETS Server Provider may allow automated registration, and will be shown in the Provider's Discovery metadata. The Site/MLS will provide a **`client_id`** and **`client_secret`**. The `client_id` is registered with a **`redirect_uri`**, which is a URL that points back to the Relying Party's callback controller.

The general algorithm a Relying Party should implement is:

Step 1: Discover the RETS Server Provider's metadata

Step 2: Accept an unauthenticated Client, and redirect to the RETS Server Provider's **`authorize`** endpoint

Step 3: Accept a Client at the callback URL (registered `redirect_uri`). The RETS Server Provider appends a **`code`** parameter to this call. Exchange the `code` for access and refresh tokens at the RETS Server Provider's **`token`** endpoint

Step 4: Use the access token in the `Authorization` HTTP header to request data from the API

Step 5: If an access token expires, use the refresh token at the RETS Server Provider's `grant` endpoint to attain a fresh one

This is known as the Authorization Code flow, and is the most common. OpenID Connect also defines an Implicit flow for use with mobile native applications that talk directly to the RETS API Server. The third flow is called the Hybrid flow, and is a combination of both Implicit and Authorization Code. This allows both a Relying Party and a native application access to the RETS API Server with a single authorization.

A few security guidelines:

1. A Relying Party **MUST NEVER** give out access tokens, refresh tokens, or client_secrets. Treat these like a password!
2. The Relying Party **MUST** use TLS for the callback URL
3. Do not mix up client sessions with different access tokens. They are a 1:1 identity relationship. Hint: The OpenID Connect ID Token will contain a cryptographic signature of the access token and authorization code. The client library should verify the correct pairing before making a request to the RETS API Server.

Dig Deeper with...

- OpenID Connect Discovery
- OpenID Connect Dynamic Client Registration

- 2.1.1 OpenID Connect Relying Party Libraries
- 2.1.2 Discover Endpoints
- 2.1.3 Authorization Code Flow
- 2.1.4 Implicit Flow
- 2.1.5 Hybrid Flow

2.1.1 OpenID Connect Relying Party Libraries

First, check out the official certified libraries page published by the OpenID Foundation:

http://openid.net/developers/libraries/

If your language or platform is not listed there, here's a few extra libraries we've found:

JavaScript

- Making a Javascript OpenID Connect Client in 4 steps
  - Source Library here
- Javascript Cookbook for OpenID Connect Public Client
- OpenID Connect Button (super easy!)
- Account Chooser

C# / ASP.NET

- https://github.com/Azure-Samples/active-directory-dotnet-webapp-openidconnect
- https://github.com/Azure-Samples/active-directory-dotnet-webapp-multitenant-openidconnect
- https://github.com/IdentityModel/IdentityModel

2.1.2 Discover Endpoints

The Discovery service gives OpenID Connect its true interoperability power. All OpenID Connect certified Relying Party libraries should support this without any effort from the developer.

In case you're writing a client from scratch, reference the OpenID Connect Discovery spec for the full details. It's very simple:

Step 1

Start with the RETS Server Provider's base URL, using the https scheme. Examples:

- https://sparkplatform.com
- https://accounts.google.com
- https://login.salesforce.com

Step 2

Append to the base URL:

```
/.well-known/openid-configuration
```

Which results in:

```
https://sparkplatform.com/.well-known/openid-configuration
https://accounts.google.com/.well-known/openid-configuration
```

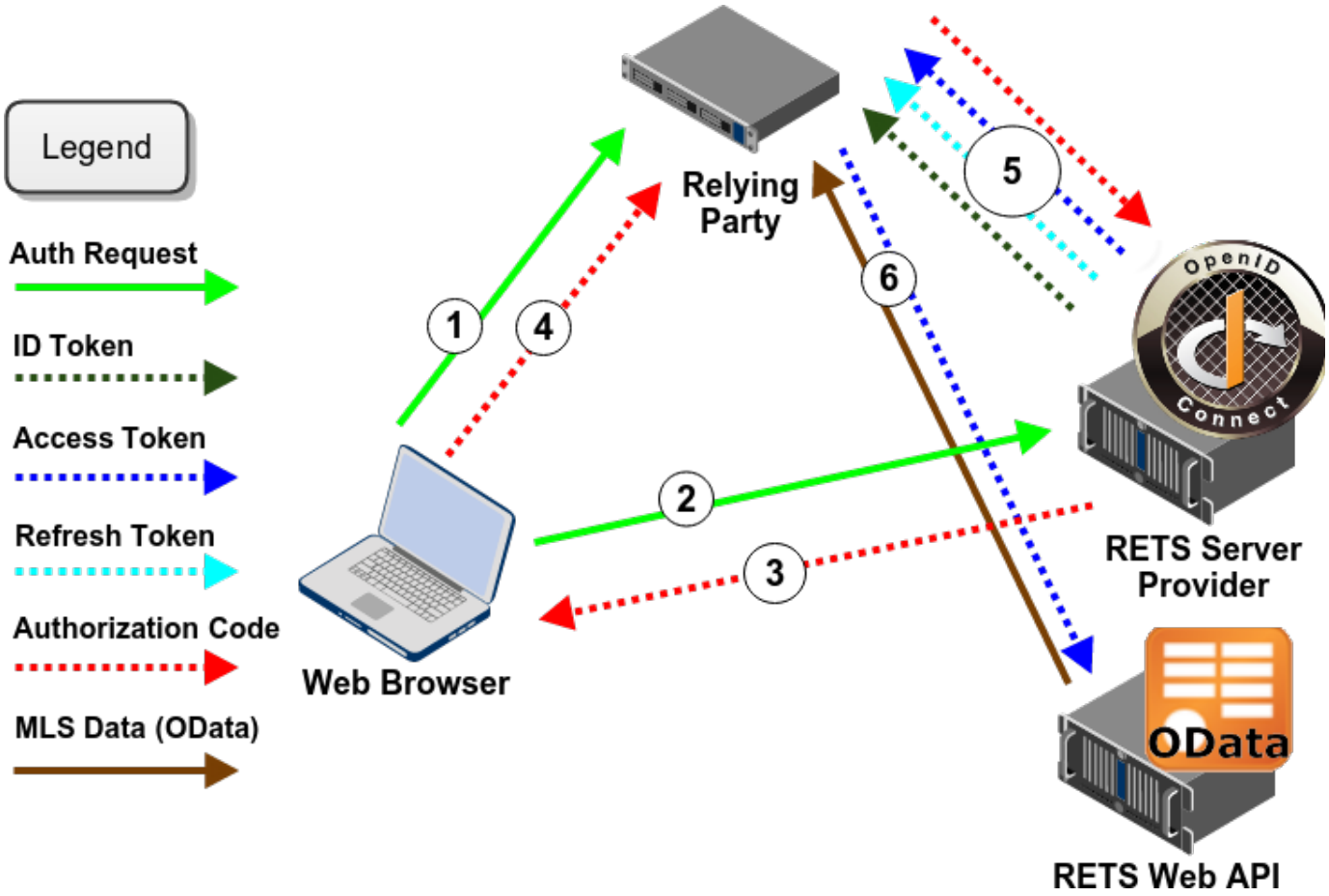
Step 3

Retrieve the JSON from that URL and save the data for future use.

```
{
 "issuer": "https://accounts.google.com",
 "authorization_endpoint": "https://accounts.google.com/o/oauth2/v2/auth",
 "token_endpoint": "https://www.googleapis.com/oauth2/v4/token",
 "userinfo_endpoint": "https://www.googleapis.com/oauth2/v3/userinfo",
 "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
  ...
```

In the previous Web API Security v1.0.1, we asked providers to write a public document with these locations. OpenID Connect provides an automated means of communicating this information to client libraries.

2.1.3 Authorization Code Flow



The Authorization Code flow is a standard OAuth2, 3-legged authentication scheme. This is the exact same as the previous Web API Security v1.0.1, with the addition of an ID Token added at the token exchange. In most cases the OpenID Connect RETS Server Provider will live on the same domain as the OData RETS Web API, but it's possible to separate these to different domains.

Refer to the diagram above for a review of how Authorization Code works:

1. The Client web browser requests the Relying Party site
    a. MLS Member chooses a login provider
    b. The Relying Party's OpenID Connect client discovers the Provider's endpoints (cached)
2. The Relying Party redirects the Client web browser to the RETS Server Provider's authorization endpoint with the client_id and redirect_uri parameters
    a. MLS Member provides a username/password
    b. MLS Member authorizes the Relying Party access to MLS data
3. RETS Server Provider responds with an Authorization Code to the Client web browser
4. The Client web browser redirects to the Relying Party with the Authorization Code
5. The Relying Party uses the Authorization Code to request an ID Token, Access Token, and/or Refresh Token
6. The Relying Party uses the Access Token to retrieve:
    a. Additional Claims (profile info) about the Member from the Provider's UserInfo endpoint
    b. OData API requests against the RETS Web API
    c. Uses the Refresh Token to request another Access Token if expired

- 2.1.3.1 Step 1 - Authorize
- 2.1.3.2 Step 2 - Callback
- 2.1.3.3 Step 3 - DATA!
- 2.1.3.4 Step 4 - Refresh

2.1.3.1 Step 1 - Authorize

Accept an unauthenticated client request:

Client Request

```
GET / HTTP/1.1
Host: app.example.com
```

Respond with a redirect to the RETS Server's `authorize` endpoint. If the Provider supports the Discovery service, this URL will be listed in the JSON as `authorize_endpoint`. Provide the `client_id` and `redirect_uri` parameters associated with the API Consumer. A `state` para meter is an extra security measure, and is a unique session ID to assist in preventing cross-site forgery attacks.

Response

```
HTTP/1.1 302 Found
Location:
https://rets.example.com/authorize?client_id=7d1wp67gl1oo8wsc8ks4csgsk
          &scope=openid
          &response_type=code
          &state=o5n9ki8kpil86vl9j11uujbn41
          &redirect_uri=https://app.example.com/callback.php
```

(Note that this looks exactly like OAuth2, with an added scope=openid)

PHP Example

A quick snippet in everyone's favorite language

index.php

```
<?php
$client_id = "7d1wp67gl1oo8wsc8ks4csgsk";
$callback = "https://app.example.com/callback.php";
if ( session_id() === "" && $_COOKIE[session_name()] == NULL )
{
  session_start();
  header("Location: https://rets.example.com/authorize?"
         . "client_id=$client_id"
         . "&scope=openid"
         . "&response_type=code"
         . "&state=" . session_id()
         . "&redirect_uri=$redirect_uri");
}
?>
```

2.1.3.2 Step 2 - Callback

Accept a client request that has been redirected from the RETS Server Provider. The MLS Member has logged in, and is being redirected back to the Relying Party.

Client Request

```
GET
/callback.php?code=5i46ka0uur7soktiyca6lcczt?state=o5n9ki8kpil86vl9j11uujb
n41 HTTP/1.1
Host: app.example.com
Referer: https://rets.example.com/authorize
Cookie: PHPSESSID=o5n9ki8kpil86vl9j11uujbn41;     (Set from the example
index.php)
```

Verify that the `state` parameter is the same as the current session ID to prevent cross-site forgery attacks.

Before responding to this client, open up a new server-side HTTP request to the RETS Server's token exchange service. Provide the **`client_id`**, **`client_secret`**, **`redirect_uri`**, and authorization **`code`**.

RETS Server Provider Request

```
POST /token HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"code":"5i46ka0uur7soktiyca6lcczt",
 "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
 "client_secret":"6pphytzx8qklfa2wi23wgiyil",
 "redirect_uri":"https://app.example.com/callback.php",
 "grant_type":"authorization_code"}
```

RETS Server Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{"access_token":"2w9wc3b8565ajpj4i9v68ivlv",
 "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
 "id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6I...",
 "expires_in":3600}
```

- Save the access and refresh tokens in a protected storage space, referenced by the client's session ID. (`o5n9ki8kpil86vl9j11uujbn41`)
- Optionally, save the `expires_in` timestamp to know in advance when a refresh will be needed.
- If an ID Token is returned, validate it according to OpenID Connect Core Section 3.1.3.7.
- You may use the subject of the ID Token in place of a unique identifier for the MLS Member
- The ID Token string itself may be used as a session cookie for the Member's browser
- Use the access token against the Provider's UserInfo endpoint to get a Member's profile information for display purposes. (Name, Email, etc)

Respond to the client with a redirect to the Relying Party's "logged in" landing location. TLS is not required after this step!

Client Response

```
HTTP/1.1 302 Found
Location: http://app.example.com/dashboard.php
```

PHP Example

callback.php

```
<?php
session_start();
$code = $_REQUEST["code"];
$state = $_REQUEST["state"];
if ( $state != session_id() )
{
   # Cross site forgery detection!
}
$ch = curl_init(); curl_setopt($ch, CURLOPT_URL,
"https://rets.example.com/grant");
curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-Type:
application/json'));
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode(array(
  "code":"5i46ka0uur7soktiyca6lcczt",
  "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
  "client_secret":"6pphytzx8qklfa2wi23wgiyil",
  "grant_type":"authorization_code"
)));

$response = curl_exec($ch);
curl_close($ch);

$response = json_decode($response);
$access_token = $response["access_token"];
$refresh_token = $response["refresh_token"];

# Calculate the timestamp a refresh will be needed at
$expires_at = strftime("%Y-%m-%d %H:%M:%S", time() +
$response["expires_in"]);

# Insert something useful into a database
$sql = "insert into keys (session_id, access_token, refresh_token,
expires_at) "
  . " values ('" . session_id() . "', '$access_token', '$refresh_token',
'$expires_at')";

# Redirect to our landing page
header("Location: http://app.example.com/dashboard.php"
?>
```

2.1.3.3 Step 3 - DATA!

The access token can now be used in the **`Authorization`** HTTP header to request data from the RETS Server API on behalf of the MLS Member. Make sure to follow these security guidelines:

1. A Relying Party **MUST NEVER** give out access tokens, refresh tokens, or client_secrets. Treat these like a password!
2. Do not mix up client sessions with different access tokens. Use the `at_hash` Claim in the ID Token to verify you have the correct access token.

Client Request

```
GET /dashboard.php HTTP/1.1
Referer: https://app.example.com/callback.php
Cookie: PHPSESSID=o5n9ki8kpil86vl9j11uujbn41;
```

The request to the RETS Server API might look something like this:

RETS API Request

```
GET /RESO/OData/Properties.svc/Properties('ListingId3') HTTP/1.1
Host: rets.example.com
Authorization: Bearer 2w9wc3b8565ajpj4i9v68ivlv
```

PHP Example

dashboard.php

```
session_start();
# Retrieve this client's access token.  expires_at condition optional,
# the RETS Server will tell us when an access token is expired
$sql = "select access_token from keys where session_id='" . session_id() .
"' and now() < expires_at";

curl_setopt($ch, CURLOPT_URL,
"https://rets.example.com/RESO/OData/Properties.svc/Properties('ListingId3
')");
curl_setopt($ch, CURLOPT_HTTPHEADER, array("Authorization: Bearer
$access_token"));
$response = curl_exec($ch);
$status = curl_getinfo($ch, CURLINFO_HTTP_CODE);
if ( $status == 401 )
{
   // See Section 1.1.3.4 Step 4 - Refresh
}

# present $response to the client
```

2.1.3.4 Step 4 - Refresh

If a RETS API Server responds from an API request with an HTTP 401 response, the access token is invalid and must be refreshed.

RETS API Response

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm='RETS Server', error='expired_token'
Content-Type: application/json

{ "message": "Access token has expired" }
```

The Relying Party uses the `client_id`, `client_secret`, and `refresh_token` to retrieve a fresh set of access and refresh tokens for the MLS Member.

RETS Server Request

```
POST /token HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
"client_secret":"6pphytzx8qklfa2wi23wgiyil",
"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
"scope":"openid",
"grant_type":"refresh_token",
}
```

RETS Server Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{ "access_token":"645nhg6ofaxunp2hfj0pou8r0",
"refresh_token":"3o0iipzrpiknijyxtjrugkt29",
"expires_in":3600
}
```

The new pair should be saved as a reference to the current session ID. Any old access and refresh tokens are invalid, and should be deleted.

PHP Example

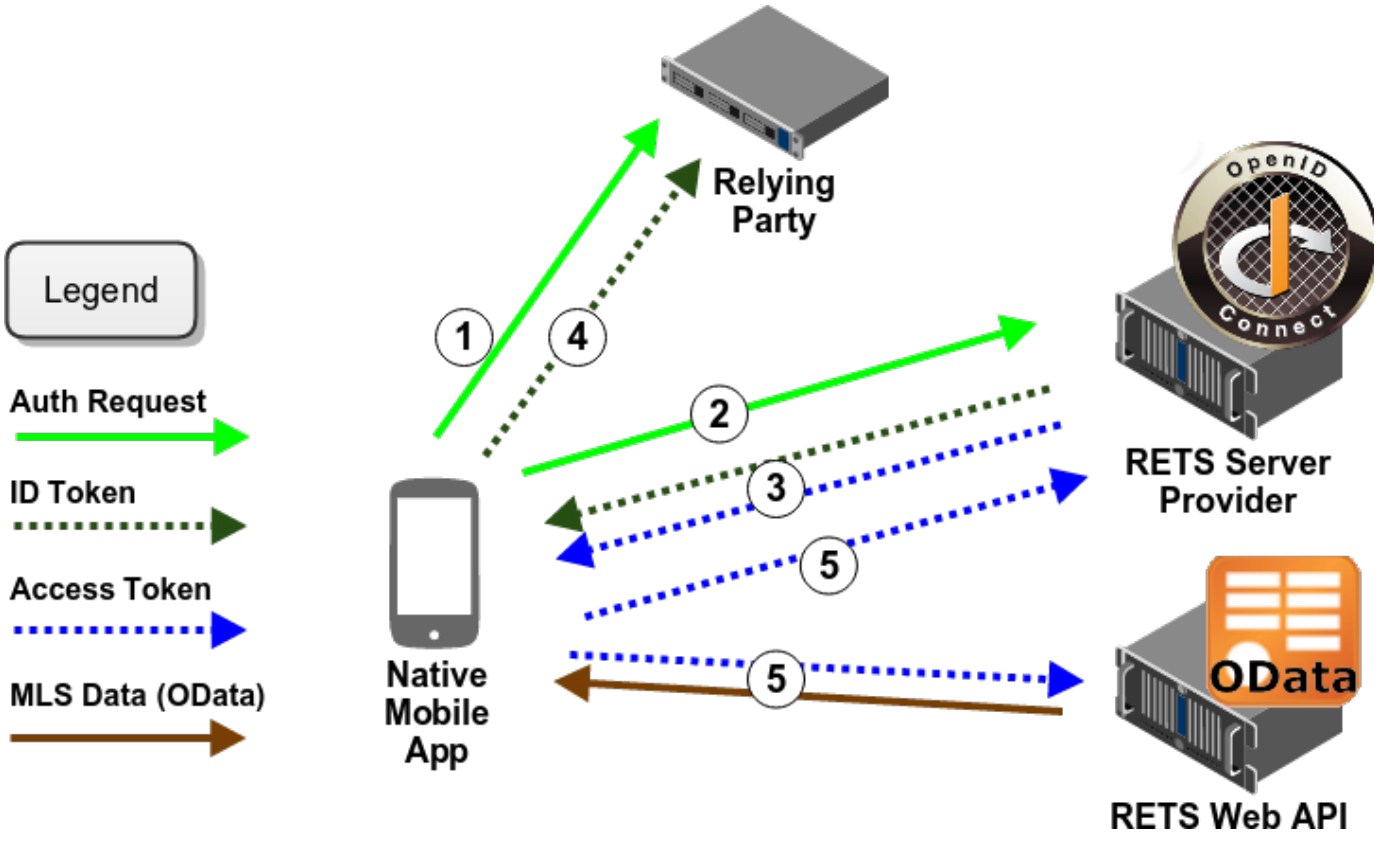
Snippet of dashboard.php

```
$status = curl_getinfo($ch, CURLINFO_HTTP_CODE);
if ( $status == 401 )
{
  $ch = curl_init();
  curl_setopt($ch, CURLOPT_URL, "https://rets.example.com/grant");
  curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-Type:
application/json'));
  curl_setopt($ch, CURLOPT_POST, 1);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
  curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode(array(
    "refresh_token":$refresh_token,
    "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
    "client_secret":"6pphytzx8qklfa2wi23wgiyil",
    "redirect_uri": "https://app.example.com/callback.php",
    "grant_type":"refresh_token"
  )));
  $response = curl_exec($ch);
  curl_close($ch);
  $response = json_decode($response);
  $access_token = $response["access_token"];
  $refresh_token = $response["refresh_token"];
  # Calculate the timestamp a refresh will be needed at
  $expires_at = strftime("%Y-%m-%d %H:%M:%S", time() +
$response["expires_in"]);
  $sql = "delete from keys where session_id = '" . session_id() . "'";
  $sql = "insert into keys (session_id, access_token, refresh_token,
expires_at) "
    . " values ('" . session_id() . "', '$access_token', '$refresh_token',
'$expires_at')";
}
```

2.1.4 Implicit Flow

The Implicit flow is tailored for mobile native applications, or simple web applications that primarily use Javascript to render the view. As the diagram below shows, the entire process is heavily controlled by the client. As a result, the Implicit flow in OAuth2 was more vulnerable to security attacks. With the introduction of the ID Token in OpenID Connect, this process has become more secure.

The decision of allowing the Implicit flow is up to the RETS Server Provider and RETS Web API vendor. Relying Parties should check the Provider's Discovery document to see if the Implicit mode is supported.



Refer to the diagram above to see how the Implicit flow works. The Native Mobile App in this diagram can be replaced with the Client web browser when using Javascript.

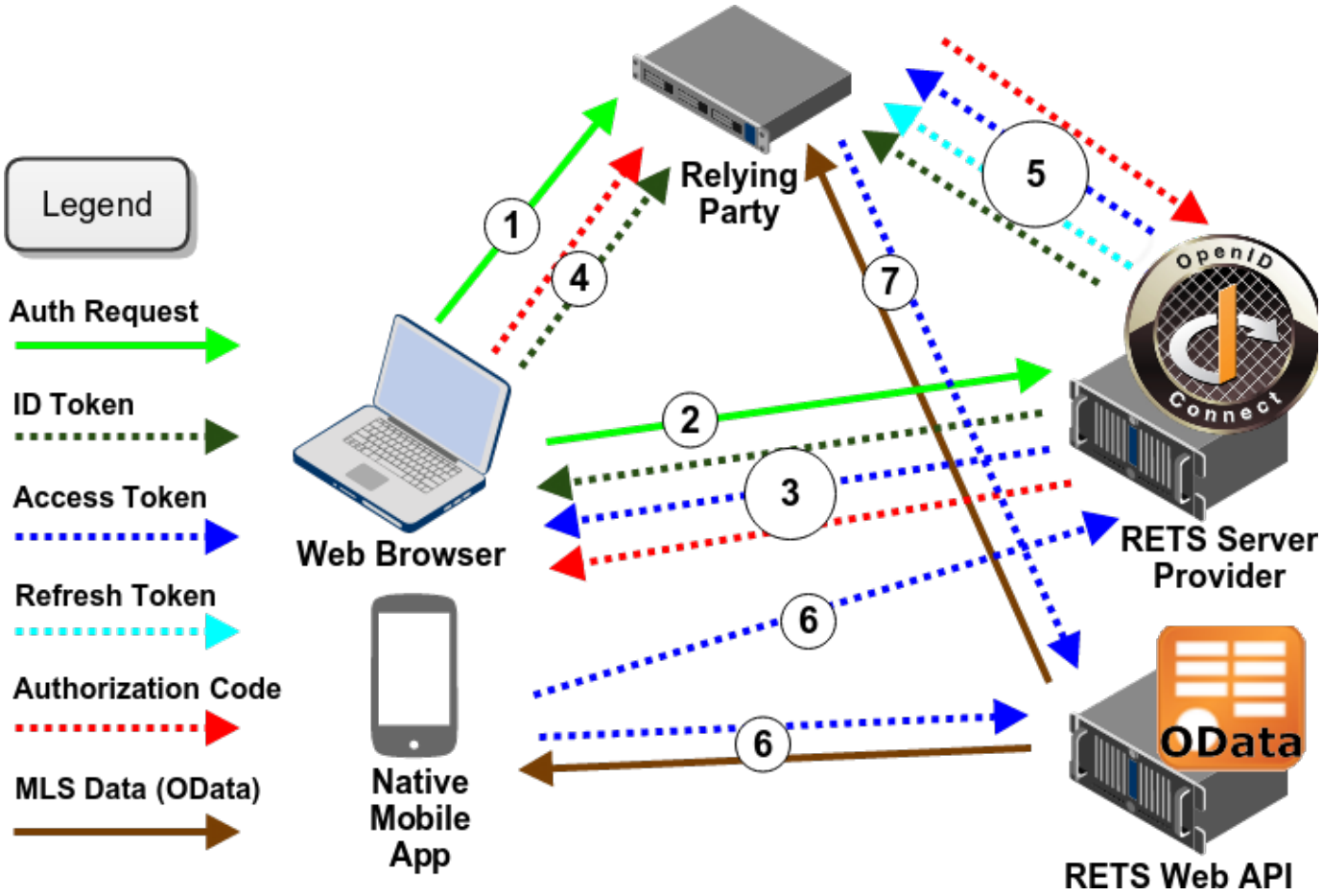
1. If using a web browser, it requests the content from the Relying Party. If using a native app, it displays a login page with a list of Provider's to log in with
    a. Member chooses a login provider
    b. The native app discovers the Provider's endpoints, and checks if the Provider supports the Implicit flow (cached)
2. Native app sends the client_id and redirect_uri to the Provider's authorization endpoint
    a. Member provides a username/password
    b. Member authorizes the native app to access MLS data
3. RETS Server Provider responds with an ID Token, and/or Access Token to the native app
    a. Client validates the ID Token
    b. The Access Token is stored in a secure location and verified against the ID Token at_hash signature
    c. Note: Refresh tokens are not allowed in Implicit
4. Native app uses the ID Token as a secure session cookie with the Relying Party (optional)
    a. Relying Party also validates the ID Token
5. Native app uses the Access Token to retrieve:
    a. Additional Claims (profile info) about the Member from the Provider's UserInfo endpoint
    b. OData API requests against the RETS Web API

The details of the requests and responses are very similar to the Authorization Code flow, with a few small changes. Refer to OpenID Connect Core Section 3.2.1 for more details.

2.1.5 Hybrid Flow

The Hybrid flow is a combination of both Authorization Code and Implicit flows. The primary use case for this is a Relying Party website that needs MLS data, and also has a mobile companion app that needs data. With the previous Web API Security v1.0.1 specification, the solution to this use case using OAuth2 required a proxy service living on the Relying Party to ship MLS Data to a mobile app. The communication between the native app and the Relying Party was then an out-of-band decision, which impacts interoperability.

OpenID Connect's Hybrid flow solves this problem by giving an access token to the native mobile app, and an Authorization Code to the Relying Party in a single operation. Both entities can then access the RETS Web API simultaneously.



Refer to the diagram above to see how the Hybrid flow works.

1. The Client web browser requests the Relying Party site, or native displays a login page with a list of Providers
    a. Member chooses a login provider
    b. The native app or browser discovers the Provider's endpoints, and checks if the Provider supports the Hybrid flow (cached)
2. Client web browser or native app sends the client_id and redirect_uri to the Provider's authorization endpoint
    a. Member provides a username/password
    b. Member authorizes the Relying Party and/or native app to access MLS Data
3. RETS Server Provider responds with an ID Token, Access Token, and Authorization Code to the native app or Client web browser
    a. Client or native app validates the ID Token
    b. The native app stores the Access Token in a secure location and verifies it against the ID Token `at_hash` signature
    c. The Authorization Code is verified against the ID Token's `c_hash` signature
4. The Client web browser or native app sends the Authorization Code to the Relying Party
    a. Optionally, the ID Token may be used as a session cookie with the Relying Party
    b. The Relying Party verifies the Authorization Code against the ID Token's `c_hash` signature
5. The Relying Party uses the Authorization Code to request its own ID Token, Access Token and Refresh Token
6. The native mobile app or Client web browser can use the Access Token to retrieve:
    a. Additional Claims (profile info) about the Member from the Provider's UserInfo endpoint
    b. OData API requests against the RETS Web API
7. The Relying Party uses its own Access Token to retrieve:
    a. Additional Claims (profile info) about the Member from the Provider's UserInfo endpoint
    b. OData API requests against the RETS Web API
    c. Uses the Refresh Token to request another Access Token after it expires

The details of the requests and responses are very similar to the Authorization Code flow, with a few small changes. Refer to OpenID Connect Core Section 3.3.1 for more details.

2.2 - OpenID Connect RETS Server Provider

The RETS Server Provider **must** implement three basic features:

1. Register new Relying Parties (manual or automated)
2. Authorize endpoint (OAuth2 + OpenID Connect)
3. Token endpoint (OAuth2 + OpenID Connect)

Optionally, a RETS Server Provider **should** implement:

1. Discovery service
2. UserInfo endpoint

A RETS API Server (OData) **must** implement one basic feature:

1. Verify access tokens

A few security guidelines

1. All requests **MUST** be over TLS
2. Relying Parties **MUST** be registered with a `redirect_uri` callback, and verified at the `authorize` endpoint

- 2.2.1 OpenID Connect Provider Libraries
- 2.2.2 Discovery service
- 2.2.3 Register New Relying Parties
- 2.2.4 Authorize Endpoint
- 2.2.5 Token Endpoint
- 2.2.6 UserInfo Endpoint
- 2.2.7 Verify Access Tokens
- 2.2.8 Refreshing an Access Token
- 2.2.9 Implicit Flow
- 2.2.10 Hybrid Flow
- 2.2.11 Extra Security Measures

2.2.1 OpenID Connect Provider Libraries

First, check out the official certified libraries page published by the OpenID Foundation:

http://openid.net/developers/libraries/

If your language or platform is not listed there, here's a few extra libraries we've found:

TODO: Find more libraries/toolkits/provider software packages

Identity-as-a-Service Providers:

If you would like to cloudify OpenID Connect, here are a list of IDaaS providers that support OpenID Connect:

- Amazon Cognito
  - Building an App using Amazon Cognito and an OpenID Connect Identity Provider
  - Use Amazon Cognito in your website for simple AWS authentication
  - Understanding Amazon Cognito Authentication
- Microsoft Azure Active Directory
- Salesforce
  - Digging Deeper into OAuth2
  - Inside OpenID Connect
- Auth0
- CA Technologies Mobile API Gateway (documentation)
- Axway API Gateway (demo)
- WSO2 Identity Server (cloud service)

2.2.2 Discovery service

Reference the OpenID Connect Discovery specification. The Discovery service is optional for RETS Server Providers to implement. However, it's very simple and is a relatively static document. It describes where the endpoints live for Authorization, Token, and UserInfo. Additionally, it allows the Provider to advertise which features are supported. OpenID Connect client libraries use this information to know how to behave when communicating with the Provider.

CORS Support
If the RETS Server Provider intends to support the Implicit or Hybrid modes, you should also support Cross-Origin Resource Sharing (CORS) headers at the Discovery service endpoint. Reference this mailing list conversation for the background

A few of the important JSON properties are listed here:

issuer

This advertises the base URL of the RETS Server Provider, and matches the issuer claim in the ID Token.

authorization_endpoint

This advertises the URI of the Authorization endpoint

token_endpoint

This advertises the URI of the Token endpoint

userinfo_endpoint

This advertises the URI of the UserInfo endpoint

response_types_supported

This is an array of all the OpenID Connect response_type values that the Provider supports. This allows a Provider to explicitly communicate if they support the Implicit, Authorization Code, and Hybrid flows. ("token" references the access token)

response_type	Flow
id_token	Implicit
id_token token	Implicit
code id_token	Hybrid
code token	Hybrid
code id_token token	Hybrid
code	Authorization Code

claims_supported

This allows an OpenID Connect Provider to advertise which of the standard claims are supported in OpenID Connect Core Section 5.1. Additionally, the Provider can advertise custom claims here as well. In the interests of RESO standards, keep these as Data Dictionary Member terms, like MemberMlsId, OfficeKey, MemberNrdsId, etc.

scopes_supported

Similar to claims_supported, scopes are like alias groups for a set of individual standard claims.

jwks_uri

URI of the Provider's JSON Web Key Set document. This document publishes the public certificate used by the Relying Party to verify ID Tokens. Read more about JSON Web Keys in RFC 7517 Section 4. For the most part, a JWT library should perform this work for you. Give it an X.509 public certificate, and it returns the JSON for the JWKS URI response. (Very simple) A list of JWT libraries is available at the OpenID Connect Libraries page, and these are more prevalent than OpenID Connect libraries.

2.2.3 Register New Relying Parties

The registration process for a new Relying Party can be a manual, human process, or include varying levels of automation at the discretion of the RETS Server Provider. OpenID Connect defines a new specification called the OpenID Connect Dynamic Client Registration. This allows a RETS Server Provider to remove (or reduce) the human interaction when registering a new Relying Party. The prime use case is obviously for public entities like Google or Amazon, however there is an important clause in Section 3:

> *The OpenID Provider MAY require an Initial Access Token that is provisioned out-of-band (in a manner that is out of scope for this specification) to restrict registration requests to only authorized Clients or developers.*

This means that the automated registration process can be restricted, rate limited, and authorized by a separate OAuth2 scope. Picture it as an API that is accessible only to developers. The credentials for this API are given via any means the RETS Server Provider chooses.

After a developer is given access to the Registration API, they're allowed to create new client_ids for themselves. This reduces the administration overhead on the RETS Server Provider and allows for quicker integrations.

Given a manual or automated process, A RETS Server Provider **MUST** still register a Relying Party's `redirect_uri` callback with a given `client_id`. Usually this is a one-to-one relationship. One `client_id` represents one Relying Party callback URL. This **MAY** be a one-to-many relationship, shown in Section 2. Each `client_id` has a `client_secret`, which is effectively a password and should be kept confidential.

2.2.4 Authorize Endpoint

The `authorize` endpoint can be any URL name chosen by the RETS Server Provider, and should be advertised in the Discovery document as a `uthorization_endpoint`. This URL **MUST** use TLS. The Relying Party will redirect a request from the Client browser to this endpoint:

Client Request

```
GET /authorize?client_id=7d1wp67gl1oo8wsc8ks4csgsk
    &scope=openid
    &response_type=code
    &state=o5n9ki8kpil86vl9j11uujbn41
    &redirect_uri=https://app.example.com/callback.php HTTP/1.1
Host: rets.example.com
```

Note that the only difference between OpenID Connect and OAuth2 is the `scope=openid` parameter. Without this, the RETS Server Provider can fall back to the functionality defined in RESO Web API Security v1.0.1. (And another option is to define a completely different endpoint for OpenID Connect)

The RETS Server Provider must match a pre-registered `client_id` and `redirect_uri` pair with the request parameters. The RETS Server Provider should also obtain end-user consent, in Section 3.1.2.4. Then, generate a new `Authorization code`. This `code` **SHOULD** expire in ten minutes. Read more about expirations in OAuth2 RFC6749 Section 4.1.2.

> **Hint**
> Database systems that have automatic TTL expirations work great for this. MongoDB and Redis are good examples.

Redirect the Client browser to the `redirect_uri` with the `code` parameter appended to the URL. Relay the same `state` parameter the API Consumer provided to prevent cross site forgery attacks.

Client Response

```
HTTP/1.1 302 Found
Location:
http://app.example.com/callback?code=5i46ka0uur7soktiyca6lcczt&state=o5n9k
i8kpil86vl9j11uujbn41
```

PHP Example

authorize.php

```
$sql = "select redirect_uri from consumers where client_id='"  .
$_REQUEST["client_id"] . "'";
if ( $redirect_uri != $_REQUEST["redirect_uri"] )
{
   # Error response defined in OAuth2 RFC 6749 Section 4.1.2.1
}
$new_code = generate_token();
$sql = "insert into codes (client_id, code, expires_at) "
  . "values('". $_REQUEST['client_id'] . "', '$new_code', now() + 600)";
header("Location: $redirect_uri?code=$new_code&state=$_REQUEST['state']");
```

2.2.5 Token Endpoint

The RETS Server Provider's **`token`** endpoint is responsible for authorization code key exchanges and refreshing access tokens. The `token` end point can be any URL name chosen by the RETS Server Provider, and should be advertised in the Discovery document as `token_endpoint`. This URL **MUST** use TLS.

All requests to the `token` endpoint require a `client_id`, `client_secret`, and `redirect_uri` at a minimum.

Authorization Codes

Relying Party Request

```
POST /token HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"code":"5i46ka0uur7soktiyca6lcczt",
 "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
 "client_secret":"6pphytzx8qklfa2wi23wgiyil",
 "redirect_uri":"http://app.example.com/callback.php",
 "grant_type":"authorization_code" }
```

First the RETS Server must validate the `client_id`, `client_secret`, and `redirect_uri`. Then verify that the `code` parameter matches with the `client_id`, and that it has not expired. If the verification is successful, generate new ID Token, access and refresh tokens in a JSON response. The RETS Server Provider **MUST** delete or invalidate authorization codes after a Relying Party uses them. An Authorization Code can only be used once.

API Consumer Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{"token_type":"Bearer",
 "access_token":"2w9wc3b8565ajpj4i9v68ivlv",
 "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
 "id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazci...",
 "expires_in":3600 }
```

Refreshing Access Tokens

Relying Party Request

```
POST /token HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
 "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
 "client_secret":"6pphytzx8qklfa2wi23wgiyil",
 "scope":"openid",
 "grant_type":"refresh_token"}
```

First the RETS Server must validate the `client_id`, `client_secret`. Then, verify the `refresh_token` is valid and pairs with the `client_id`. Although refresh tokens do not carry an expiration, they can be manually revoked from the RETS Server Provider by the MLS Member, or if security tripwires are triggered. This effectively blocks the Relying Party from accessing data on behalf of the specific MLS Member.

If the verification is successful, generate new access and refresh tokens in a JSON response. The RETS Server Provider **MAY** optionally create a refreshed ID Token with a few requirements as described in OpenID Connect Core Section 12.2. The RETS Server Provider **MUST** delete or invalidate old access and refresh tokens after the Relying Party uses them.

Relying Party Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{"access_token":"645nhg6ofaxunp2hfj0pou8r0",
 "refresh_token":"3o0iipzrpiknijyxtjrugkt29",
 "id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazci...",
 "expires_in":3600}
```

2.2.6 UserInfo Endpoint

The UserInfo endpoint is a resource that returns Claims, or profile information, about the current user. Read OpenID Connect Core Section 5.3 for detailed information. The UserInfo URI is protected by an access token and returns a JSON structure of data. A Relying Party can also request Claims be added to the ID Token. The UserInfo endpoint's purpose is to transfer Claims that might be too large for an ID Token. The general goal is to request an ID Token with only the required, persistent parameters to identify a user. If additional profile information is needed, request them from the UserInfo endpoint.

Here's a simple example:

```
GET /userinfo
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUz...
-------------------------
Content-Type: application/json

{
  "sub":"201510581635287728470000000"
  "name":"Bob"
  "preferred_username":"agent_bob",
  "email":"bob@provider.com",
  "zoneinfo":"America/Chicago",
  "MemberNrdsId":"123456789"
}
```

Notice that this response included a custom Claim. A Relying Party can know what Claims to expect based on the `claims_supported` property in the Discovery metadata. This example does not contain an exhaustive list of the possible claims. To see a list of the standard claims, check out OpenID Connect Core Section 5.1.

2.2.7 Verify Access Tokens

On every API request, the RETS API Server must verify that the access token is valid and has not expired.

API Consumer Request

```
GET /RESO/OData/Properties.svc/Properties('ListingId3') HTTP/1.1
Host: rets.example.com
Authorization: Bearer 2w9wc3b8565ajpj4i9v68ivlv
```

On success, return the OData response using the identity tied to this `access_token`

On failure, return an HTTP 401 to tell the Relying Party that the access token is invalid

Failure Response

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm='RETS Server', error='expired_token'
Content-Type: application/json

{ "message": "Access token has expired" }
```

A Note on Federated Access Tokens

With OpenID Connect, it's possible to separate the RETS API Server from the OpenID Connect RETS Server Provider. The RETS Server Provider maintains user sessions and is responsible for creating tokens. If the access token format is a self signed JSON Web Token, the RETS API Server can verify access tokens using the issuer, subject, and expiration fields in the JWT. As long as the expiration time is relatively short, a premature revocation of the authorization by the RETS Server Provider will be reflected on the next refresh token request.

The mechanism of which access tokens are constructed, verified, and revoked is out of the scope of OpenID Connect and this document. OpenID Connect just opens the possibility for such a system.

2.2.8 Refreshing an Access Token

- 2.2.8.1 An expired access token returns HTTP 401
- 2.2.8.2 Relying Party makes a request to the RETS Server Provider's token endpoint
- 2.2.8.3 Relying Party saves the access and refresh tokens

2.2.8.1 An expired access token returns HTTP 401

An expired access token returns HTTP 401 Unauthorized on a given API request.

```
GET /my/listings HTTP/1.1

Host: rets.example.com

Authorization: Bearer 2w9wc3b8565ajpj4i9v68ivlv
```

Response:

```
HTTP/1.1 401 Unauthorized

WWW-Authenticate: Bearer realm='RETS Server', error='expired_token'

Content-Type: application/json

        { "message": "Access token has expired" }
```

2.2.8.2 Relying Party makes a request to the RETS Server Provider's token endpoint

Relying Party makes a request to the RETS Server Provider's token endpoint

The previously saved refresh token is used to request another access token. The client_id and client_secret pair are required.

Request:

```
POST /token HTTP/1.1
Host: rets.example.com
Content-Type: application/json


{"client_id":"1234",
"client_secret":"dedyhcrynzeza6vljncfn5mxj",
"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
"redirect_uri":"https://test_app.example.com/callback",
"grant_type":"refresh_token",
"scope":"openid" }
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json


{ "access_token":"645nhg6ofaxunp2hfj0pou8r0",
"refresh_token":"3o0iipzrpiknijyxtjrugkt29",
"id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazci...",
"expires_in":3600 }
```

2.2.8.3 Relying Party saves the access and refresh tokens

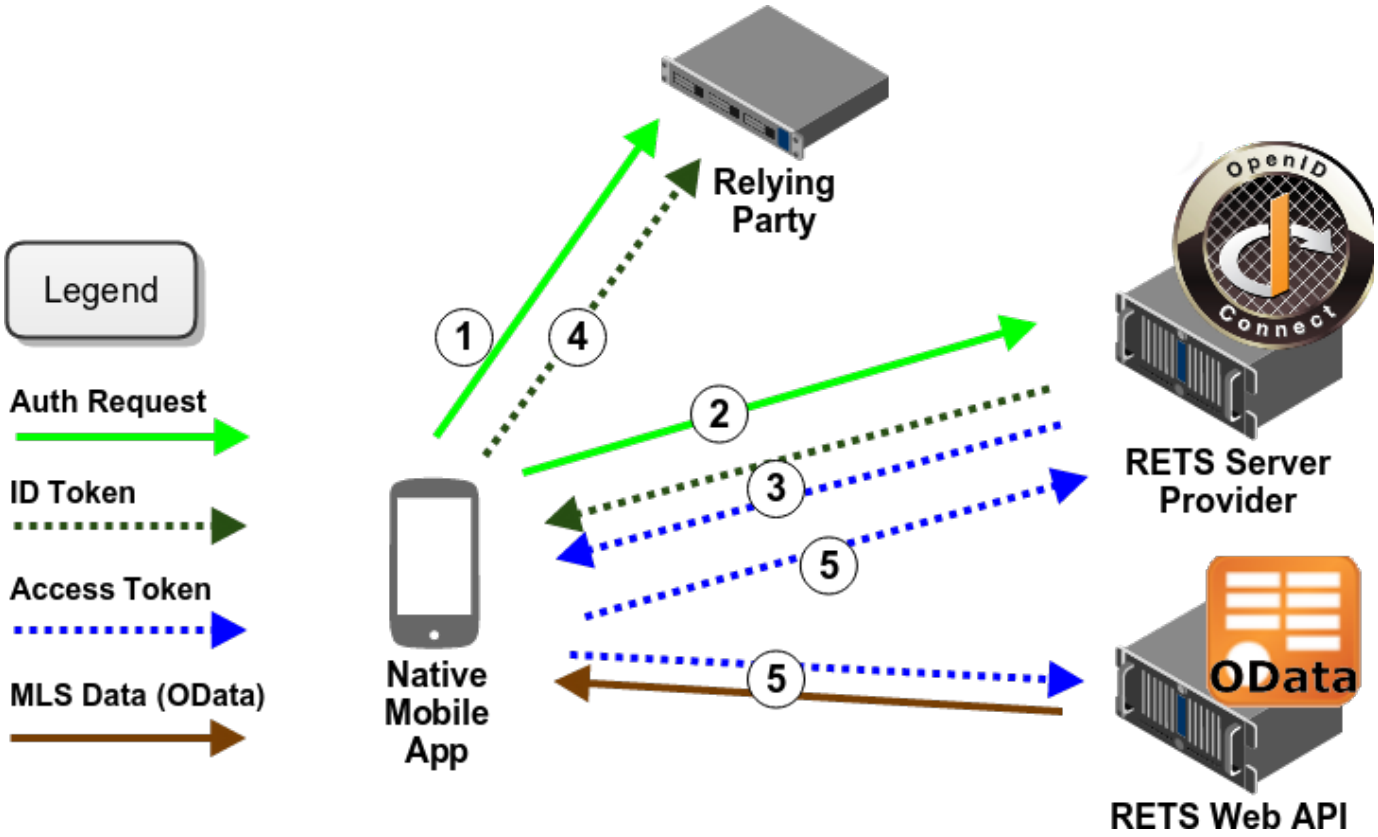
Relying Party saves the access and refresh tokens for future use.

The new access and refresh tokens are saved in a database and used in subsequent RETS API Server requests. Any old access and refresh tokens should be discarded.

2.2.9 Implicit Flow

The Implicit flow is tailored for mobile native applications, or simple web applications that primarily use Javascript to render the view. As the diagram below shows, the entire process is heavily controlled by the client. As a result, the Implicit flow in OAuth2 was more vulnerable to security attacks. With the introduction of the ID Token in OpenID Connect, this process has become more secure.

The decision of allowing the Implicit flow is up to the RETS Server Provider and RETS Web API vendor. The Discovery document's `response_types_supported` property defines which flows the Provider supports.



Refer to the diagram above to see how the Implicit flow works. The Native Mobile App in this diagram can be replaced with the Client web browser when using Javascript.

1. If using a web browser, it requests the content from the Relying Party. If using a native app, it displays a login page with a list of Provider's to log in with
    a. Member chooses a login provider
    b. The native app discovers the Provider's endpoints, and checks if the Provider supports the Implicit flow (cached)
2. Native app sends the client_id and redirect_uri to the Provider's authorization endpoint
    a. Member provides a username/password
    b. Member authorizes the native app to access MLS data
3. RETS Server Provider responds with an ID Token, and/or Access Token to the native app
    a. Client validates the ID Token
    b. The Access Token is stored in a secure location and verified against the ID Token at_hash signature
    c. Note: Refresh tokens are not allowed in Implicit
4. Native app uses the ID Token as a secure session cookie with the Relying Party (optional)
    a. Relying Party also validates the ID Token
5. Native app uses the Access Token to retrieve:
    a. Additional Claims (profile info) about the Member from the Provider's UserInfo endpoint
    b. OData API requests against the RETS Web API

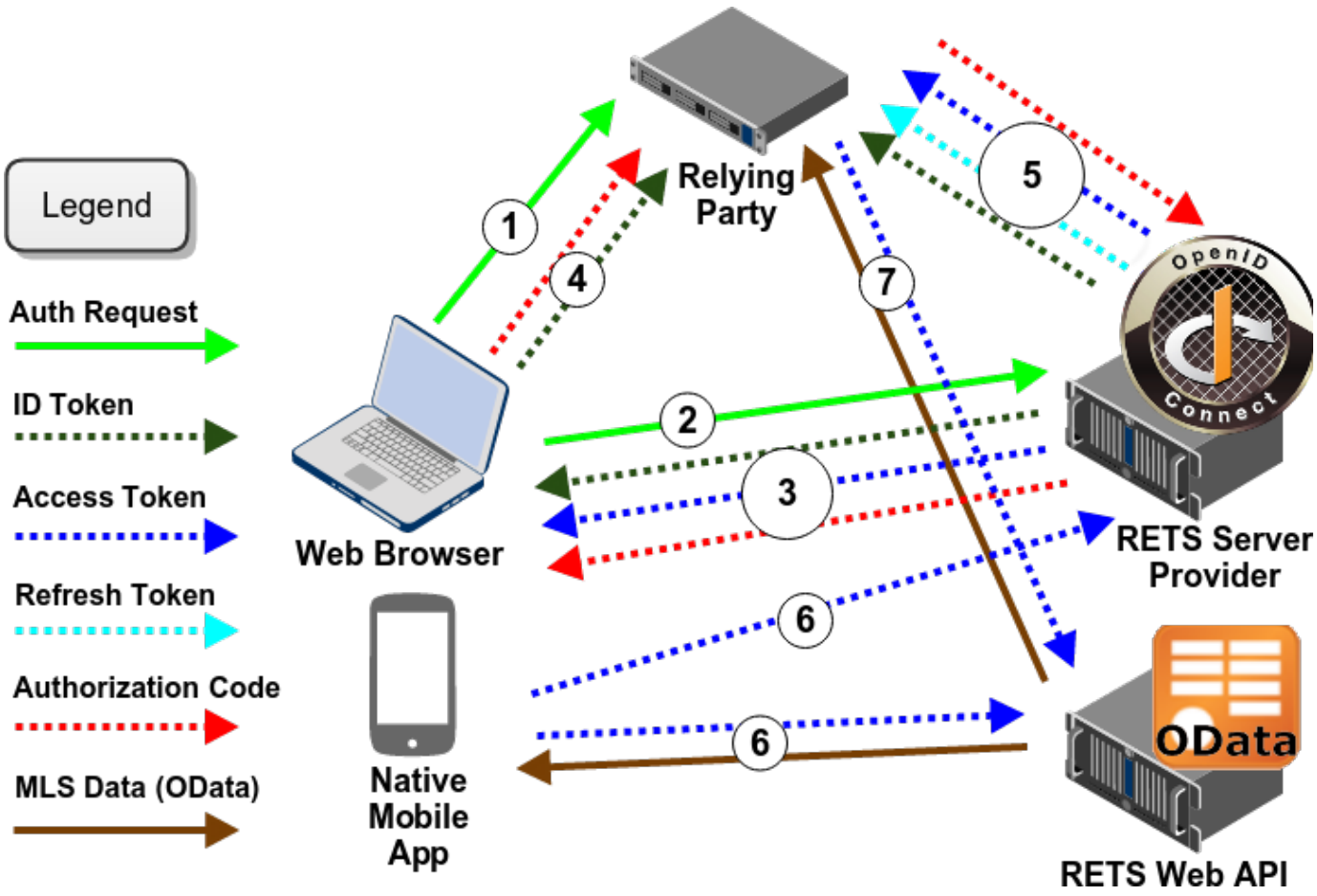
The details of the requests and responses are very similar to the Authorization Code flow, with a few small changes. Refer to OpenID Connect Core Section 3.2.1 for more details.

2.2.10 Hybrid Flow

The Hybrid flow is a combination of both Authorization Code and Implicit flows. The primary use case for this is a Relying Party website that needs MLS data, and also has a mobile companion app that needs data. With the previous Web API Security v1.0.1 specification, the solution to this use case using OAuth2 required a proxy service living on the Relying Party to ship MLS Data to a mobile app. The communication between the native app and the Relying Party was then an out-of-band decision, which impacts interoperability.

OpenID Connect's Hybrid flow solves this problem by giving an access token to the native mobile app, and an Authorization Code to the Relying Party in a single operation. Both entities can then access the RETS Web API simultaneously.

The decision of allowing the Hybrid flow is up to the RETS Server Provider and RETS Web API vendor. The Discovery document's `response_types_supported` property defines which flows the Provider supports.



Refer to the diagram above to see how the Hybrid flow works.

1. The Client web browser requests the Relying Party site, or native displays a login page with a list of Providers
    a. Member chooses a login provider
    b. The native app or browser discovers the Provider's endpoints, and checks if the Provider supports the Hybrid flow (cached)
2. Client web browser or native app sends the client_id and redirect_uri to the Provider's authorization endpoint
    a. Member provides a username/password
    b. Member authorizes the Relying Party and/or native app to access MLS Data
3. RETS Server Provider responds with an ID Token, Access Token, and Authorization Code to the native app or Client web browser
    a. Client or native app validates the ID Token
    b. The native app stores the Access Token in a secure location and verifies it against the ID Token `at_hash` signature
    c. The Authorization Code is verified against the ID Token's `c_hash` signature
4. The Client web browser or native app sends the Authorization Code to the Relying Party
    a. Optionally, the ID Token may be used as a session cookie with the Relying Party
    b. The Relying Party verifies the Authorization Code against the ID Token's `c_hash` signature
5. The Relying Party uses the Authorization Code to request its own ID Token, Access Token and Refresh Token
6. The native mobile app or Client web browser can use the Access Token to retrieve:
    a. Additional Claims (profile info) about the Member from the Provider's UserInfo endpoint
    b. OData API requests against the RETS Web API
7. The Relying Party uses its own Access Token to retrieve:

a. Additional Claims (profile info) about the Member from the Provider's UserInfo endpoint
b. OData API requests against the RETS Web API
c. Uses the Refresh Token to request another Access Token after it expires

The details of the requests and responses are very similar to the Authorization Code flow, with a few small changes. Refer to OpenID Connect Core Section 3.3.1 for more details.

2.2.11 Extra Security Measures

Although not defined in OpenID Connect or OAuth2 RFC, there are a few extra security measures a RETS Server may implement for extra security

IP Address Accounting

Keep a history of all IP addresses a Relying Party uses. Most server-side applications should not cycle through IPs very often. If there is a sudden influx of many IP addresses seen from a given client_id or `access_token`, invalidate them.

User-Agent Verification

Along with a `redirect_uri` , register an Relying Party's User-Agent. On each API request, verify the requested User-Agent is the same. Return an HTTP-401, or possibly invalidate the client_id and access tokens. A less invasive approach would be to keep a history of User-Agents, and perform a similar algorithm to the IP address accounting.

Rate Limiting

Keep track of the request rate at the `token` and `authorize` endpoints. Brute force attacks can be easily caught and disallowed with these two services. Rate limit API requests by IP address, access_token, and client_id.

Lower Access Token Expirations

Set an access token expiration time of less than 24 hours for production traffic. The lower the expiration time, the quicker access tokens must be discarded, which results in less time for an attacker to use a stolen access token.

OAuth 2.0 Threat Model and Security Considerations

Read RFC 6819.

OpenID Connect Security Considerations

See Section 16 in the OpenID Connect Core specification.

Section 3 - FAQ

Where can I find help with OpenID Connect?

Here is a list of OpenID Foundation mailing lists: http://openid.net/foundation/community/mailing-lists/

Most people should join the General list. Everyone on the list is very helpful, and quick to respond. If you're struggling with something, this should be the first place to ask a question.

Which Flow(s) do I need to implement?

There are three authentication flows with OpenID Connect: Implicit, Authorization Code, and Hybrid. The only requirement is that you implement at least one. Implicit is suited for native mobile apps. Authorization Code is most common for 3-legged APIs, and Hybrid is a combination of both Implicit and Authorization Code.

How can I test my implementation?

Relying Parties may test client libraries against sample Providers:

- https://connect-op.herokuapp.com
- https://rp.certification.openid.net:8080/test_list (RP certification test tools)

RETS Server Providers can test using sample Relying Parties:

- https://TestFormVendor.com (email Cal Heldenbrand for credentials)
- https://connect-rp.herokuapp.com
- https://op.certification.openid.net:60000 (The certification tool, great for testing as well)

Why do some of the callback URIs contain a hash symbol (#) and not a question mark (?) with query string?

When using the Implicit and Hybrid modes, OpenID Connect Core Section 3.2.2.5 states:

> *When using the Implicit Flow, all response parameters are added to the **fragment** component of the Redirection URI, as specified in OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses], unless a different Response Mode was specified.*

The **fragment** portion of a URI is separate from the **query string** portion, and is delimited by the hash symbol. (The same thing used for HTML anchors) What is the reason for this odd decision? The fragment portion of a URI is not transferred to the server side – the browser keeps this data private. After all, it was originally meant for scrolling a browser to a spot in a page, and doesn't make sense to waste the bandwidth on transferring them to the server. This logic becomes a powerful security feature when used with Implicit mode. If a server cannot see the client_id and redirect_uri parameters, then attacks like click-jacking and DNS spoofing are completely useless. Since the intended goal for the Implicit mode is for displaying simple profile information in a view, it is not necessary for a Relying Party's server-side application to see this information.

However, if you need to override this behavior, read about the response_mode parameter in OpenID Connect Core Section 3.1.2.1. This allows the Relying Party to specify the value "query" to change this behavior.

Does OpenID Connect protect my system from password sharing?

Nope.

You still need to use heuristics, two factor authentication, and security warnings to mitigate password sharing.

Section 4 - Authors

Author	Company	Email
Matt Cohen	Clareity Consulting	matt.cohen@clareity.com
Cal Heldenbrand	FBS	cal@fbsdata.com
Mark Lesswing	NAR	
Matt McGuire	Corelogic	

Section 5 - Revision List

- October 2013 – First working draft: RETS Web API Security Group Google Document
- February 2014 – First RESO document: RETS Web API Security v1.0
- October 2014 – First Standardized OAuth2 Version: RESO Web API Security v1.0.1
- October 2015 – OpenID Connect: RESO Web API Security v1.0.3

Section 6 - Appendices

- 6.1 Use Case Diagrams
  - 6.1.1 SP (Service Provider) to SP/IdP (Identity Provider)
  - 6.1.2 SP to IdP to SP Typical three-way authorization
  - 6.1.3 SP to SP/IdP Transparent three-way authorization
  - 6.1.4 SP to SP/IdP Transparent, recurring "on behalf of" authorization
  - 6.1.5 2-legged Client-Server Auth
  - 6.1.6 4-legged Federated Identities
- 6.2 Resources and Links
  - 6.2.1 Help Guides and Introductions
  - 6.2.2 Library Demos and Examples
  - 6.2.3 Identity-as-a-Service Providers

6.1 Use Case Diagrams

The first task of the Authentication & Authorization workgroup was to brainstorm the various use cases that would be required for a Security standard. Before OpenID Connect, no single standard had the ability to solve every use case desired. With the introduction of OpenID Connect, all of the initial use cases are now possible, and some additional use cases have been added as well.

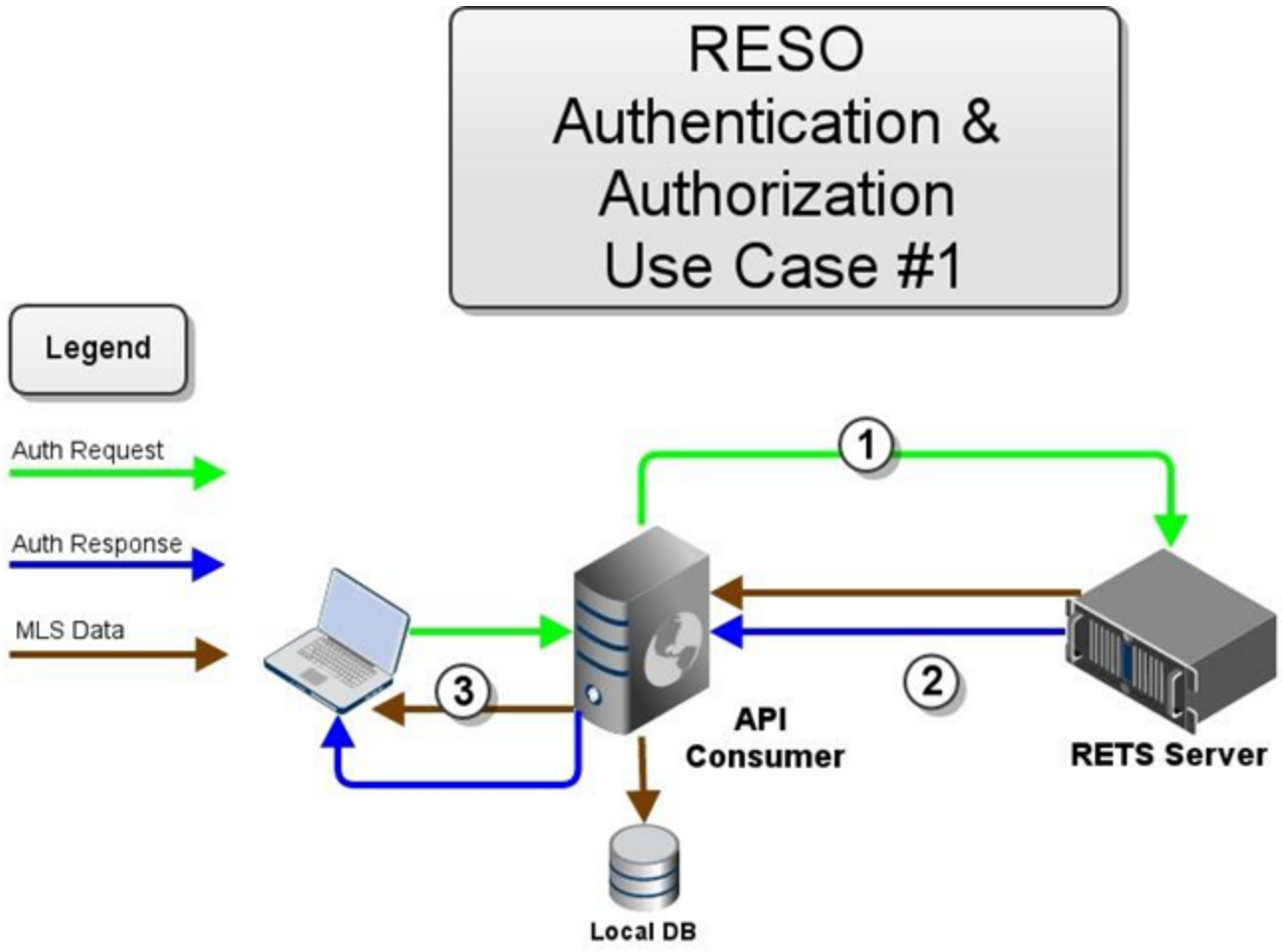
Each diagram has a general algorithm describing it in a non-protocol specific manner. Each page has a short description of how OpenID Connect can be leveraged to solve each use case.

- 6.1.1 SP (Service Provider) to SP/IdP (Identity Provider)
- 6.1.2 SP to IdP to SP Typical three-way authorization
- 6.1.3 SP to SP/IdP Transparent three-way authorization
- 6.1.4 SP to SP/IdP Transparent, recurring "on behalf of" authorization
- 6.1.5 2-legged Client-Server Auth
- 6.1.6 4-legged Federated Identities

6.1.1 SP (Service Provider) to SP/IdP (Identity Provider)

Server or Client to Server authorization (*without human intervention*)

Example: RETS 1.x style flow. A syndicator's recurring bulk download of listing data.



1. An API consumer submits a request for authentication to the RETS server. The API consumer declares its own identity to the RETS server (*Not on behalf of an MLS member*). No human interaction takes place.
2. The RETS server responds with an authentication success message along with any extra authentication session information. It may also respond with MLS data in the same response.
3. A web browser client requests MLS data directly from the API consumer. They may perform authentication using locally stored credentials, which may be independent of the MLS vendor's credentials. The request might also be unauthenticated for IDX sites.

How does OpenID Connect solve this use case?

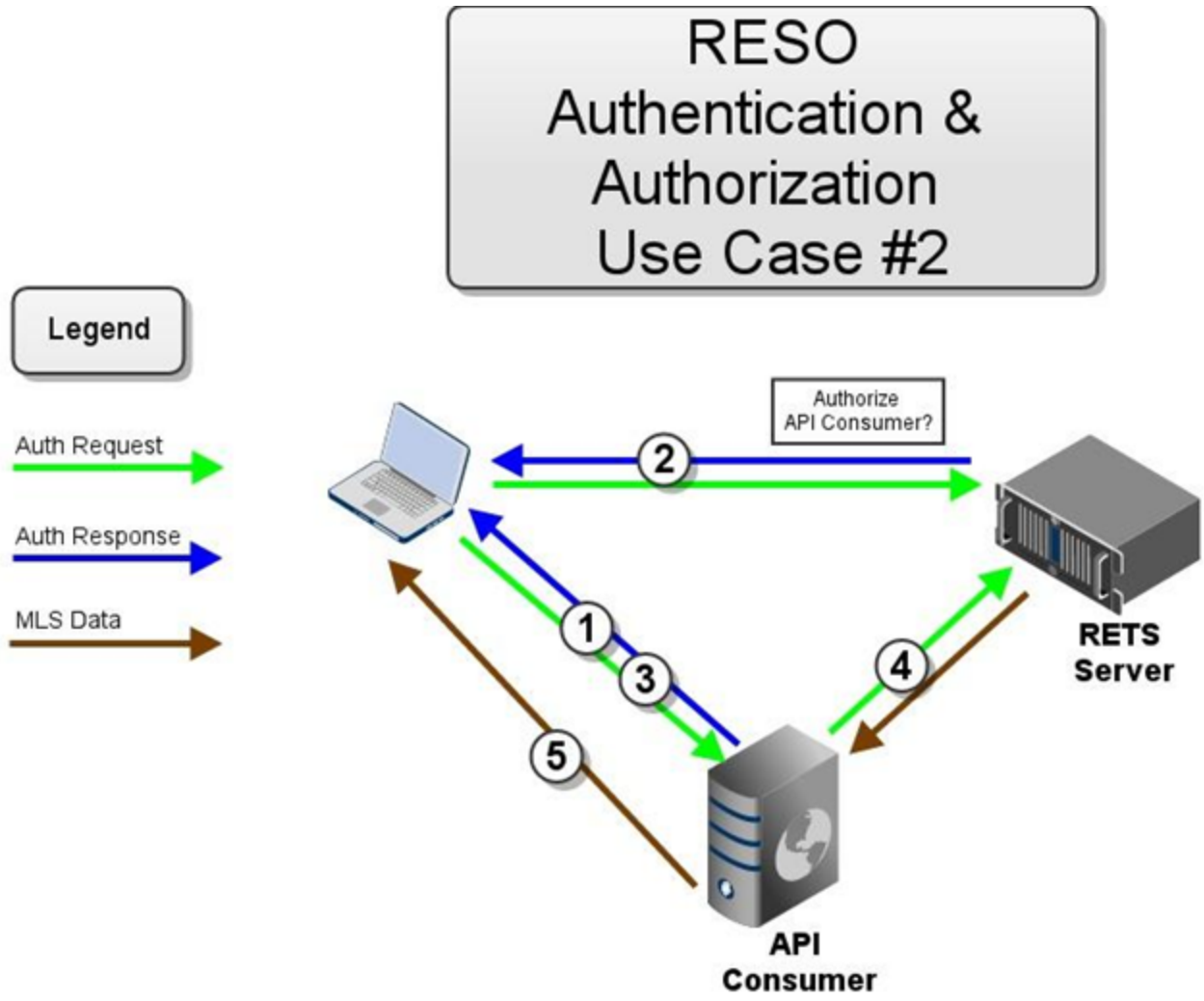
The key phrase in this use case is "human intervention." How much human intervention is acceptable to solve the problem? If a single-click authorization by the MLS Member is acceptable, then the standard Authorization Code flow solves this use case. Refresh tokens can be used indefinitely for the Relying Party (API Consumer) for recurring bulk transfer of data.

If zero human intervention is required, the RETS Server has the ability to make long lived access tokens for the Relying Party with any range of scope. The Relying Party must keep this access token in secure storage. This method has a benefit over Basic or Digest authentication, in that the access token can be revoked by the RETS Server if abuse is detected.

6.1.2 SP to IdP to SP Typical three-way authorization

Typical three-way authorization of a user (*Transient authentication of an API Consumer on behalf of an MLS member*).

Example: A web application that interacts with the MLS on behalf of a user, e.g., a real-time CMA.



1. An unauthenticated MLS member requests access to the API consumer. The API consumer responds with a failure redirect to the RETS server.
2. The MLS member enters a username / password at the RETS IdP. They also agree to authorize the API consumer product to access Site/MLS data..
3. The authenticated MLS member makes another request to the API consumer with valid authentication.
4. The API consumer makes a data request to the RETS server with the authentication supplied by the MLS member. The API consumer processes the data for presentation to the MLS member.
5. The API consumer responds with Site/MLS data to the MLS member.

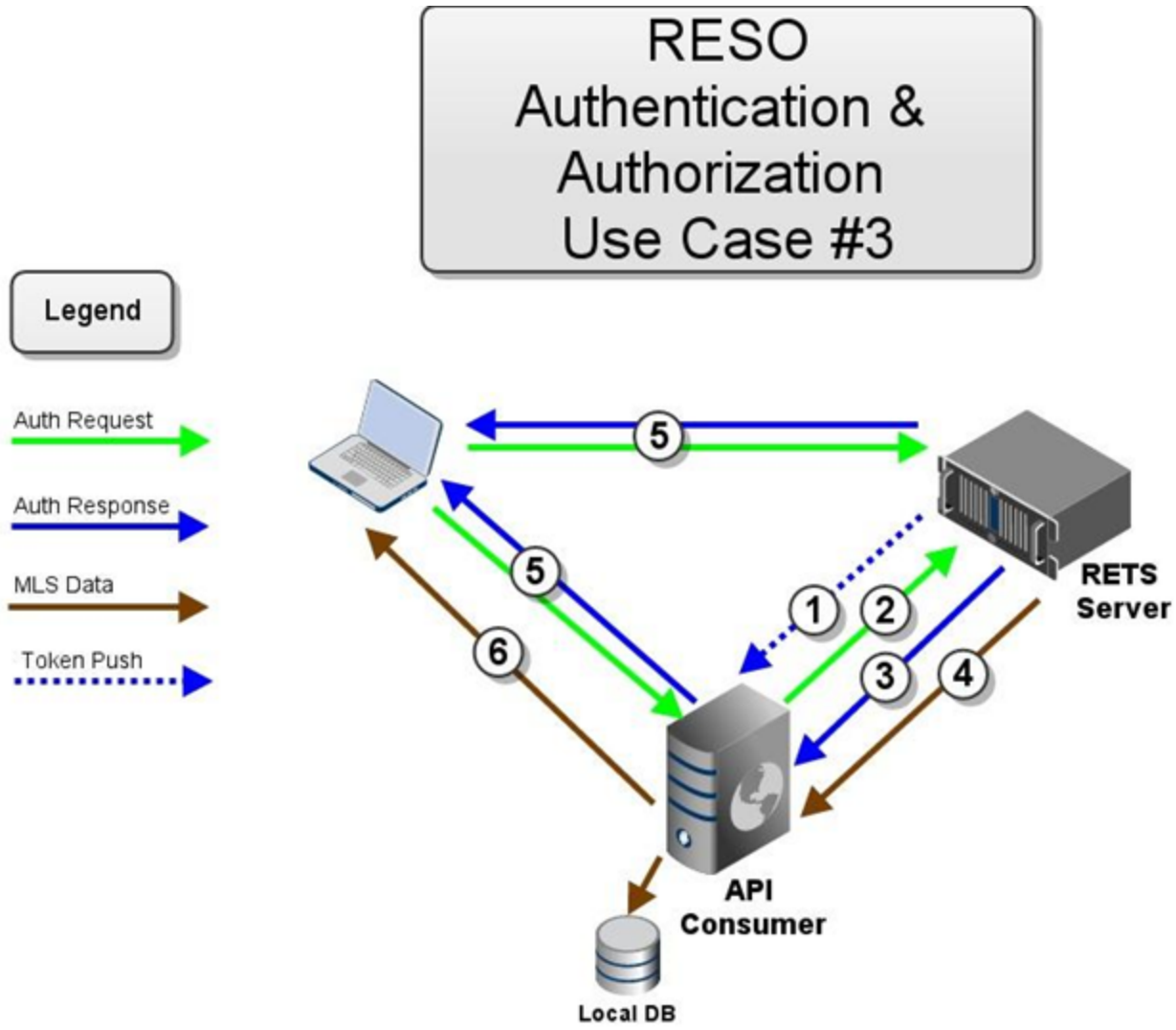
How does OpenID Connect solve this use case?

This is a standard, 3-legged OAuth2 flow, and OpenID Connect is the perfect solution for this case.

6.1.3 SP to SP/IdP Transparent three-way authorization

Transparent three-way authorization of a user.
(Transient authentication of an API consumer on behalf of a user without human intervention)

Example: A VOW provider's validation of eligibility for an existing customer.



1. The Site/MLS (*or an MLS member*) gives an authentication token to the API consumer. This could be a manual process, or a batch process of tokens for many users. This token ***may*** have an expired lifetime.

2. The API consumer uses the token to request authorization from the RETS server. The API consumer could also request VOW authorization of a customer on behalf of an MLS member using that token.

3. The RETS server checks the token for authorization and expiration. They respond with a success, and possibly another (updated) token for this member. For VOW authorizations, this could respond with a token that identifies the MLS member's customer.

4. The API consumer requests Site/MLS data, same as Use Case 6.1.1. (*Use Case: SP to IdP to SP Typical three-way authorization*).

5. At some point in the future, the MLS member authenticates against the MLS, similar as Use Case 6.1.1. (*Use Case: SP to IdP to SP Typical three-way authorization*). The RETS server does not need to ask for authorization again, since this happened in step 1 (*VOW customers would have a similar authentication experience*).

6. The MLS member or VOW customer has access to previously loaded Site/MLS data.

How does OpenID Connect solve this use case?

This use case is similar to Use Case 6.1.2 with the added restriction of "no human intervention." How much human intervention is acceptable to solve the problem? If a single-click authorization by the MLS Member is acceptable, then the standard Authorization Code flow solves this use case. Refresh tokens can be used indefinitely for the Relying Party (API Consumer) for recurring bulk transfer of data.

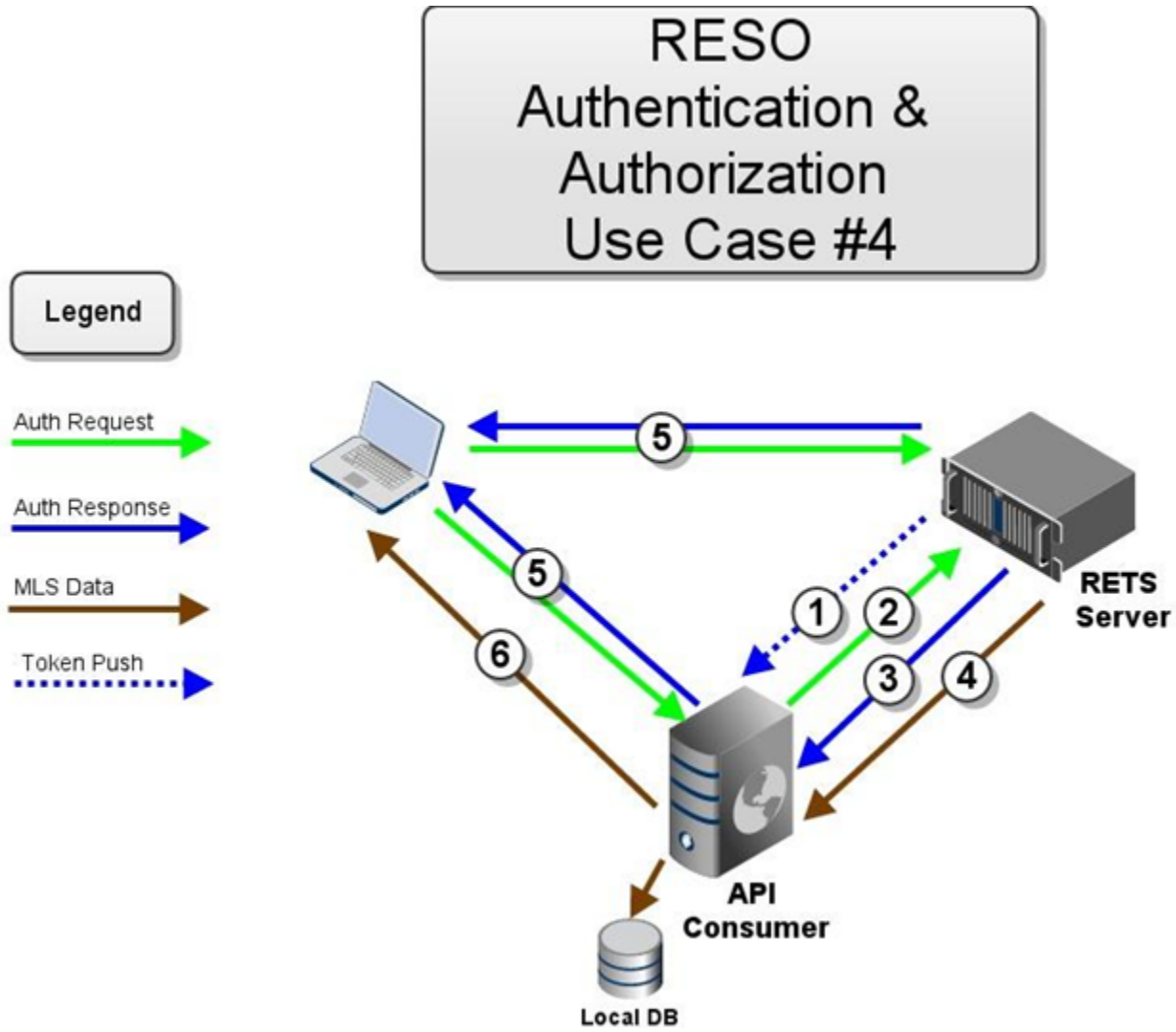
If zero human intervention is required, the RETS Server has the ability to make long lived access tokens for the Relying Party with any range of scope. The Relying Party must keep this access token in secure storage. This method has a benefit over Basic or Digest authentication, in that

the access token can be revoked by the RETS Server if abuse is detected.

6.1.4 SP to SP/IdP Transparent, recurring "on behalf of" authorization

Transparent, recurring "on behalf of" authorization of a user.
(Persistent, transient authentication of an API consumer on behalf of a user without human intervention)

Example: Lead Management software that pulls leads from multiple sources for a given customer.



1. The Site/MLS (*or an MLS member*) gives an authentication token to the API consumer. This could be a manual process, or a batch process of tokens for many users. This token must have an infinite lifetime (*Or perhaps very long*).
2. The API consumer uses the token to request authorization from the RETS server. The API consumer could also request VOW authorization of a customer on behalf of an MLS member using that token.
3. The RETS server verifies the token. They respond with a success or failure. For VOW authorizations, this could respond with a token that identifies the MLS member's customer.
4. The API consumer requests Site/MLS data
5. At some point in the future, the MLS member authenticates against the Site/MLS. The RETS server does not need to ask for authorization again, since this happened in step 1 (*VOW customers would have a similar authentication experience*).
6. The MLS member or VOW customer has access to previously loaded Site/MLS data.

Note: This use case is similar to the standard 3-legged authentication, yet pushes into the area of federated authorizations from the example of "Lead Management software that pulls leads from multiple sources for a given customer."

How does OpenID Connect solve this use case?

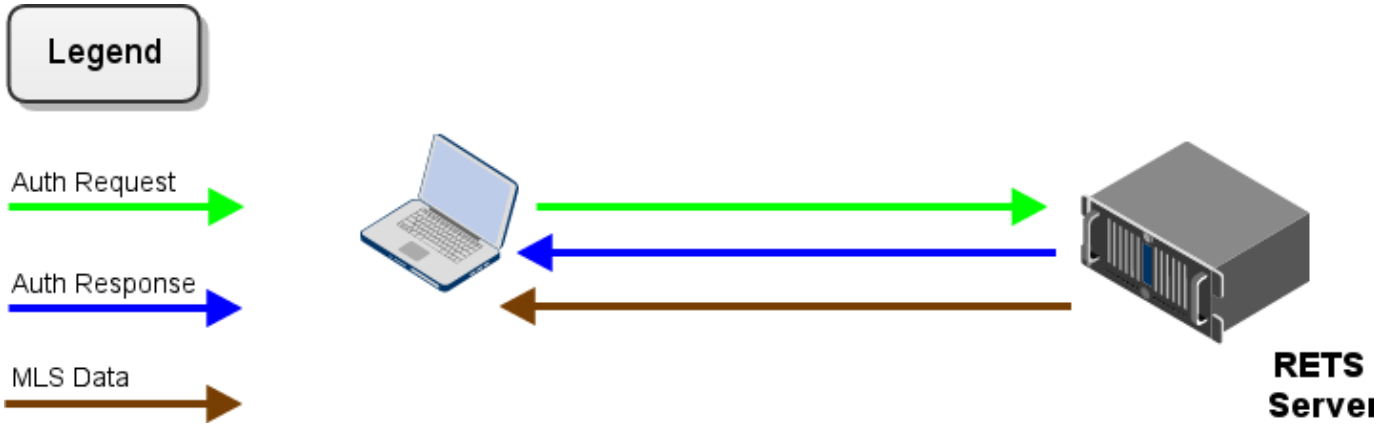
A few key terms set this use case apart from the others. The first, is "persistent, transient authentication" ID Tokens are persistent, because they can be refreshed indefinitely at the RETS Server Provider. ID Tokens are also transient, because they can be given to multiple actors in the system, and each party knows where the ID Token came from (the issuer) and how to validate its authenticity. (The signature) Refer to Use Case 6.1.6 4-legged Federated Identities for more information on this area.

Again, the "without human intervention" is a subjective qualifier to the use case. How much human intervention is acceptable to solve the problem? If a single-click authorization by the MLS Member is acceptable, then the standard Authorization Code flow solves this use case. Refresh tokens can be used indefinitely for the Relying Party (API Consumer) for recurring bulk transfer of data.

If zero human intervention is required, the RETS Server has the ability to make long lived access tokens for the Relying Party with any range of scope. The Relying Party must keep this access token in secure storage. This method has a benefit over Basic or Digest authentication, in that the access token can be revoked by the RETS Server if abuse is detected.

6.1.5 2-legged Client-Server Auth

This is a typical scenario in RETS 1.x. Native OS software that talks directly to a RETS feed. This was disallowed in the previous Web API Security v1.0.1. OpenID Connect's Implicit flow is very similar to this style, and provides a good amount of security when compared to Basic or Digest authentication.



6.1.6 4-legged Federated Identities

The term federation, just like SSO are broadly defined terms. In general, when we think of Single Sign-On, we think of a spoke-hub architecture with the IdP in the center and the Service Providers at each spoke. An identity lives on the Identity Provider (IdP), and each Service Provider (SP) must talk to the IdP in order to receive an identity. The Service Providers cannot talk to each other, nor can an identity on an IdP be transferred to another IdP.

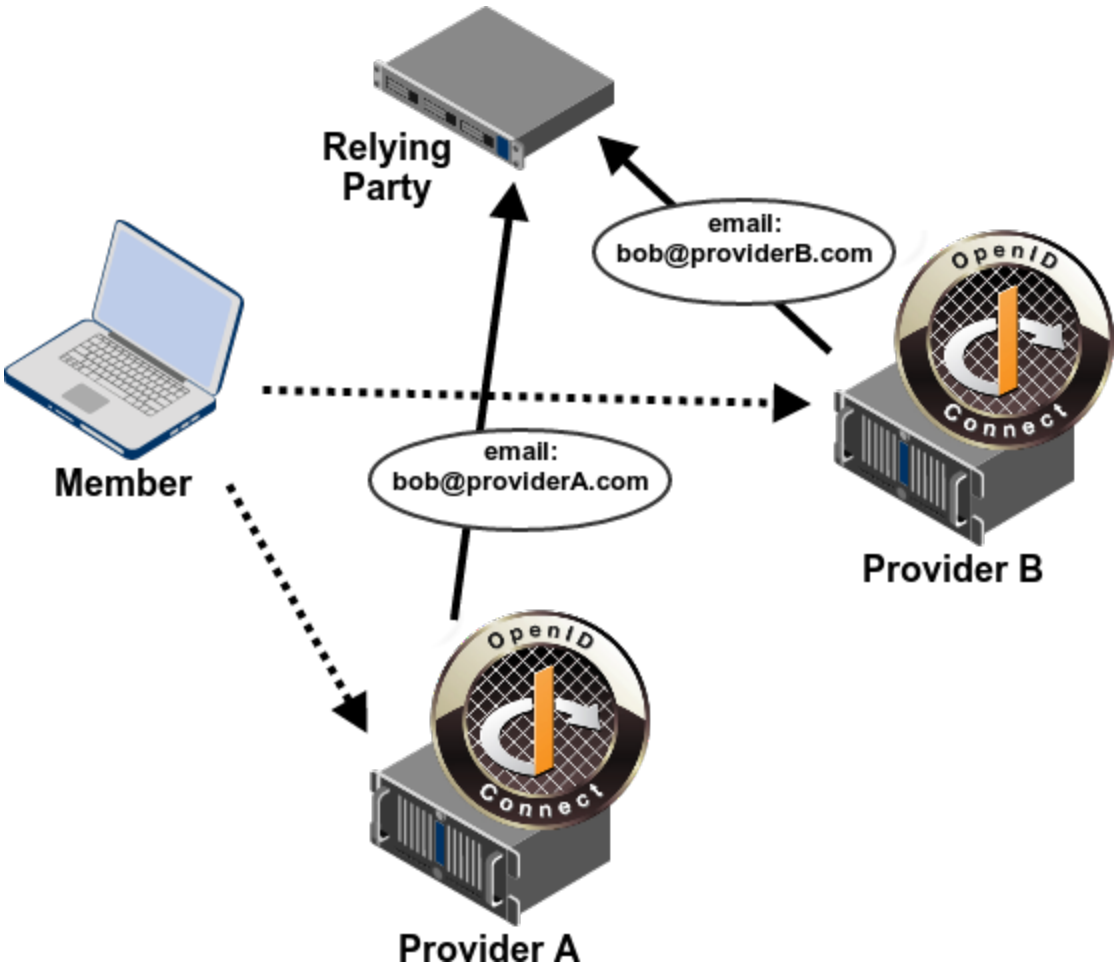
Federation expands on the concept of SSO by transferring the ownership of an identity to the ***person***. The identity of a person can travel between Service Providers (Relying Parties in OpenID Connect terms) or different identity providers.

To be clear, the OpenID Connect standard **does not define any specifics for performing federation**. OpenID Connect simply gives you the tools to accomplish the goal. Since each use case for federation is very implementation specific, an enforced standard method would be prohibitive to solving the problem.

There are two main methods of implementing federation: **account linking**, and **back-channel trust relationships**.

Account Linking

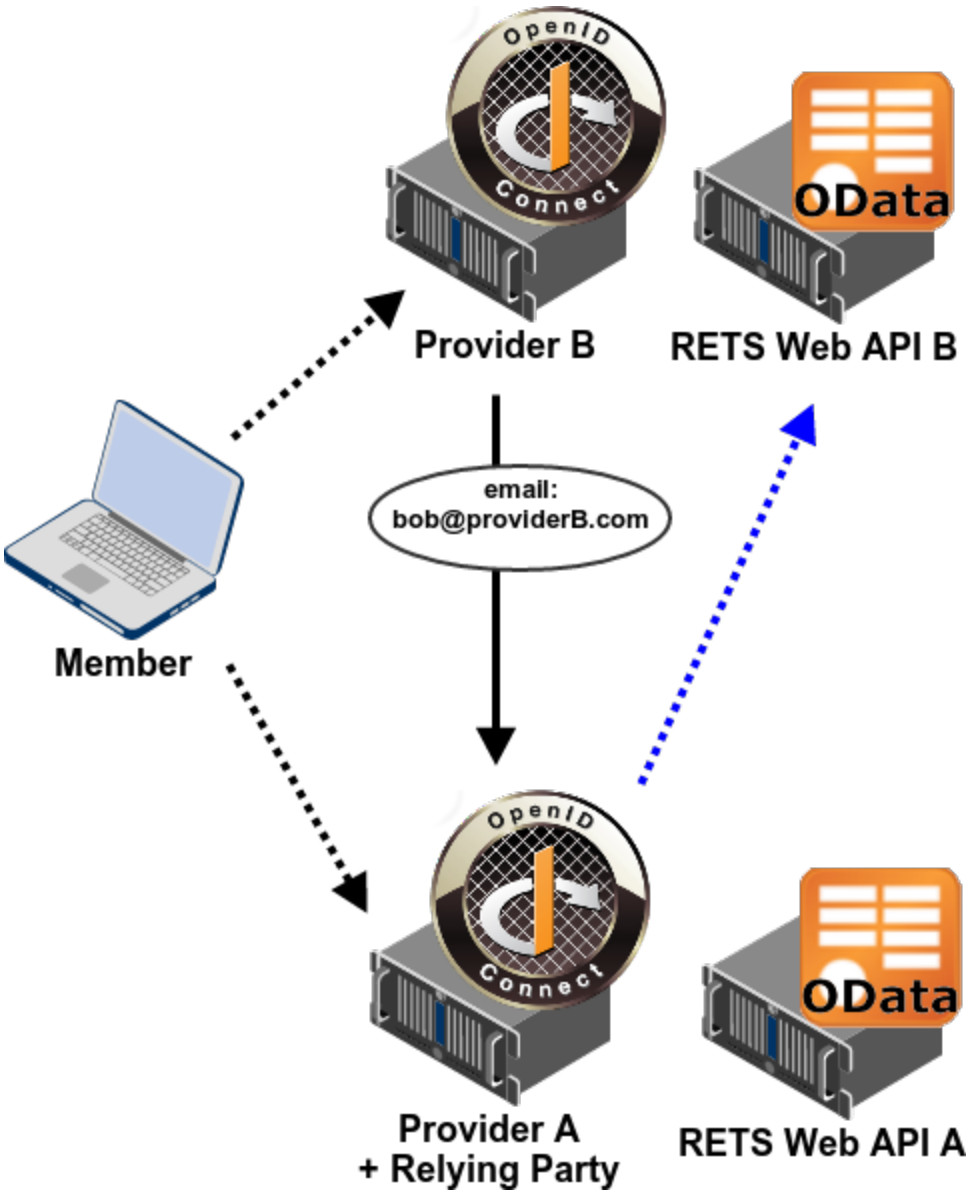
Account linking uses the concept that if a Member has logged in from Provider A, and then logged in from Provider B, the Relying Party knows that both of those accounts represent the same *person*. Subsequent logins from either Provider will land on the commonly linked account at the Relying Party.



The identifying piece of information for a user can be anything unique on that Provider. NRDS ID, username, Agent ID, or email address. Or a combination of all. If a member Bob logs into a Relying Party website using his Provider A account, his email address defined at Provider A will be a Claim in the ID Token. If Bob wants to link his account with Provider B, he proceeds with a second login flow, and the Relying Party now has two authentic ID Tokens for the same person. Bob can now log in at the Relying Party with either Provider in the future. If both Providers are also a RETS API Server, the Relying Party now has an access token to retrieve MLS Data at both Providers. (Reference use case 6.1.4 SP to SP/IdP Transparent, recurring "on behalf of" authorization)

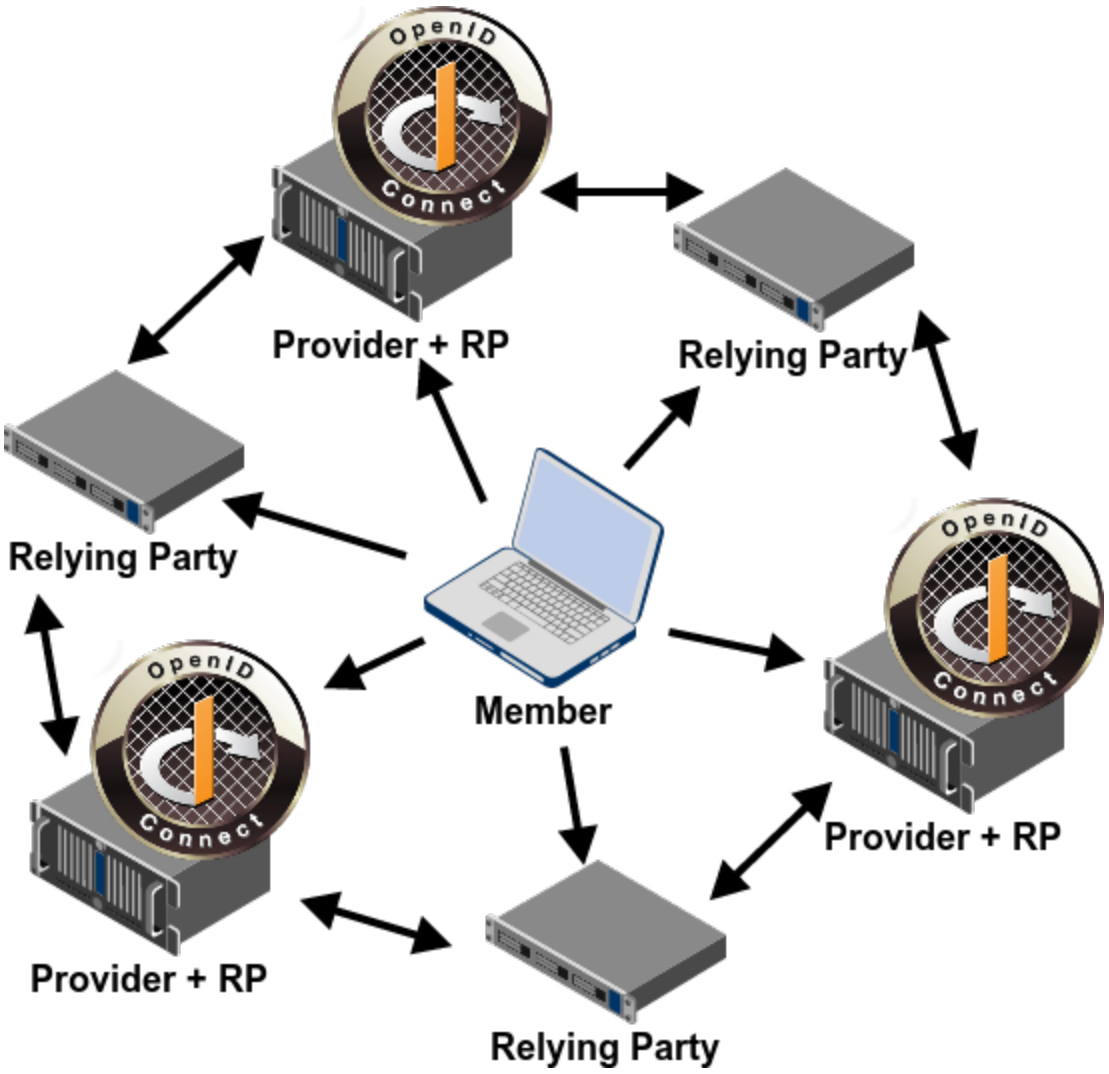
The advantage to this approach is that it does not require any modifications on behalf of the RETS Server Provider. The Relying Party is in control of this model and has the freedom to adjust the architecture to suit their needs. Additionally, Bob is now in control of his data, and may revoke access for the Relying Party to either Provider at any time.

Furthermore, we can extend this concept to Site/MLS vendors. Imagine if a RETS Server Provider also acts as a Relying Party to other RETS Server Providers. Picture this example:



Bob logs in to Provider A, then links his MLS account to another MLS at Provider B. RETS Server Provider A also receives an access token for Provider B's MLS Data. This diagram shows a unidirectional trust relationship for simplicity purposes, but Provider B could also trust Provider A in order to create a bidirectional trust between each other. Add in a RETS Server Provider C and we have a party! COOL!

Adding both of these techniques into a single diagram, we now have a familiar spoke-hub arrangement, except the Member is now at the center of control instead of the identity provider:

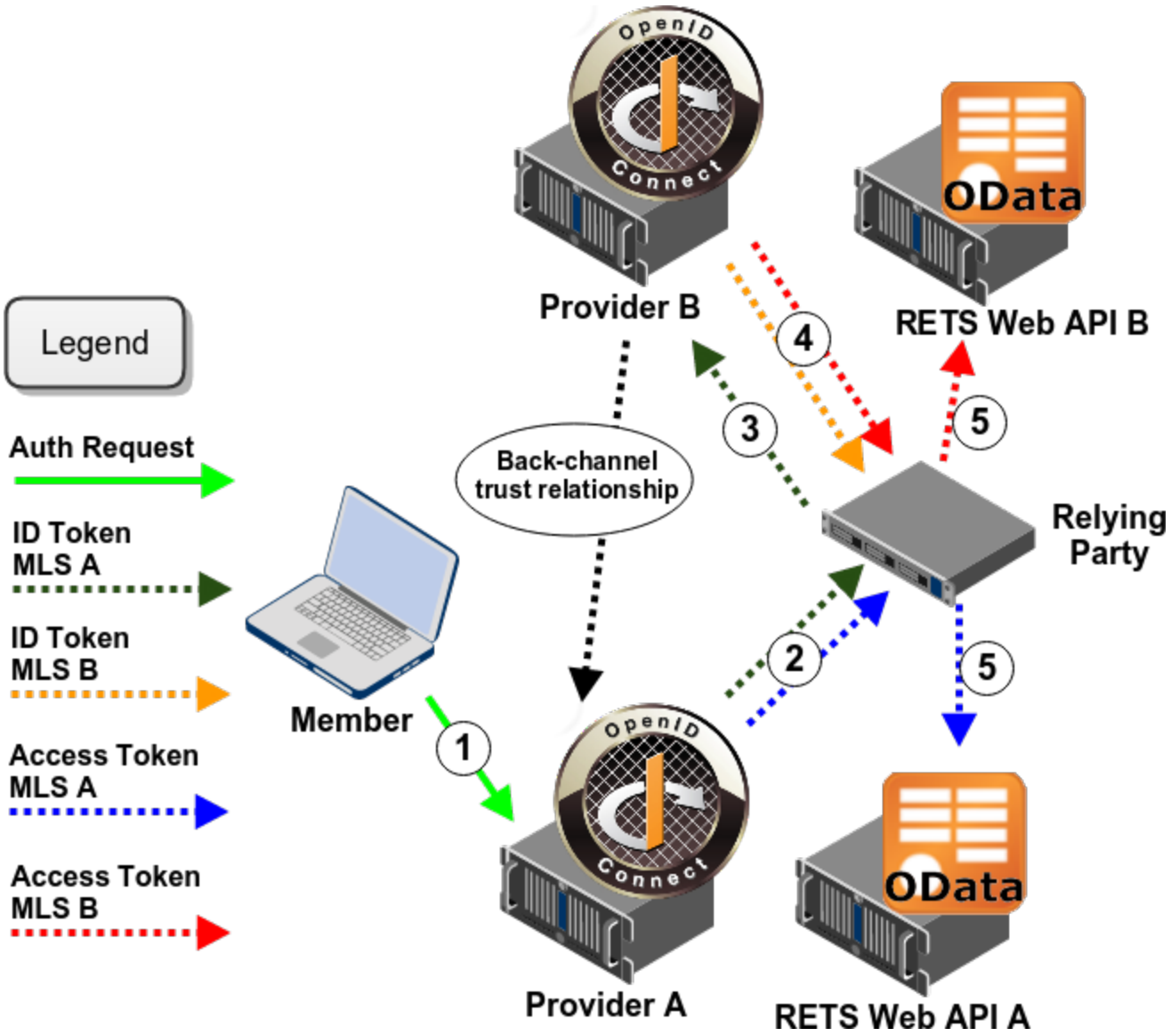


To see an example of account linking in action, read more about Account Chooser and the user experience flowchart it uses. Then try out the Favorite Number app to see it in action.

Back-Channel Trust Relationships

If account linking delegates the federation power to the Relying Party, the back-channel trust shifts that power to the RETS Server Provider. Instead of the Relying Party forming the link, the RETS Server Providers trust each other's ID Tokens at the token exchange endpoint. Amazon Cognito is a popular implementation of this method. After adding a trusted Provider in the AWS IAM Console, the Cognito IdP will exchange ID Tokens from that Provider for access tokens to various Amazon cloud services. (DynamoDB, S3, etc)

Here's a diagram to show how it works:



out-of-band: Provider B sets up a trust relationship with Provider A using a list of trusted client_ids or users from Provider B

1. MLS Member chooses to log in with Provider A at the Relying Party
2. RETS Server Provider A gives an ID Token and Access Token to the Relying Party
3. The Relying Party uses the MLS A's ID Token at Provider B's token exchange endpoint
    a. Provider B compares the ID Token's audience and/or user ID against the list of trusted client_ids or users from Provider A
4. Provider B returns a new ID Token and Access Token for MLS B's Web API
5. The Relying Party can now access both MLS's OData services using the correct Access Token

The level at which Provider B trusts Provider A's ID Tokens can be fine grained, or wide open. Trust everything from Provider A from any Relying Party. Or per Relying Party client_id, or even a specific set of users within a client_id. There are use cases for every possibility.

This diagram shows a unidirectional trust relationship for simplicity. Just as with the account linking method, a bidirectional relationship can be created.

This method does have caveats, and might not be the best solution depending on implementation details:

Ownership

It reduces the Member's ownership of his or her identity, and provides implicit access to another RETS API Server without consent. However, this is typically how SSO relationships have been set up in the past, and it could be beneficial depending on the use case.

client_id lists

Another caveat is that this exchange of ID Tokens requires that Provider B must maintain a database of trusted client_ids and/or users from

Provider A. (That is, if fine grained access control is desired) This access control list and the means to transfer the data might become a tedious task, and is out of the scope of OpenID Connect and RESO Web API Security.

Non-standard

If account linking is slightly out of the scope of OpenID Connect, back-channel trusts are the MacGyver and duct tape scope. Many websites in the world use account linking, and Amazon is the only example of this federation method. Additionally, the format of the request parameters at the token exchange endpoint are also non-standard and might surface interoperability issues between Providers.

Requires some work from the Relying Party

While account linking requires no work on behalf of the Provider, back-channel trusts require work by both entities. The Relying Party must also know which providers trust each other, and where ID Tokens can be exchanged. (Unless they brute force the combinations)

> To see an example of Amazon Cognito in action, check out TestCMAVendor.com, and the source code behind it. This was written as a live example of the guide Building an App Using Amazon Cognito and an OpenID Connect Identity Provider.

6.2 Resources and Links

- 6.2.1 Help Guides and Introductions
- 6.2.2 Library Demos and Examples
- 6.2.3 Identity-as-a-Service Providers

6.2.1 Help Guides and Introductions

- OpenID Connect Explained
- API Security: Deep Dive into OAuth and OpenID Connect
- On ID Tokens
- OpenID Foundation General mailing list

OpenID Connect Specifications

While it's not easiest to jump right into the specs, these documents are the go-to guide for a reference if you need to write your own library:

http://openid.net/developers/specs/

- OpenID Connect Core
- OpenID Connect Discovery
- OpenID Connect Dynamic Registration
- OpenID Connect Session Management

6.2.2 Library Demos and Examples

Demos

- TestFormVendor.com – A simple demo for OIDC, by Cal Heldenbrand
    - Source code is available on GitHub
    - Contact Cal for credentials – cal@fbsdata.com

- TestCMAVendor.com – A federated demo for OIDC using Amazon Cognito, by Cal Heldenbrand
    - Source code is available on GitHub
- Google OAuth2/OIDC Playground
- OpenID Connect Demo with Google
- Amazon Web Identity Federation Playground
- Account Chooser – a simple way to perform federated linking on your website
    - Google Identity Toolkit – based on account chooser, tailored to Google Plus and friends.
    - User experience flowchart for Account Chooser

Examples

- Making a Javascript OpenID Connect Client in 4 steps
    - Source Library here
- Javascript Cookbook for OpenID Connect Public Client

6.2.3 Identity-as-a-Service Providers

If you would like to cloudify OpenID Connect, here are a list of IDaaS providers that support OpenID Connect:

- Amazon Cognito
    - Building an App using Amazon Cognito and an OpenID Connect Identity Provider
    - Use Amazon Cognito in your website for simple AWS authentication
    - Understanding Amazon Cognito Authentication
- Microsoft Azure Active Directory
- Salesforce
    - Digging Deeper into OAuth2
    - Inside OpenID Connect
- Auth0
- CA Technologies Mobile API Gateway (documentation)
- Axway API Gateway (demo)
- WSO2 Identity Server (cloud service)