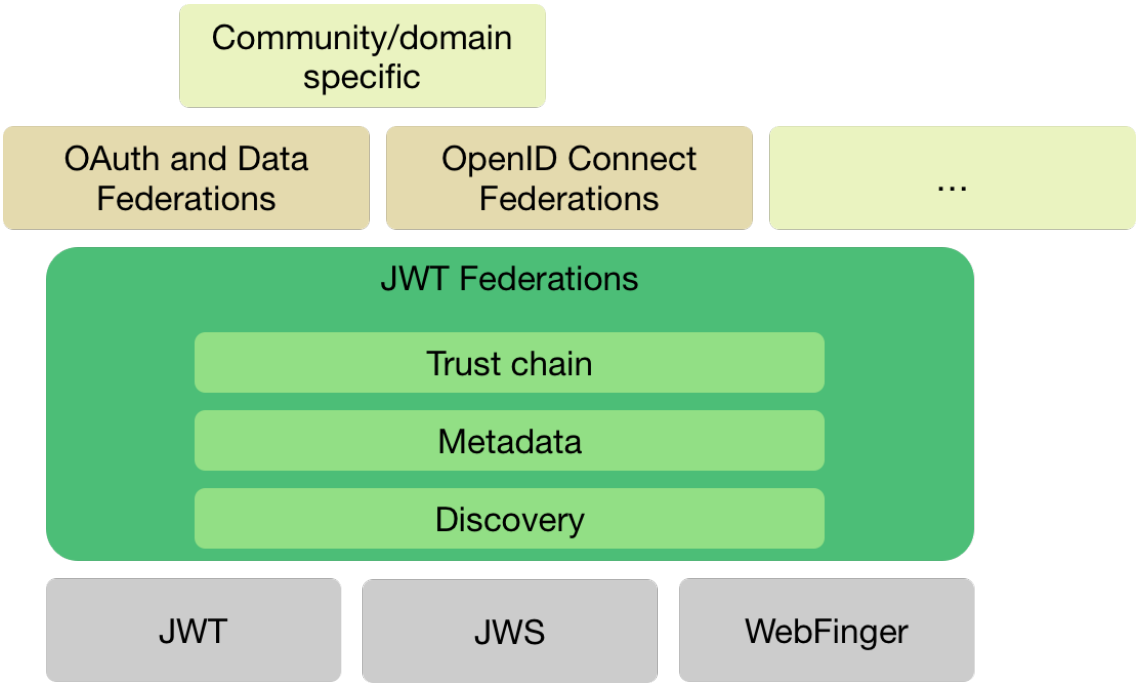
JWT Federations

JWT Federations is a basic layer of trust infrastructure that allows entities to discover each other, distribute public keys and resolve trusted metadata. JWT Federations is making use of signed JWT documents to contain this information, but is protocol agnostic, and can be used to build any kind of federations between entities that can communicate using asymmetric crypto and can be associated with URLs and hostnames.

JWT Federations can be layered and extended, an example of OpenID Connect federations built on this principles is described in this document.

This is a draft. Focus on the principle and design, and more work need to detail all the needed entity claims and more...

This is an alternative approach to the OpenID Connect Federation specification now being standardized with in OpenID Foundation.



Entities generates its own key pairs

All entities generates its own key pairs. The public key may look like this:

```
{
  "kty": "RSA",
  "kid": "https://serviceprovider.org/application#1",
  "n": "l7rt1yRvbiOKg8XeP_ICo0yDif-
kOLWkUL5FAWKVWhWWAdnN2o1t_otuBX1xLeItE24he4qGHBzh2PQ4SRqau6ZVzx4-
aJFzGZSbw6SswVXPlFR5dRkJMn4wxFOOVsSUnltO4K27X2Pf-
gwlLFdH4q4QTNU5U8ijr76BnuUThdBYrxf2UQT7DDz6cPHaRdOUbuj_Ids9CmV6HyzdIFOfBx7DKS8o2fqH9Fa6-
PKdMtDJiZ1KfjgstiNB04JAbQ1RI9Bl-No6NTUcZbD7Q0JF8iqY3Hogo9J_mL-
SgQFGgwAoxQKoNeLk7uLHc69yIlyBJegrVkmHUKehIp3OZ5CW9w",
  "e": "AQAB"
}
```

Copy

The client may re-use its key-pair across different federations, and in communication with multiple parties. JWT federations enforces the use of asymmetric keys.

Entity identifiers

All entities will choose an identifier to use. For OIDC clients, this identifier will be the `client_id`. For OIDC providers, this identifier will be the provider ID. For protected APIs, the identifier should be the *base URL* of the API. For all entities the identifier MUST be an URL including the hostname under control of the entity.

End user discovery

Given an identifier of a user or some hints of the user, resolve to entities associated with the user. Typically one could resolve the identifier of the entity, and then perform a separate entity discovery process to resolve the needed metadata for the entity.

An example, could be resolving an OpenID Connect Provider associated with an end user e-mail address, using OpenID Connect Discovery 1.0 (http://openid.net/specs/openid-connect-discovery-1_0.html)

Entity Discovery

Entities can issue statements about other entities. All entities MUST be able to issue statements about themselves. These statements contains technical and non-technical metadata and trust information.

Trust is established when an entity signs a statement about another entity, including the public key of the target entity. Trust can be chained, a root trust anchor can issue a statement about an intermediate entity, which in turn issue a statement about an OAuth provider, which also issues a statement about itself.

To discovery an entity, one MUST know the entity identifier. The entity is discovered using WebFinger [RFC7033] (https://tools.ietf.org/html/rfc7033), with the `rel` value:

`http://oauth.net/specs/federation/1.0/entity`

The provider performs normalization rules to the entity identifier to determine the hostname. The client_id URL will be the `resource` .

In example the entity identifier `https://serviceprovider.org/application` may result in the following WebFinger request:

```
GET /.well-known/webfinger?
  resource=https%3A%2F%2Fserviceprovider.org%2Fapplication&
  rel=http%3A%2F%2Foauth.net%2Fspecs%2Ffederation%2F1.0%2Fentity
  HTTP/1.1
Host: serviceprovider.org
```

Copy Copy

The response should include a link to a signed entity statement.

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: application/jrd+json

{
  "subject" : "https://serviceprovider.org/application",
  "links" :
  [
    {
      "rel" : "http://oauth.net/specs/federation/1.0/entity",
      "href" : "https://serviceprovider.org/discover?id=1"
    }
  ]
}
```

Copy Copy

Next, a HTTP GET request should be directed at the `href` reference pointing at a signed entity statement. The response will be a signed JWT entity statement document.

```
GET /discover?id=1 HTTP/1.1
Host: serviceprovider.org

200 OK
Last-Modified: Wed, 22 Jul 2018 19:15:56 GMT
Content-Type: application/jwt

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJodHRwczovL3NlcnZpY2Vwcm92aWRlci5vcmcvYXBwbGljYXRpb24iLCJ
```

Copy Copy

The correct MIME type is `application/jwt` .

Entity statements, metadata and trust

A entity statement document is always a JWT following the following rules.

An entity statement is issued by the issuer entity. The issued entity identifier MUST be included as the `iss` claim. An entity statement is always describing a subject, identified by the `sub` claim.

In a trust chain, there is always a leaf node, which is the entity in which an seeking entity is about to establish trust with. The seeking entity has established a local trust root configuration, which should be represented as a list of decoded entity metadata statements, representing the which external entities that is trusted.

A trust chain may look like this:

```
Local trust configuration -> External Trust root -> Intermediate authority -> Leaf node
```

Copy Copy

A leaf node that issues a statement about itself will have both the `iss` and `sub` claims set to the same.

The entity statement should contain a list of JSON Web key sets (JKWs) (https://tools.ietf.org/html/rfc7517#section-3). The keys are represented using RFC7517 and included in a JSON array in the `jwks` claim. The `jwks` claim MUST be included in all entity statements, except a leaf nodes statement about itself, where it is optional. If the claim is left out, the consumer should consider the keys of leaf node to be those specified by the successor in the trust chain.

The `leafNode` claim MUST be included and set to `true` when the subject is a leaf node. A leaf node cannot issue statements about other entities. If the `leafNode` claim is left out it is considered to be `false` .

The claim `subTypes` contains an JSON array including specific entity types that corresponding leaf nodes implements. The `subTypes` array MUST include all or a subset of the values included in the trust parent.

```
{
    "iss": "https://feide.no",
    "sub": "https://serviceprovider.org/application",
    "subTypes": ["samlProvider", "openidProvider"],
    "metadata": {
        "openidProvider": {

        }
    },
    "jwks": [
        {
            "kty": "EC",
            "crv": "P-256",
            "x":   "f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
            "y":   "x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0",
            "kid": "v7"
        }
    ]
}
```

Copy
Copy

The `metadata` claim is type specific, and includes a JSON object, where the property matches the corresponding `subType` . The metadata properties are not reflecting the subject, but always the leaf node.

Entity type specifications

Each entity type needs its own specification that defines:

- the type identifier, such as `openidProvider`
- the syntax and semantics of the metadata document
- specific rules on how to match the entity identifier into the relevant protocol. In example the `openidClient` specification should define that the entity `sub` MUST match the `metadata.openidClient.client_id` .
- specific rules on how the public keys are used within the relevant protocol.

All metadata specifications should be extensible, allowing for a community to define additional properties and rules for handling them.

Discovering the trust chain

The seeking entity starts by knowing the identifier of the target entity, and needs to resolve a possible trust chain between the target entity and a locally configured trust root.

The seeking entity uses entity discovery (described above) to get the self issued entity statement of the target entity, which should be a leaf node and has signed its own statement.

The entity statement should contain a `authorityHints` claim with a JSON array including other entities that may issue statements about this entity. This claim is needed only for discovery, and is not considered in the security validation. The seeking entity may configure and use other means of resolving entities that can be contacted in order to complete the trust chain.

The seeking entity should recursively resolve all `authorityHints` until it find a match from an asserting entity with the `sub` of the locally configured trust root, or it has resolved all possible paths without the ability to establish trust.

The seeking entity may find multiple possible trust paths between local trust root and the target entity. It is left to the client to choose which trust path to use. One could choose the first found, allow local configured priority, or choose the resulting metadata giving preferable level of trusted functionality.

Validating the trust chain

As soon as the seeking entity has resolved all the components in the trust chain, it needs to start with the configured local trust root, and validate the first entity root against the local configuration. This means validating the locally configured `sub` , `subTypes` and `jwks` values against the JWS signature, the `subTypes` and the `iss` claim of the first statement. Furthermore subsequent validation is performed in all steps including the last self issued statement from the leaf node, where the `iss` and `sub` matches. The result of the trust validation would be:

- the identifier of the trusted leaf node
- the entity type(s) of the trusted leaf node
- the resolved calculated metadata of the trusted leaf node (see next section)

Resolving the metadata of leaf nodes

All components in the trust chain (including the local trust configuration) may contribute content to the leaf node metadata. Content closest to the local trust root has higher priority and will override properties from more distant content.

To resolve the metadata object, start with the object issued by the leaf node it self.

For each parent entity statement issuers, traverse the metadata properties, and merge the values depending on the data type (from the parent entity).

- if the data types does not match, then use the value from the parent
- if a boolean, integer or flat value, override the value from the parent
- if the property is missing from the parent object, then do not change
- if a JSON array, then remove the elements that are not present in the parent (subsection)
- if a JSON object then iterate all children properties with the same rules, resulting in a deep merge.

Here is an example trust chain:

Local configuration. Trusting a federation *eduGAIN*:

```
[
    {
        "sub": "https://edugain.org",
        "subTypes": ["openidProvider", "openidClient"],
        "metadata": {
        }
    }
]
```

Intermediate trusted party (eduGAIN asserts the Norwegian federation Feide as trusted):

```
{
    "iss": "https://edugain.org",
    "sub": "https://feide.no",
    "subTypes": ["openidProvider", "openidClient"],
    "metadata": {
        "openidProvider": {
            "userTLDs": ["no"],
            "id_token_signing_alg_values_supported": ["RS384", "RS512", "ES512"],
        }
    }
}
```

Organization as a trusted party (Feide asserts statement about an university that can provide both OpenID providers and clients)

```
{
    "iss": "https://feide.no",
    "sub": "https://ntnu.no",
    "subTypes": ["openidProvider", "openidClient"],
    "metadata": {
        "openidProvider": {
            "issuer": "https://ntnu.no",
            "organization": "NTNU",
            "legal_contact": "info@ntnu.no",
            "userRealms": ["ntnu.no", "hials.no"],
            "id_token_signing_alg_values_supported": ["RS256", "RS384", "RS512"],
        },
        "openidClient": {
            "organization": "NTNU",
            "legal_contact": "info@ntnu.no",
            "grant_types_supported": ["authorization_code", "implicit"],
            "scopes": ["openid", "profile", "email", "phone"]
        }
    }
}
```

Self issued statement from leaf node (NTNU OpenID Provider):

```
{
    "iss": "https://ntnu.no",
    "sub": "https://ntnu.no",
    "authorityHints": ["https://feide.no", "https://swamid.se"],
    "subTypes": ["openidProvider"],
    "leafNode": true,
    "metadata": {
        "openidProvider": {
            "issuer": "https://ntnu.no",
            "id_token_signing_alg_values_supported": ["RS256", "RS512"],
            "authorization_endpoint": "https://openid.ntnu.no/authorization",
            "technical_contact": "tech-support@ntnu.no"
        }
    }
}
```

Service as a trusted party (NTNU asserts statement about the service provider Blackboard)

```
{
    "iss": "https://ntnu.no",
    "sub": "https://blackboard.ntnu.no",
    "subTypes": ["openidClient"],
    "leafNode": true,
    "metadata": {
        "openidClient": {
            "client_id": "https://blackboard.ntnu.no",
            "client_name": "NTNU Blackboard",
            "application_type": "web",
            "technical_contact": "tech-support@ntnu.no",
            "grant_types_supported": ["authorization_code"],
            "redirect_uri_prefixes": ["https://blackboard.ntnu.no/"],
            "scopes": ["openid", "email"]
        }
    }
}
```

Self issued statement from leaf node (Blackboard service)

```
{
    "iss": "https://blackboard.ntnu.no",
    "sub": "https://blackboard.ntnu.no",
    "authorityHints": ["https://ntnu.no"],
    "client_name": "Ignored client name - cannot be overridden",
    "subTypes": ["openidClient"],
    "metadata": {
        "openidClient": {
            "redirect_uris": ["https://blackboard.ntnu.no/callback"]
        }
    }
}
```

These set of statements, can be resolved to two leaf node entities.

One OpenID Provider with identifier `https://ntnu.no` :

```
{
    "issuer": "https://ntnu.no",
    "organization": "NTNU",
    "legal_contact": "info@ntnu.no",
    "technical_contact": "tech-support@ntnu.no"
    "userTLDs": ["no"],
    "userRealms": ["ntnu.no", "hials.no"],
    "id_token_signing_alg_values_supported": ["RS512"],
    "authorization_endpoint": "https://openid.ntnu.no/authorization"
}
```

and one service provider with idenfifier `https://blackboard.ntnu.no` :

```
{
    "organization": "NTNU",
    "client_id": "https://blackboard.ntnu.no",
    "client_name": "NTNU Blackboard",
    "grant_types_supported": ["authorization_code"],
    "technical_contact": "tech-support@ntnu.no",
    "legal_contact": "info@ntnu.no",
    "application_type": "web",
    "grant_types_supported": ["authorization_code", "implicit"],
    "scopes": ["openid", "email"],
    "redirect_uri_prefixes": ["https://blackboard.ntnu.no/"],
    "redirect_uris": ["https://blackboard.ntnu.no/callback"]
}
```

The public key(s) of the leaf nodes can be extracted from the self issued leaf node statements.

OpenID Connect Federations

Federations of OpenID Connect providers and clients can be built using JWT Federations basic, with the addition of entity types for OpenID clients and providers.

In order to support OpenID Connect federations there are some requirements for clients and providers:

- Support for resolving and validating metadata needs to be built-in or performed in a side-process.
- Client SHOULD send a signed authentication request as specified in OpenID Connect Core section 6 (http://openid.net/specs/openid-connect-core-1_0.html#JWTRequests). If signed it MUST sign the request with a key matching the public key referred to in the `jwks` claim in the self-issued entity statement.
- The provider will use the `client_id` to dynamically fetch the relevant entity statements in order to obtain trust and metadata as

soon as it gets the signed request.
- If client is using the token endpoint, it MUST authenticate the request by including the `private_key_jwt` parameter described in OpenID Connect Core Section 9 (http://openid.net/specs/openid-connect-core-1_0.html#ClientAuthentication).

It is up to a specific entity if it also will allow interaction self asserted entities that it does not have a trust relationship with.

It is possible to build federations where both client and providers is validated by a trusted third party. It is possible to build a federation where a provider is allowing any clients to connect (release of personal information may instead be controlled using user consent as part of the UI interaction). And it is possible to build a federation where a client is allowing any possible provider without pre-established trust (user centric identity). In this last case, the client must ensure non-overlapping value spaces for user identifiers, by in example prefixing the userid locally. All these scenarios can be implemented using JWT Federations.

Use of the implicit flow: We outline two alternative ways to implement implicit flow with OpenID Connect Federations.

Alt 1: Basic security

The client is only authenticated by the link to the `redirect_uri` . For mobile applications, the `redirect_uri` can be a custom url scheme, such as `backboard_app://` . The client does not contain any secrets or key pair. However the OpenID Connect Federations requires that there is a backend server representing the application owner to host a WebFinger discovery service, as well as provide the self issued entity statements on behalf of all installed clients. The self issued statement will then not contain a `jwks` claim, because there is no easy way distribute such key pairs to all the client runtime environments (browsers, mobile phones).

This use case does not allow for encryption of content aimed for the client.

Alt 2: Enhanced security

The application has a common backend for registration of client deployments. All deployments will go through this process:

- During installation, generate a key pair
- Somehow register to the application backend (may or may not be authenticated). Application backend generates and hosts entity statements and separate client_ids for all clients. If client is able to authenticate, in example using EAP-SIM, the application backend may add application specific identifiers in the entity statement in order to track misuse.

In this alternative the client will be able to generate authenticated requests, and receive encrypted idtokens.

TODO: Need to add processing rules for language specific human readable attributes, such as `client_name#nb` .

OpenID Connect Providers

Type identifier `openidProvider`

The metadata object is based upon OpenID Connect Discovery (http://openid.net/specs/openid-connect-discovery-1_0.html) (section 2), with the following additions:

- `organization` : The name of the organization that asserts the identity of the users.
- `userTLDs` : For use of user identifier with realm suffix. Limits the trusted user ID issed to this list of TLDs.
- `userRealms` : For use of user identifier with realm suffix. Limits the trusted user ID to these realms.
- `geo` : The geo location of the organization. This is used to easier locate the nearby identity providers of an end user. The format is a JSON array with objects `[{"lat": 37.7898, "lon": -122.3942}]`

These attributes are examples, one should evaluate which information is needed to specify in a common metadata

Processing rules:

- Validate matching between `userTLDs` and `userRealms`
- Validate that the leaf node statement `sub` matches the metadata `issuer`
- Replace the metadata `jwks_url` and `jwks` claims with the `jwks` claim from the outer entity statement.

OpenID Connect Clients

The metadata object is based upon OpenID Connect Dynamic Client Registration (http://openid.net/specs/openid-connect-registration-1_0.html) (section 2), with the following additions:

- `organization` : The name of the organization that owns the service.
- `scopes` : An JSON array of OAuth scopes that the client is trusted to obtain.
- `claims` : An JSON array of OpenID claims that the client is trusted to receive.
- `redirect_uri_prefixes` : A limitation on which `redirect_uris` this entity is allowed to use.

These attributes are examples, one should evaluate which information is needed to specify in a common metadata

Processing rules:

- Validate that the leaf node statement `sub` matches the metadata `client_id`
- Replace the metadata `jwks_url` and `jwks` claims with the `jwks` claim from the outer entity statement.
- Remove all the `redirect_uris` that does not match the prefixes from `redirect_uri_prefixes`

OAuth and Data Federations

To allow for resource providers that are far less integrated with the authorization servers than we've seen so far, we should make use of JWT Federations and JWT bearer tokens.

We will use JWT bearer tokens with the `client_id` and `scope` claims introduced in OAuth 2.0 Token Exchange (https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-14). The token may be issued by an OpenID Connect Provider or an STS.

Here is an example of such a JWT bearer token:

```
{
    "iss": "https://sts.feide.no",
    "aud": "https://smime-certificate-store.uninett.no",
    "client_id": "https://blackboard.ntnu.no",
    "sub": "andreas@ntnu.no",
    "scope" : "profile certread history"
}
```

Copy
Copy

This example shows a token issued from a *STS service* (`iss`) to the *Blackboard service* (`client_id`) allowing to interact with a resource provider, the *smime-certificate-store* (`aud`), on behalf of the authenticated end user *andreas @ ntnu.no* (`sub`).

Using a JWT bearer token with an resource provider

A resource provider receiving a request with a bearer token like the one, it will use the entity discovery to establish trust with the client. It will off course also verify that the `aud` claim is correct.

The provider may put additional requirements for how the client authenticates itself. It can be sufficient to alone use the bearer token, but the resource provider could also require that the client uses a TLS client certificate that matches the key provided in the `jwks` of the resolved metadata, or a signed HTTP request (using in.e. *draft-ietf-oauth-signed-http-request-03*).

The resource provider typically would depend on a different trust chain than the OpenID provider. If the resource provider is part of a distributed community service, it may host its own trust anchor that keeps track of trusted clients and which clients that has access to which scopes.

In addition to trusting the client, through a trust chain, it also need to trust the entity that have asserted the identity of the end-user. The resource provider may not know the `iss` in advance, and will use entity discovery to obtain the entity statement of the issuer (STS or OpenID Provider).

If the resource provider establishes a trust chain with the JWT bearer token issuer, it will resolve the metadata of the issuer, and validate the signature of the bearer token, and perform some additional validation steps.

Client obtaining bearer token

There is several possible ways for the client to obtain a bearer token.

This section is currently just outlining a few alternative directions, and does not include a complete specification.

(A) Obtaining a bearer token directly from the OpenID Provider

The OpenID provider that authenticates the end user also has the ability to understand and issue delegated tokens to any resource providers.

The OpenID providers need an additional STS functionality to be able to issue the bearer tokens. This could be indicated through the support of the `sts` OAuth scope.

The STS protocol would probably involve presenting the access token issued during OpenID authentication to a STS endpoint and getting back a bearer token according the provided request parameters.

There is two variants of this. Either the STS endpoint could be a front-channel redirect endpoint, allowing end-users to be presented with a proper grant display before the token is issued, or the endpoint to be a simple API endpoint used back-channel in combination with additional request parameters in the OpenID authentication request that allowed the OpenID provider to embed the necessary user information in the existing grant page.

(B) Introducing independent STS components

The draw back of model A is the addition of new STS component at an already well deployed network of OpenID providers. By introducing a completely independent STS component we would re-use all existing infrastructure for user authentication to also build federation of trusted data exchange.

The STS component would need to implement a front-channel endpoint allowing the client to request which resource provider to set as an audience, which scopes to use.

Either the STS component will need to take care of the authentication of the end-user itself, or it will accept an incoming ID token that is targeted to the client.

Unfortunately, the OAuth 2.0 Token Exchange specification (https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-14) is back-channel only, making it unsuited for this use case.

Resource Providers offers its own metadata

A community may choose to distribute resource providers in example across thousands of universities. There is a need for the client to be able to establish a trust relationship that allows it to trust a specific resource provider instance that it has configured, but somehow discovered.

All resource providers MUST have an identifier. It SHOULD be the base URL of the API, and will allow for discovery of the API by using entity discovery.

Type identifier `resourceProvider` .

Metadata example:

```
{
    "organization": "Uninett AS",
    "latestAPIVersion": ["1.5.3"],
    "componentsSupported": ["base", "func1", "func2"],
    "domain": ".uninett.no",
    "technical_contact": "tech@uninett.no",
    "support": "https://api-support.uninett.no",
    "documentation": "https://docs.uninett.no",
    "tos_url": "https://smime-certificate-store.uninett.no/terms-of-use.html",
    "scopesDef": {
        "certread": "Ability to read users public cerificates",
        "history": "View users history of public certificates"
    },
    "authentication": ["bearertoken"],
    "clientAuthorityHints": ["https://feide.no"],
    "userAuthorityHints": ["https://edugain.org", "https://eid.as", "https://facebook.com"]
}
```

Because of the generic nature of resource providers, very few metadata attributes are predefined. Instead each deployment of wide scale distributed resource providers should define their own custom attributes allowing for delegation of trust, creating hierarchies and important assertions about each provider instance.

Resource provider Discovery

Entity discovery may be used for the client to obtain metadata for the resource provider, the OpenID provider and the STS. The other entities may also discover each others metadata using the same approach.

Entity directories

The entity discovery process defines how to discover trust and metadata for a single given entity, where you know the entity identifier in advance.

The OpenID Connect Discovery 1.0 (http://openid.net/specs/openid-connect-discovery-1_0.html) aids a client to resolve an entity identifier of a single provider given some user hints such as an email-address.

However there are many use cases where an entity would like to list all available entities with a set of characteristics within a federation. The recommended approach to this would be to add a new entity type `entityDirectory` that presents an easy and flexible REST API for fetching list of entities with a good query language. This role can be implemented by in example federations that knows all the entities in advance, or it can be an independent service that relies on entities to publish information about themselves. The API to append new entities to such a catalog could be very simple, just sending an entity identifier – and the directory would be able to fetch all the metadata by it self, as well as keep track of the validity, and properly remove the entity when no longer accepted.