Contents

1 Introduction 2
2 Why Implement an Authentication Authorization framework like OpenID Connect? 2
3 OpenID Connect Explained 3
3.1 What is the value of the Identity Layer 3
3.2 Flows in OpenID Connect 5
3.3 Roles used by OpenID Connect 5
3.4 The tokens used by OpenID Connect 6
3.4.1 Sending ID Tokens By Reference 8
3.5 Clients in Open ID Connect 9
3.6 Specifications 11
3.7 Authentication 12
3.8 Token Endpoint 13
3.9 Userinfo endpoint 14
4 Implementation example 15
5 Test Setup 15
5.1 Use Case 1: Multiple DD-Nodes with browser front-end 16
5.1.1 Authentication Steps: 16
5.1.2 Example Requests 16
5.2 Use Case 2: Multiple DD-Nodes with a DD Web application 17
5.2.1 Authentication Steps: 18
5.2.2 Example Requests 19
5.3 Use Case 3: Query DD-Node without user input 20
5.3.1 OpenID Refresh Token 20
5.3.2 OAuth2 Authorization 20
5.3.3 Authentication Steps (OpenID): *One-time steps* 20
5.3.4 Example Requests 21
5.3.5 Authentication Steps (OAuth2): 23
5.4 Validating ID_TOKEN 23

1 Introduction

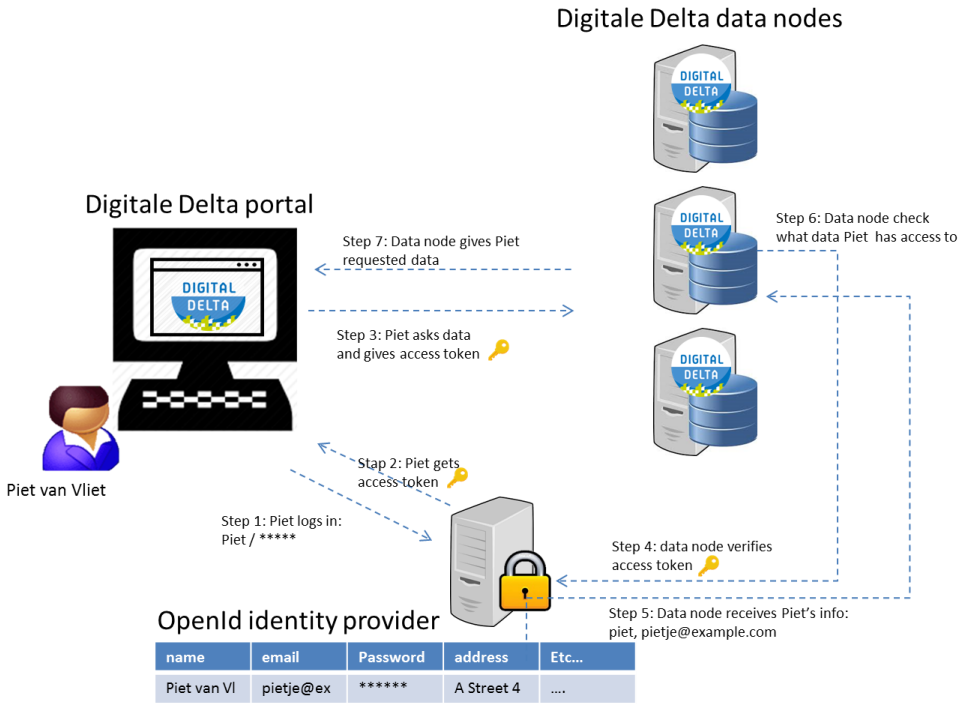
This document describes the framework OpenID Connect, what it is and how it works. The following subjects will be described:

- The roles used by OpenID Connect;
- How to connect your application with an OpenID Connect instance.
- Which tokens are available and what they mean.
- How the tokens are sent over the network.

2 Why Implement an Authentication Authorization framework like OpenID Connect?

Authentication frameworks like OpenID Connect and OAuth 2.0 are implemented because, users do not want to create accounts for every single application they want to use. Users need to create so many accounts and remember so many passwords it is not doable any longer. That is why twitter created OAuth 1.0 which made it possible to log in on an application with your username and password of twitter. This way the application didn't know your username and password, because you login at twitters authentication server and you didn't need to remember your password and username for that given application.

The figurer below briefly shows the security setup we wish to achieve with Digitale Delta (DD).



In the next chapters we will give a more in-depth explanation how OAuth 2.0 and OpenID Connect does this.

3 OpenID Connect Explained

3.1 What is the value of the Identity Layer

OpenID Connect is an Authentication framework. Its core implementation is the same as OAuth 2.0, but it contains an extra layer on top of the known framework OAuth 2.0. This extra layer contains the Identity Layer.

The basic flow of OAuth 2.0 is shown in the following figure: Fig. 1

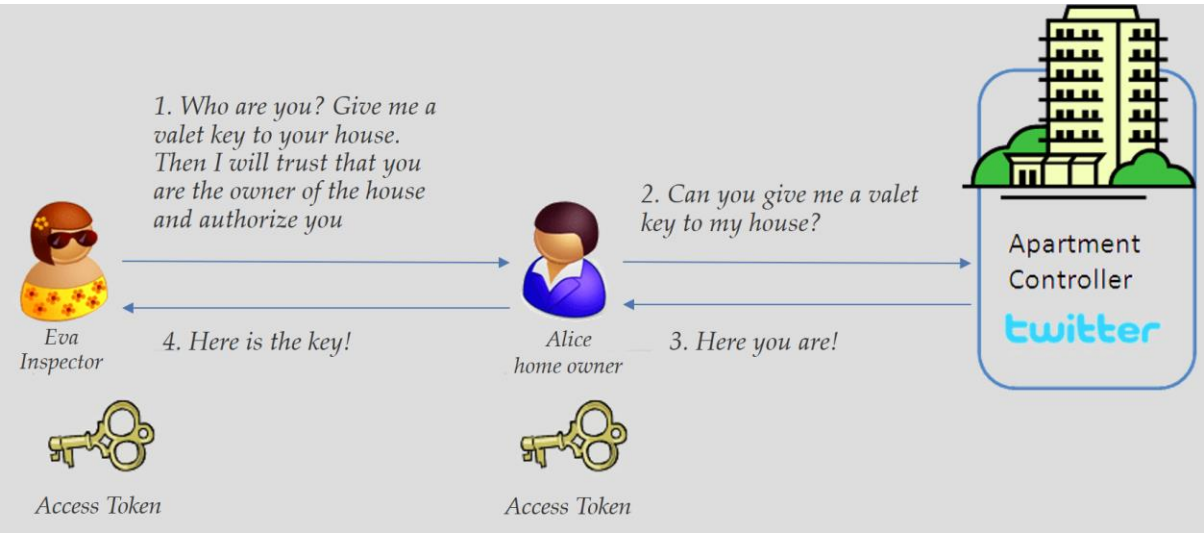


Fig. 1. The simplified flow of OAuth 2.0.

This figure shows that with only the access token which is given with OAuth 2.0. The inspector or the application doesn't know who gave the key. It only knows that the key belongs to Alice and therefore will authorize the user. But Eva or the application will never be sure that it really is Alice who gave the key.

Fig. 2, shown below, shows the simplified flow of OpenID Connect. With OpenID Connect Eva or the application does not only request a key it requests the key (access token) and a formal signed contract with name, and email or other requested claims (ID token). With this signed contract Eva can validate if the person who gave the key is also the real owner of the key and therefor perform Authentication.

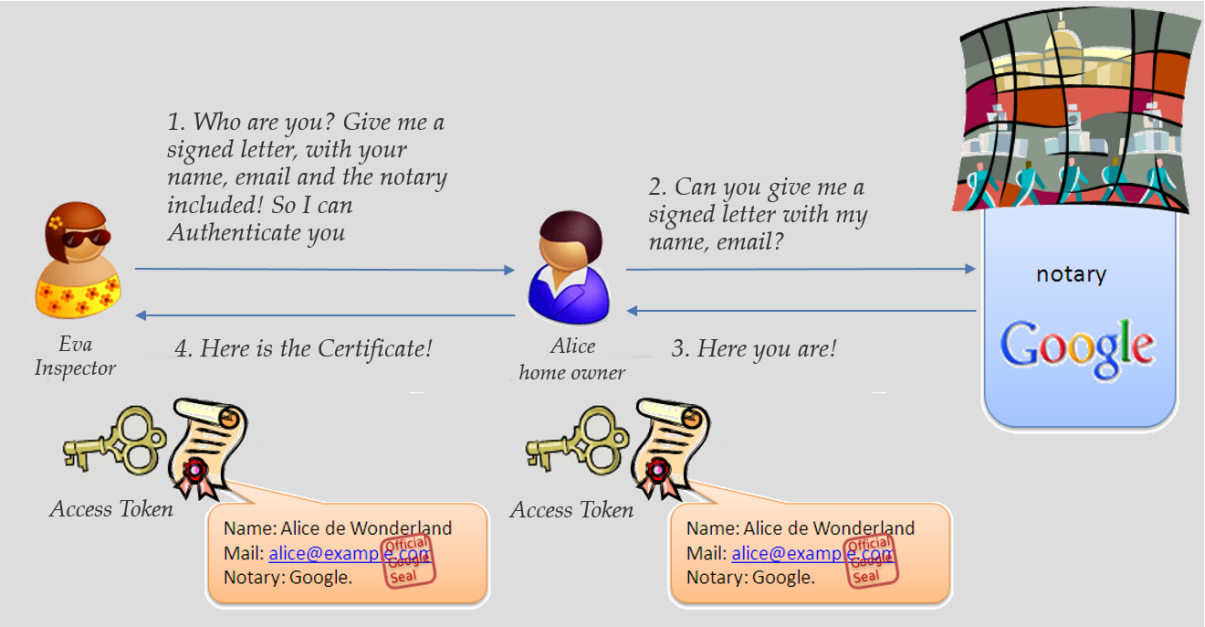


Fig. 2. The simplified flow of OpenID Connect

Note! The information about the user that is returned by the identity provider depends on what the user (Eva) agrees to share on sign in. For the Digitale Delta (DD) we require that users share their email address with the data nodes.

3.2 Flows in OpenID Connect

There are a few different flows to request the tokens in OpenID Connect. Fig. 3 shows when which flow should be used. If a server without user interaction is requesting data via OpenID, the Client Credentials grant should be used. If a user is logging into a DD-web portal via a web application that makes use of a backend system, the Authorization Grant flow or Hybrid flow should be used. Because the tokens are then requested and processed through the backend instead of the front end. If a user is logging into a DD-web portal that does not incorporate a backend system, the Implicit Grant flow should be used. If the user logs in directly to a DD web portal using a user name and password then the Password Grant flow is being used. If the application is a native app and first party then the password Grant flow is valid otherwise the Hybrid flow should be used.

"A user agent based application is a public client in which the client code is downloaded from a web server and executes within a browser"

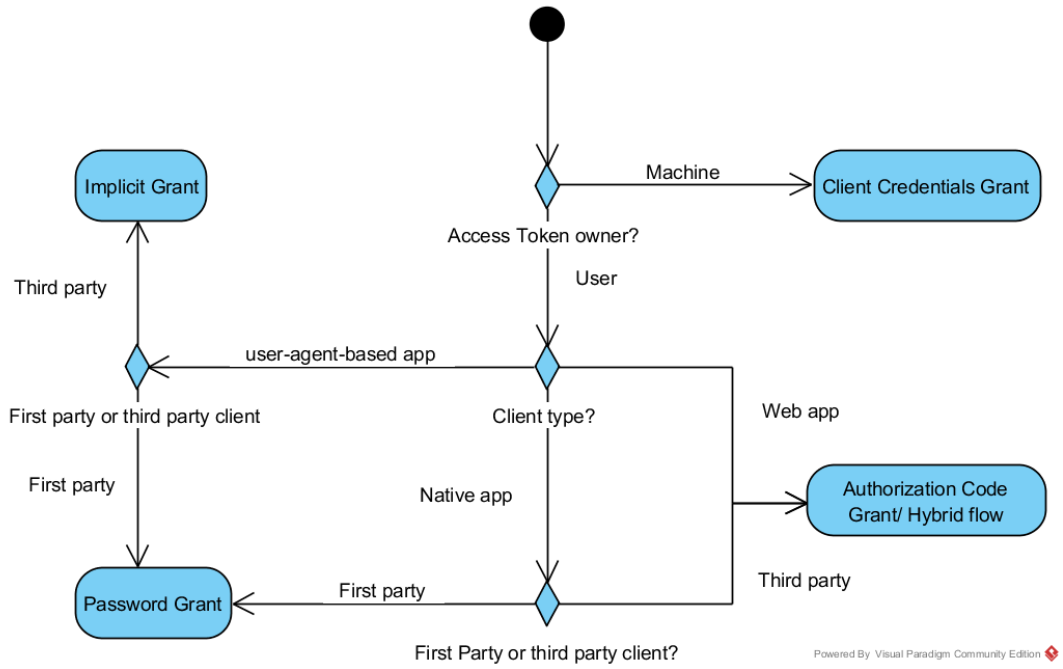


Fig. 3: A flow diagram on which OpenID Connect flow is the correct in a given situation.

3.3 Roles used by OpenID Connect

OpenID Connect has four roles defined:

- The Resource Owner: Organization that owns the data hosted in a DD Node
- The Resource Server: A DD Node
- The Authentication Server: Identity provider server
- The Client: the user or machine requesting data from a DD-Node

The following chapters will describe them in greater detail.

The Resource Owner
The resource owner is the entity that owns the data being hosted in a Resource Server. The Resource Owner defines what data a Client can access. The resources can be:

- Data (photos, documents, contacts).
- Services (posting, collecting data).

The Resource Server
This is the server that hosts the data owned by the Resource Owner. The Resource Server must be able to accept and validate access tokens. With the access token the Resource Server asks the Authorization Server to return information about the Client. When the Resource Server know who the client is, it decides what information can be returned.

The Authorization Server
The authorization server is what the Client interacts with when requesting an access token. The server shows the OpenID Connect login prompt, where the client can select what personal information can be shared with the Resource Servers. The Authorization Server is provides Clients with access tokens and Resource Servers with client information. The authentication step usually uses two endpoint URLs. One for login requests and the other for obtaining the access token. The two links are mostly something like:
HTTPS://example.org/authorize
HTTPS://example.org/token

The Client
The Client is the entity (person or machine) that wants to receive data from the Resource Server. Before the client can access resource data, authentication is required. For authentication the client interacts with the Authorization Server to obtain an access token. This access token is then passed to the Resource Server. With the access token the Resource Server can retrieve information about the Client from the Authorization Server. It is up to the Resource Server to decide what data a Client can retrieve.

3.4 The tokens used by OpenID Connect

OpenID Connect is an extra implementation on top of OAuth 2.0. The limitation of OAuth 2.0 is that the application itself does not know which person is logged in because the user has logged in at the Authentication provider and the application has only received an access token, to verify that the user, (which is unknown to the application), is authorized to login.
The problem with this is that when the application needs data from an external application, the user needs to authenticate itself again at the authentication provider of that application. This is unwanted behaviour. To solve this problem the application needs to know who is using their application, but without the user explicitly logging in on the application itself. This is possible with OpenID Connect. With OpenID Connect the application does not only receive an access token, but it also receives an ID token. The ID token contains the information of the user in a set of claims. These claims are defined beforehand and can contain the birthday, first-name, E-mail, last-name, the given OpenID Provider, the requested application and more.

The difference between the ID token, the access token and the refresh token can be found in Fig. 4. The access token tells nothing about a given user. It only gives the application and the resource server the information that the application is authorized to receive the information. It has for security reasons a fast expiration date of a few minutes. To make OpenID Connect and OAuth 2.0 more user friendly a refresh token can be given to an application. With this refresh token the application can

request a new access token when their old access token is expired without the user needing the reauthenticate. It is also possible to set a timestamp and an expiration time on the refresh token, so that user need to renter their user credentials eventually.

Fig. 4: Difference between ID Token, Access token and refresh token.

The ID token is represented as a JSON Web Token (JWT). How the JWT is defined can be found in Fig. 5 the headers contain the protocol used, in this instance JWT and the used hash algorithm. The body contains the information which was requested in the predefined scopes: like email, first name and birthday, it also contains a few standard parameters, like iss, sub, aud, exp, iat. The checksum is for validation purposes. All the data in the JWT is used to create the checksum. With this the OpenID Provider, can check if the data in the JWT Is changed. When the JWT is created, it is finally signed by the OpenID Provider which signs it by making use of the JSON Web Signature (JWS) with its own private key. This is to make it possible for the resource server to validate the ID token that comes from the given OpenID Provider. After it will be encrypted with JSON Web Encryption (JWE) to make it more secure.

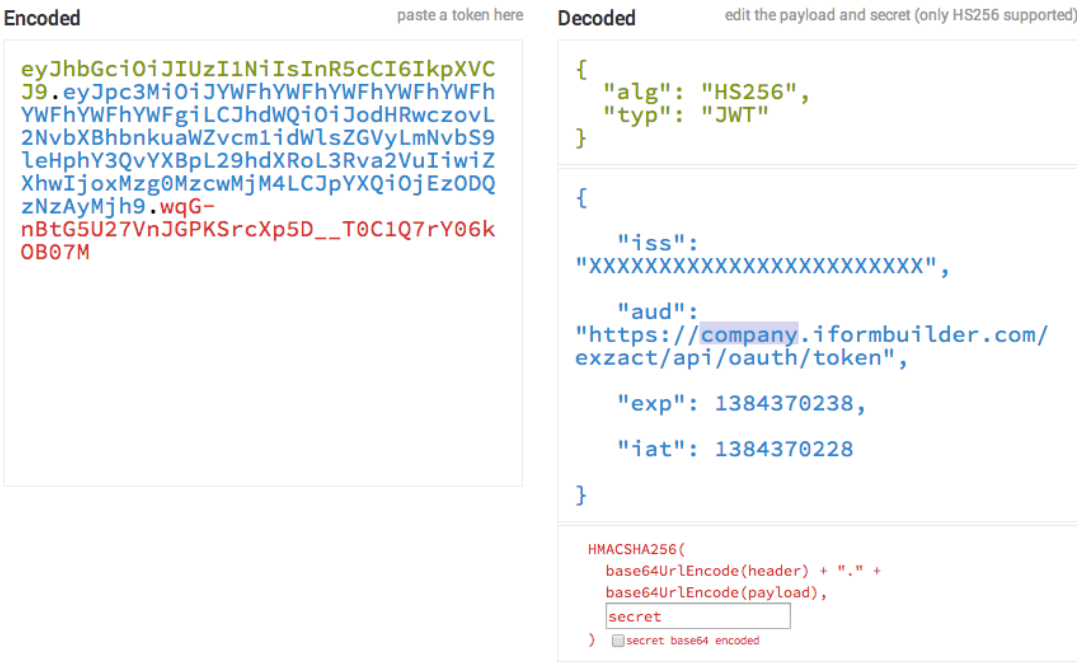


Fig. 5. How a JWT is Build up.

The JWS translates the JWT to a new format which can be found in Fig. 6. The same parts are still visible all are separated by a point. So, that the OpenID Provider and the resource server knows where which part ends.

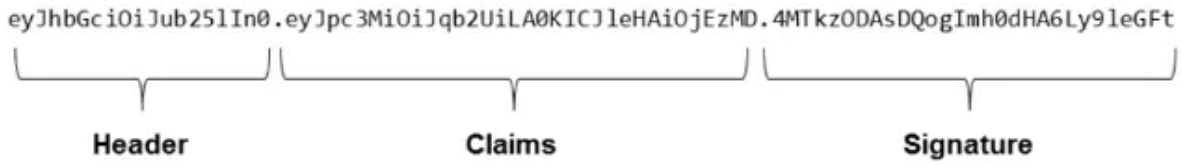


Fig. 6: Build up JWS.

With this the application has the possibility to know who is using the application. The application can then send the ID Token to a resource server, who can validate the ID token with authentication server, to check if the given user is authorized to see the requested data.

3.4.1 Sending ID Tokens By Reference

Fig. 7: ID token and the difference between by Value and by reference.

Fig. 7 shows the difference with the ID token sent by reference and the ID token sent by value. By value the Id token itself contains all the information. Which is for this instance John Doe. And by

reference a unique string is generated for the given Id token, which is linked to the original ID Token in the OpenID Provider. The ID token should only be sent by value between the resource server and the authorization server. As shown in Fig. 8

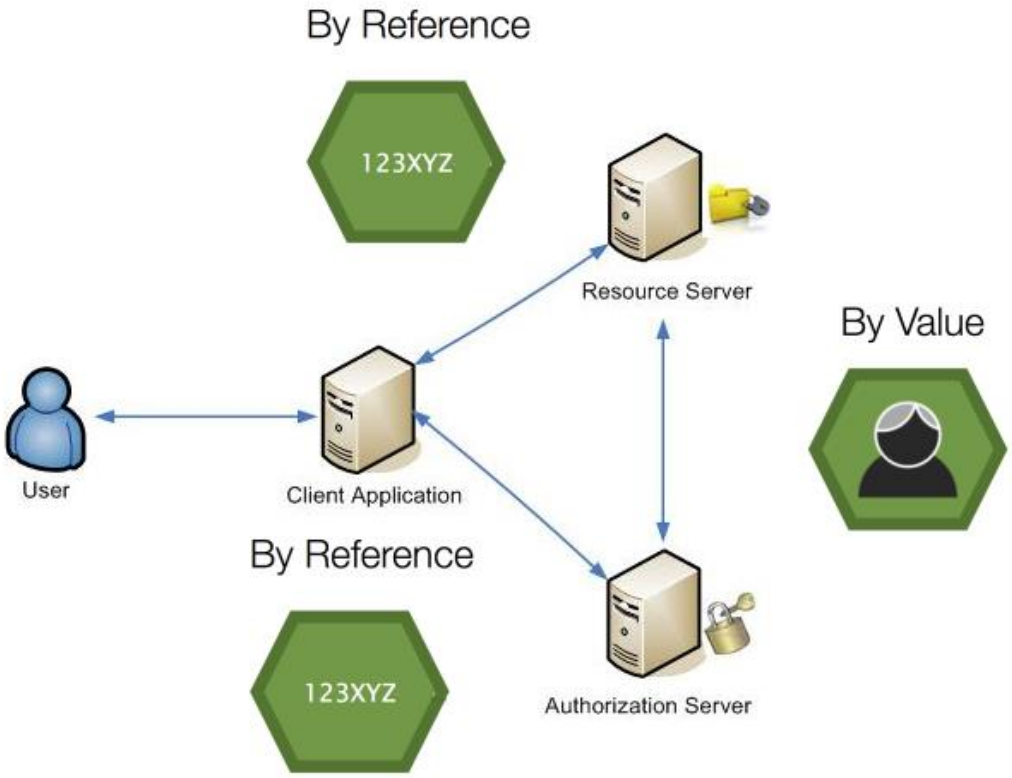


Fig. 8: ID token send by reference and by value flow.

With the given reference, it is possible for the application to create a session management database. Because the application "knows" which user is now using the application with the given reference of the ID Token. With this it is also possible to cancel a session, by revoking an ID token at the authorization server. Then the reference to the id token will be erased from the authentication provider which makes it impossible to use the given ID token by external applications.

3.5 Clients in Open ID Connect

Before a client can use an OpenID Provider it must first register itself. For this registration, the application owner must go to the OpenID Provider server's website where there is a possibility to register a new client, and edit an existing client. When the application owner wants to edit its client. It must fill in the Client-ID and Client-Secret which are used by the application.

When an application wants to register itself at an OpenID Connect provider. A client must enter:

- The name of the application.
- Which scopes are required by the application?
- Where the OpenID Provider should redirect the user too after completing the login and accepting the request of the application.
- The needed Grant type.

- The response types.

After registration, the client will receive a Client-ID and a Client-Secret. This Client-ID and Client-Secret must be stored within the application itself. Because these are used for the first security check by the OpenID Provider. The application sends these to the OpenID Provider, so the OpenID Provider can validate if the application (client) has the right to request data from it, and which data may be requested.

Every Client has a Client-ID and a Client-Secret. An example of such a representation is:

```
{
        Client-ID: a94a8fe5ccb19ba61c4c0873d391e987982fbbd3,
        Client-Secret: 423580962eee1a7c5f820186c3e2c7685518c921
}
```

These are added to the header of the access request.

For the user this progress will look different than normal. After the user, has opened the application and wants to log in. The application will redirect the user to the identity providers login page.

After the user has entered and submitted the username and password the user will be redirected to another page. Which shows the “scopes” which are requested by the user. At this moment, the user will have the possibility to accept or decline the request from the application. If it denies, the authentication flow is stopped and the application will not receive an access token and ID token. Which will result in that the user is not logged in. If the user accepts. The standard authentication flow will continue, which will result in the application receiving an access token and the ID Token, where the body of the ID Token is filled with the shown scopes.

3.6 Specifications

OpenID connect is built upon specifications, which can be found at: http://openid.net/specs/openid-connect-core-1_0.html

There are special software frameworks which help with the development of an Identity Provider or a Client. An example is: Identity Server: (http://identityserver.io/) The Identity Server software is tested against the specifications of OpenID Connect at each new release. Identity Server software is Opensource and can be easily setup and integrated.

As described in the chapter in chapter 3: OpenID Connect explained the extra layer which OpenID Connect defines is the Identity layer, which is implemented by the ID Token. The description of the ID token can be found at chapter 2: ID Token. from the following url: http://openid.net/specs/openid-connect-core-1_0.html#Overview.

The ID token contains Claims about the Authentication of an End-user by an Authorization Server. For security measures an ID Token must be represented as a JSON WEB Token(JWT) and be signed with JSON Web Signature (JWS). More information about JWT and JWS can be found at: https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32 and https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41

The following Claims are required for use in an ID Token:

- iss: An identifier for the Identity Server
- sub: The identifier of the subject (the client)
- aud: For which audience the ID token is intended.
- Exp: When the Identity token will expire and should no longer be excepted.
- Iat: the date time when the JWT was issued.

An example is stated below.

```
{
   "iss": "https://Identity.DigitalDelta.com",
   "sub": "24400320",
   "aud": "s6BhdRkqt3",
   "exp": 1311281970,
   "iat": 1311280970,
}
```

For more information about ID-tokens please refer to the Open ID Connect specifications in chapter 2.

3.7 Authentication

OpenID connect adds three new authentication flows:

- The Authorization Code Flow
- The Implicit Flow
- The Hybrid Flow

In the Digital Delta will only support:
Authorization Code Flow and Hybrid flow for user Authentication on their nodes. This because it is not possible to use Refresh tokens in the implicit flow.

The following table specifies which functionality each flow can and cannot do.

Property	Authorization Code Flow	Implicit Flow	Hybrid Flow
All tokens returned from Authorization Endpoint	no	yes	no
All tokens returned from Token Endpoint	yes	no	no
Tokens not revealed to User Agent	yes	no	no
Client can be authenticated	yes	no	yes
Refresh Token possible	yes	no	yes
Communication in one round trip	no	yes	no
Most communication server-to-server	yes	no	varies

When a client wants to use the Identity Server it must first be registered at the Identity Server.
The Registration process is unique for each Identity Server and how this is performed isn't bound to the specifications, However be sure the following values are requested:
Scope: which scopes your application requires. Such as: email, profile…):
Response_type: Which authorization flow is used. (In the Digital Delta it isn't possible to use the Implicit flow. Therefor, the response type: "id_token"or "id_token token" are illegal. For the complete list of response_type values, please refer the OpenId Connect specifications at chapter 3.
Client_id: The unique identifier for the client. This value must be unique for an Identity provider.
Redirect_uri: The URL which the user will be redirected to after a successful authentication process.
For a complete list of possible parameters of the Authentication Request see the OpenID Connect specifications at chapter 3.1.2.1
Registration is necessary because the above described parameters are required to perform an authorization request. The Authentication request which a client sends to the Identity server is structured in the following manner:

<urlIdentityServer/authorize><response_type><Scope><clientId><redirect_uri>

```
https:// Identity.DigitalDelta.com/authorize?
    response_type=code%20id_token%20token
    &scope=openid%20profile%20email
    &client_id=s6BhdRkqt3
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

3.8 Token Endpoint

When a client wants to obtain an Access token, an ID token or a refresh token: it should communicate with the token endpoint. This can be performed when a client presents its authorization grant to the token endpoint. By setting the grant_type parameter to authorization_code.

An example request can be viewed below.

```
POST /token HTTP/1.1
  Host: Identity.DigitalDelta.com
  Content-Type: application/x-www-form-urlencoded
  Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

  grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

When the Authentication server has validated the token Request (according to the OpenID connect Specifications chapter 3.1.3.2) and the validation was successful the Identity server will return a successful token response:

```
HTTP/1.1 200 OK
  Content-Type: application/json
  Cache-Control: no-store
  Pragma: no-cache

  {
   "access_token": "SlAV32hkKG",
   "token_type": "Bearer",
   "refresh_token": "8xLOxBtZp8",
   "expires_in": 3600,
"id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogIml
zcyI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4M
jg5NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAibi0
wUzZfV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyO
DA5NzAKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96
Sr_P6qJp6IcmD3HP99Obi1PRs-cwh3LO-
p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJNqeGpe-
gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7TpdQyHE5l
cMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoSK5hoD
alrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4XUVr
WOLrLl0nx7RkKU8NXNHq-rvKMzqg"
  }
```

It is important for the client to also validate the token Response, which it received from the Identity provider. See chapter 3.1.3.5 for how to validate the token response.

3.9 Userinfo endpoint

When the client is authorized and has a valid access token (indicating a user has logged in). It is possible to retrieve the user claims for the requested person by making a request to the UserInfo Endpoint. Therefor the client must submit the access token as a bearer token.

An example of a call to the userinfo endpoint is visualized below:

```
GET /userinfo HTTP/1.1
Host: Identity.DigitalDelta.com
Authorization: Bearer SlAV32hkKG
```

When the access token is successfully validated by the identity server. It will return an Userinfo response as shown below

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

{
 "sub": "248289761001",
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "email": "janedoe@example.com",
}
```

The request above will return all the claims which are excepted by the user and are requested by the client. For more specific information go to chapter 5.3 in the OpenID Connect specifications.

Be sure to validate which claims your application really needs, because of the recent changes in the AVG.

4 Implementation example

The purpose of this chapter is to describe what possibilities there are for implementing a Digitale Delta (DD) network that uses OpenID connect for authenticating (human) users and external applications. A DD network can be setup in different ways. Here we foresee the following possible configurations:
1. Separate DD-Nodes: Each DD-Node acts as an individual open endpoint to which users connect directly for retrieving data.
2. Multiple DD-Nodes with a browser front-end: A number of DD-Nodes are queried from a web browser that runs a JavaScript application.
3. Multiple DD-Nodes with a DD Web application: End Users run queries on a DD Web application. The web application runs on a server and has the ability to securely store configuration and make connections to DD-Nodes.
4. Multiple DD-Nodes are queried without user input required: An automated process runs queries on DD-Nodes to extract data.

5 Test Setup

For all tests we make use of the Google Identity provider.
(google configuration page: https://console.developers.google.com/apis)

Client IDs:

email :	pietje @voorbeeld.com
pw:	*********
clientId:	456785614089-tcnlliukb8stl16oq3qe0q9o9eojpdr9.apps.googleusercontent.com
client secret:	newbBbFu_ezMy4U2pr_v3M6B

email :	klaas@gmail.com
pw:	*********
clientId:	394638190954-adfeqo2jfkf2e5pnjs1a71tmbl8l8aoc.apps.googleusercontent.com
client secret:	fgl3dWp8Pr9Z68w3gyA0PGLq

Redirect URI's:
These endpoints represent DD-Nodes or DD- Web application. They have been registered as valid endpoints for the above client ids using the Google configuration page.
http://localhost:9000/Callback
http://localhost:9001/Callback

Google Authorization URL
https://accounts.google.com/o/oauth2/auth

Google Token URL
https://accounts.google.com/o/oauth2/token

Google Toke INFO URL
https://www.googleapis.com/oauth2/v3/tokeninfo

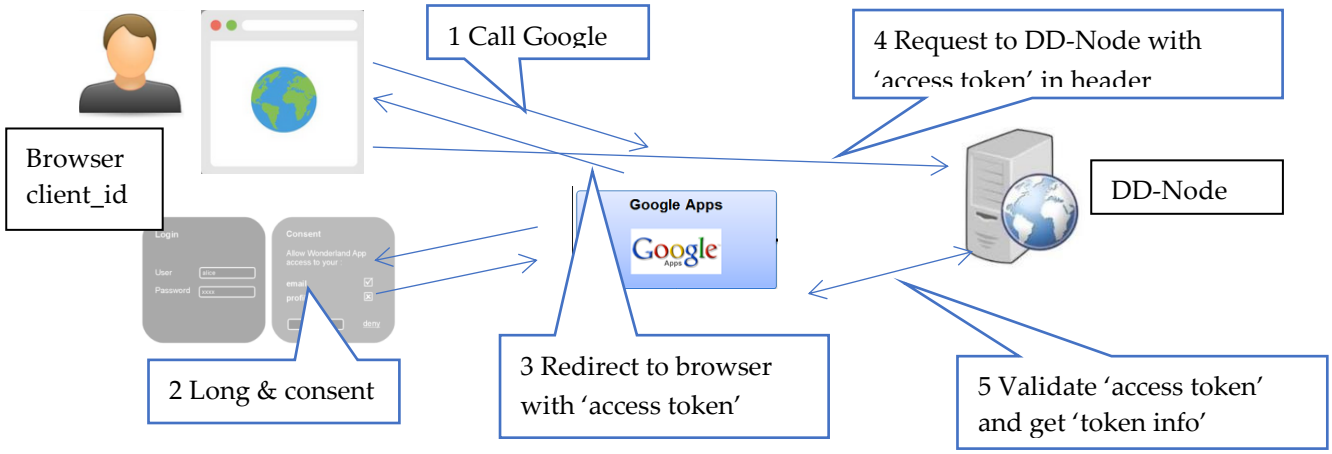
5.1 Use Case 1: Multiple DD-Nodes with browser front-end

In this use case an actual person is querying multiple DD-Nodes through a web browser. In this case the web browser takes care of the authentication steps. The browser queries the authentication endpoint and retrieves the retrieved 'access_token'. This token is validated and added as an Authorization header in the requests to the DD-Nodes as follows:

Authorization = Bearer <access_token>
This allows each DD-Node to retrieve information about the user without the user having to login to each DD-Node separately. This form of authentication is supported by OpenId but is considered the least secure. Each DD-Node will validate the 'access token' and use it to obtain the 'user info'.

5.1.1 Authentication Steps:

1. User calls the Google Auth endpoint with response_type=id_token token
2. From the Google Auth endpoint the user is directed to the Login & consent pages
3. After successful login Google redirects user to the redirect url of the browser application.
4. The browser application retrieves the 'access token' and inserts it into the header of the request to the DD-Node.
5. The DD-Node validates the 'access token' and uses it to retrieve the user information from the Google Token Info endpoint.



5.1.2 Example Requests

1. Call Google Auth endpoint:

https://accounts.google.com/o/oauth2/auth?client_id=2dsafa955614089-tcnlliukb8stl16oq3qe0qadfajpdr9.apps.googleusercontent.com&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2FCallback&response_type=id_token%20token&scope=openid%20email

&nonce=n0ts0R@nd0m

2. Redirect to Login & Consent page.
3. Redirect to web browser with 'access_token' and 'id_token':

id_token:
JWT Header :
{
"alg":"RS256",
"kid":"affc6290dsafd6182adc1fa4e81fdba6310dce63f"
}
JWT Body :
{
"azp":"249575614089-tcnlliukb8asdfsdfaqe0q9o9eojpdr9.apps.googleusercontent.com",
"aud":"249575614089-tcnlliukb8asdfsdfaqe0q9o9eojpdr9.apps.googleusercontent.com",
"sub":"101365017802132495088",
"email":"pietje@example.com",
"email_verified":true,
"at_hash":"610NGX8tROJ9URipOVZ8Qw",
"nonce":"n0ts0R@nd0m",
"exp":1524641334,
"iss":"accounts.google.com",
"jti":"337f99457d625d176c63821fbaf62f6243068ca2",
"iat":1524637734
}
JWT Signature:
{
<left out>
}

access_token:
ya29.GmCoBZMt345-2dsRfECrAO2HmUGx-ZaQENuSqr4Jfuc2YwyvxY6RH-zfvDFuYniByG70b26D3F0XoVje8oGrH4Y_GXmwNqhUR0rMCm_yoZAwB9SaIcy6aHs6PLyye0r9_IQ

4. Validate the 'id_token' for correctness
5. Web browser inserts 'access_token' into header of DD-Node request
6. DD-Node uses 'access_token' to call Google Token Info endpoint and retrieves the user email address.

5.2 Use Case 2: Multiple DD-Nodes with a DD Web application

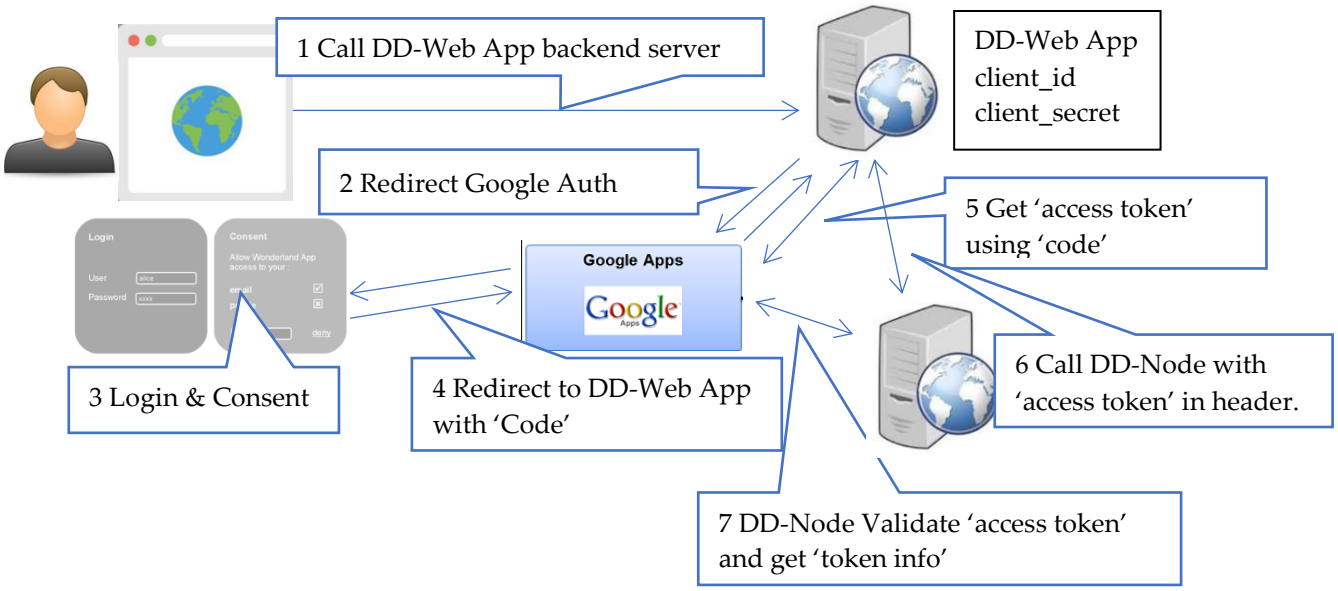
In this use case an actual person is querying multiple DD-Nodes through the DD Web Application. In this case the web application takes care of the authentication steps. The web application calls the Google endpoints; auth and token. The retrieved 'access_token' is inserted as an Authorization header in the requests to the DD-Nodes as follows:

Authorization = Bearer <access_token>

This allows each DD-Node to retrieve information about the user without the user having to login to each DD-Node separately. This form of authentication is more secure then the example in use case 2 as the web application is able to retrieve the 'access_token' using the 'client_id' and 'client_secret'. The Web Application can store these credentials securely opposed to the Web browser of use case 2. Each DD-Node will validate the 'access token' and use it to obtain the 'user info'.

5.2.1 Authentication Steps:

1. User calls the DD-Web Application endpoint with response_type=code.
2. DD-Web Application redirects call to Google Auth endpoint with response_type=code
3. From the Google Auth endpoint the user is directed to the Login & consent pages
4. After successful login Google redirects user to the redirect url of the DD-Web Application
5. The DD-Web Application retrieves the 'code' from from the response and uses this to obtain an 'access token' from Googles Token endpoint.
6. The DD-Web Application inserts the 'access token' into the request header to the DD-Nodes.
7. The DD-Node validates the 'access token' and uses it to retrieve the user information from the Google Token Info endpoint.



5.2.2 Example Requests

1. Call DD-Web Application: http://localhost:9000/ddwebapp/query
2. Call Google Auth endpoint:

   https://accounts.google.com/o/oauth2/auth?client_id=249asdf614089-tcnlliukb8stl16oq3qeasdfo9eojpdr9.apps.googleusercontent.com&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2FCallback&response_type=code&scope=openid%20email&nonce=n0ts0R@nd0m

3. Redirect to Login & Consent page.
4. Redirect to DD-Web Application with 'code':
   code= 4/AABCUB_dc7C1H3u9CVH4vHUlivbBiE5jeiXZPKThUkEae9xx2dJ3dDSvABP-6WgFtaUFUrANwKDWIoGdYpXisNA
5. Call Google Token endpoint:
   POST https://accounts.google.com/o/oauth2/token
   Content-Type: application/x-www-form-urlencoded
   code=4/AABCUB_dc7C1H3u9CVH4vHUlivbBiE5jeiXZPKThUkEae9xx2dJ3dDSvABP-6WgFtaUFUrANwKDWIoGdYpXisNA&client_id=2asd75614089-tcnlliukb8stl16oq3qe0adsf9eojpdr9.apps.googleusercontent.com&client_secret=oldbBbFu_ezMy4U2pr_v2M6B&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2FCallback&grant_type=authorization_code

   Response:
   id_token:
   JWT Header :
   {
   "alg":"RS256",
   "kid":"affc62907a446182adc1fa4e81fdba6310dce63f"
   }
   JWT Body :
   {
   "azp":"249575614089-tcnlliukbasdf6oq3qe0q9o9eojpdr9.apps.googleusercontent.com",
   "aud":"249575614089-tcnlliukbasdf6oq3qe0q9o9eojpdr9.apps.googleusercontent.com",
   "sub":"101365017802132495088",
   "email":"pietje@example.com",
   "email_verified":true,
   "at_hash":" hbUAqa7-D4l0ykDByNwhag",
   "nonce":"n0ts0R@nd0m",
   "exp": 1524643093,
   "iss":"accounts.google.com",
   "iat": 1524639493
   }
   JWT Signature:
   {
   <left out>
   }

   access_token:
   ya29.GmCoBWQFFp1s5zcqt65wUke5No4lyifZuhkbl8xoDl7Wabyejo4s1JnnntXkQepgUXfq8nBcc_zMg7gWn-IHNgpTwOkyMAwBpcPu2rZixQgYSswqEDOXUfbV7a59QPFTDhA

6. Validate the 'id_token' for correctness
7. DD Web Application inserts 'access_token' into header of DD-Node request
8. DD-Node uses 'access_token' to call Google Token Info endpoint and retrieves the user email address.

5.3 Use Case 3: Query DD-Node without user input

In this use case there is no actual person. An automated process is trying the query the DD-Nodes directly or through the DD-Web application. In the above OpenID calls a user is always required to perform a login action. For this use case the only OpenID option available is to connect using a refresh_token. Other authentication options for connecting without a user are available in the OAuth2 protocol.

5.3.1 OpenID Refresh Token

To retrieve a refresh token a one-time manual authentication is required. Afterwards the access_token can be retrieved indefinitely with the refresh_token. The newly obtained access_token can be passed in the header of the data request to the DD-Nodes. This approach works for automated clients accessing a DD-Node directly or through a DD –Web Application.

5.3.2 OAuth2 Authorization

If you do not require OpenID then another option would be to use the OAuth2 protocol with grant types 'Password Grant' or 'Client Credential' instead.
Password Grant: Here a 'username' and 'password' are passed in the request to retrieve a token. If the client was issued a secret, the following parameters are also required a 'client_id' and 'client_secret'.
Client Credential: Here a 'client_id' and 'client_secret' are passed in the request to retrieve a token.

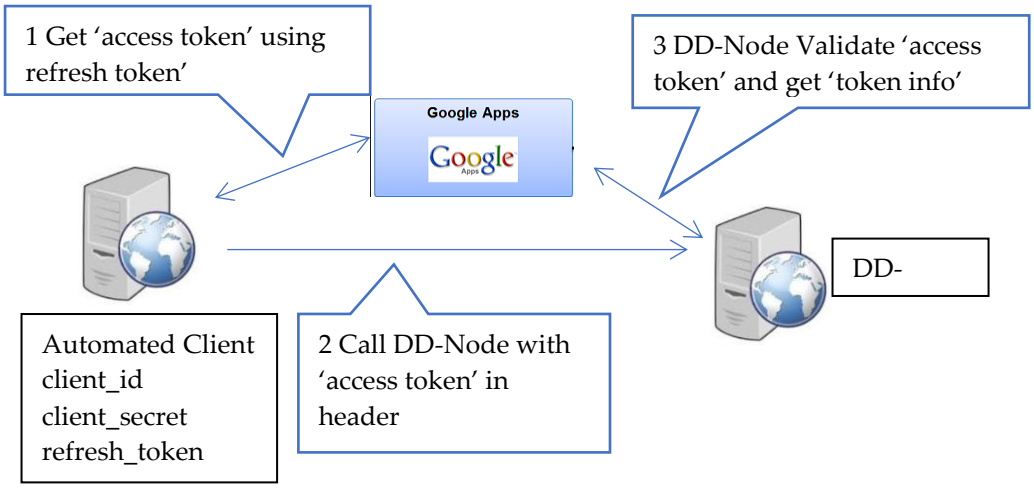
5.3.3 Authentication Steps (OpenID):

One-time steps

1. Call to Google Auth endpoint with response_type=code and access_type=offline.
2. From the Google Auth endpoint the user is directed to the Login & consent pages
3. Call the Google Token endpoint with the code value and grant_type=authorization_code
4. From the response extract and store the refresh_token.

Automated login steps

1. Call the Google Token endpoint using the 'refresh token', 'client_id' and 'client_secret'. Redirect the response back to the caller.
2. From the response retrieve the 'access token'. Insert 'access token' into header of to the DD-Node.
3. The DD-Node validates the 'access token' and uses it to retrieve the user information from the Google Token Info endpoint.



5.3.4 Example Requests

One-time steps

1. Call Google Auth endpoint:
   https://accounts.google.com/o/oauth2/auth?client_id=249adsf614089-tcnlliukb8stl1adsf3qe0q9o9eojpdr9.apps.googleusercontent.com&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2FCallback&response_type=code&scope=openid%20email&access_type=offline&nonce=n0ts0R@nd0m

2. Redirect to Login & Consent page. Login with user **pietje@example.com**
3. Redirect to caller application with 'code':
   http://localhost:9000/Callback?code= 4/AACGgC3DnOzv-cBubXtpY6m2TYJFnrgw2J3gT89ZPPFgE9m-5THq2DgsgoDiXIiYgRLnWMcoi7y00ZcOdA3Qubc

4. Call Google Token endpoint:
   POST https://accounts.google.com/o/oauth2/token
   Content-Type: application/x-www-form-urlencoded
   code=4/AACGgC3DnOzv-cBubXtpY6m2TYJFnrgw2J3gT89ZPPFgE9m-5THq2DgsgoDiXIiYgRLnWMcoi7y00ZcOdA3Qubc&client_id=249575614089-tcnlliukb8stladsf3qe0q9o9eojpdr9.apps.googleusercontent.com&client_secret=asdfdabFu_ezMy4U2pr_v2M6B&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2FCallback&grant_type=authorization_code

   Response:
   id_token:
   JWT Header :
   {
   "alg":"RS256",
   "kid":"affc62907a446182adc1fa4e81fdba6310dce63f"
   }
   JWT Body :
   {
   "azp":"249575614089-tasdfb8stl16oq3qe0q9o9eojpdr9.apps.googleusercontent.com",
   "aud":"249575614089-tcnlsdfb8stl16oq3qe0q9o9eojpdr9.apps.googleusercontent.com",
   "sub":" 100190971737603621826",

"email":"pietje@example.com",
"email_verified":true,
"at_hash":"Xj0jESz8jfFi3sXivYm22g",
"nonce":"n0ts0R@nd0m",
"exp": 1524645051,
"iss":"accounts.google.com",
"iat": 1524641451
}
JWT Signature:
{
<left out>
}

access_token:
ya29.GluoBW9kxfz5Qb-Pj-JUoJ6GQRq0tu3qBM3Ke0Ru3uXO1voljTZe3JHD_3KfDnisqSCa-IdlLy5fqBh2oUlZMxO8nk_fP301bR9saGKnrNvryXuzVSczx_kUMfgR

refresh_token:
1/nAjk0n94ssy1EFGXRPRIoGsdfg1fN5LfzsgfzCG_qH9UptA06fpx4YsE7AzAmtis

Now we have the refresh token. With this token we can make repeated calls to the token endpoint without having to re-authenticate ourselves.

Automated login steps

1. Call Google Token endpoint with 'refresh token':
   POST https://accounts.google.com/o/oauth2/token
   Content-Type: application/x-www-form-urlencoded
   refresh_token=1/nAjk0n94ssy1EFdfaIoGTdZx1fN5LfzURmzCG_qH9UptA06fpx4YsE7AzAmtis&client_id=249asdf89-tcnlliukb8stl16oq3qe0qadsfpdr9.apps.googleusercontent.com&client_secret=aslasdBbad_ezMy4U2pr_v2M6B&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2FCallback&grant_type=refresh_token

   Reponse:
   id_token:
   JWT Header :
   {
   "alg":"RS256",
   "kid":"affc62907a446182adc1fa4e81fdba6310dce63f"
   }
   JWT Body :
   {
   "azp":"249575614089-tcnlliukbasdfaoq3qe0q9o9eojpdr9.apps.googleusercontent.com",
   "aud":"249575614089-tcnlliukbasdfaoq3qe0q9o9eojpdr9.apps.googleusercontent.com",
   "sub":" 100190971737603621826",
   "email":"pietje@example.com",
   "email_verified":true,
   "at_hash":" z7lyHL5ozZorbZi9HhJ80Q",
   "nonce":"n0ts0R@nd0m",
   "exp": 1524645625,

"iss":"accounts.google.com",
"iat": 1524642025
}
JWT Signature:
{
<left out>
}

access_token:
ya29.GluoBQKfQUqRQYgts7olEJephDBpWoouDpFFRIiAceN9M3MXX9xOqpQt2w8XiBZAp
OCaFYiszi-Y6avuo1FVNB9YnxCgW37GDDJxtVNsat6eb8ZZ9WXjiiESj4b4

2. Caller application inserts 'access_token' into header of DD-Node request
3. DD-Node uses 'access_token' to call Google Token Info endpoint and retrieves the user email address.

5.3.5 Authentication Steps (OAuth2):

1. Call the Authentication Token endpoint using 'grant_type' with values 'password' or 'client_credentials'. Further parameters are:
   'username' and 'password' for grant _type=password, 'client_id' and 'client_secret'.
2. From the response retrieve the 'access token'. Insert 'access token' into header of to the DD-Node.

5.4 Validating ID_TOKEN

Each time an ID_Token is returned, it must be checked for validity.
An ID_TOKEN contains the following information:
Header:

- alg : Algorithm
- kid :

Body:

- azp : Audience this ID_Token is intended for. This must contain the client_ID of pietje@example.com
- aud : Audience this ID_Token is intended for. This must contain the client_ID of pietje@example.com
- sub : A unique identifier for the end user issued by issuer
- email :Email of authenticated user
- email_verified:
- at_hash: Access Token hash value
- nonce : String value used to associate a Client session with an ID Token, and to mitigate replay attacks.
- exp : After what time the ID_Token should not accepted
- iss : Issuing authority
- iat : Time that token is issued

1. Required parameters check: iss, sub, aud, exp and iat are required parameters in the id_token

   Validation steps:
2. Check that the ID token's crypto algorithm matches the one which the client has registered with the OpenID provider;
3. Validate the ID token signature; Public key can be found under endpoint:

   https://www.googleapis.com/oauth2/v1/certs

   Use the public key and 'alg' to encode token 'header'.'body'. The output should be equal to the signature provided in the token.

4. Validate the ID token claims:
   - expiration: The current date/time must be before the expiration date/time listed in the exp claim (which is a Unix timestamp). If not, the request must be rejected.
   - issuer — does the token originate from the expected IdP (accounts.google.com)?
   - audience — is the token intended for me?
   - timestamps — is the token within its validity window?
   - nonce — if set, is it the same as the one in my request?