Chapitre 3

Modern Authentication and Authorization

Business Training

1

Outline

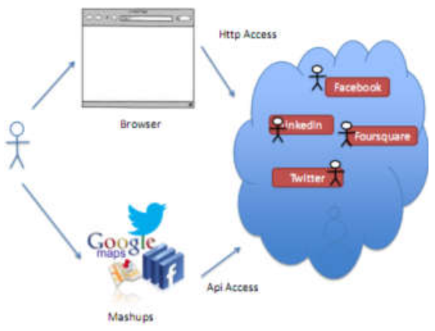
- The Internet and a way of sharing
- Introducing claims based security
- Understanding tokens and their representation on the net
- Introducing OAuth 2
- OpenID Connect: Adding sign-in to OAuth2
- EU Login (aka ECAS)

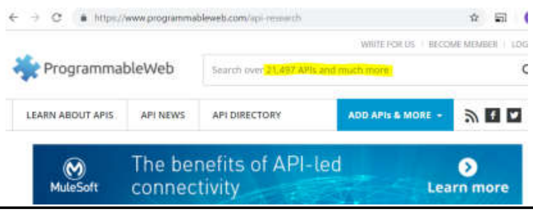
2

The Internet and a way of sharing

3

REST APIs Proliferation

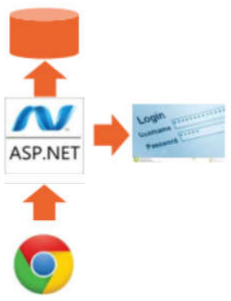


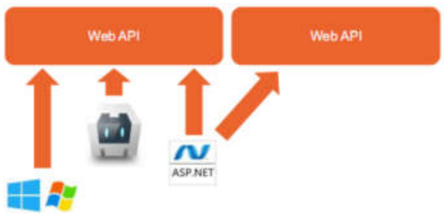

https://www.programmableweb.com/api-research
WRITE FOR US | BECOME MEMBER | LOG
ProgrammableWeb
Search over 21,497 APIs and much more
LEARN ABOUT APIS
API NEWS
API DIRECTORY
ADD APIs & MORE
MuleSoft
The benefits of API-led connectivity
Learn more

4

Classic Web Access vs Modern Web Access

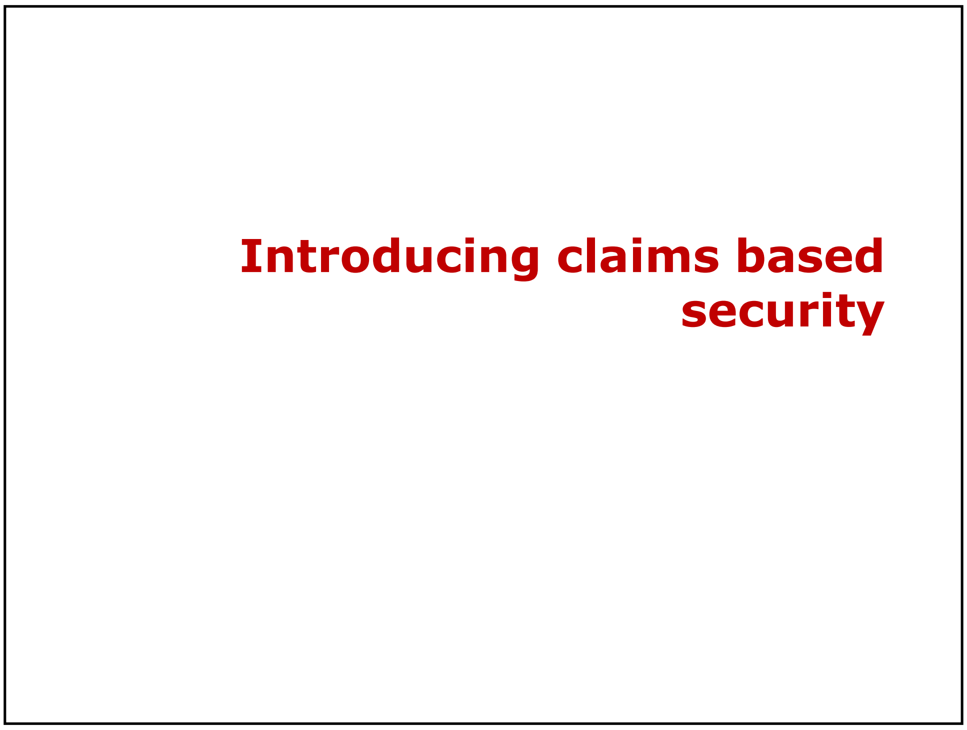
Classic Web acess **Modern applications**





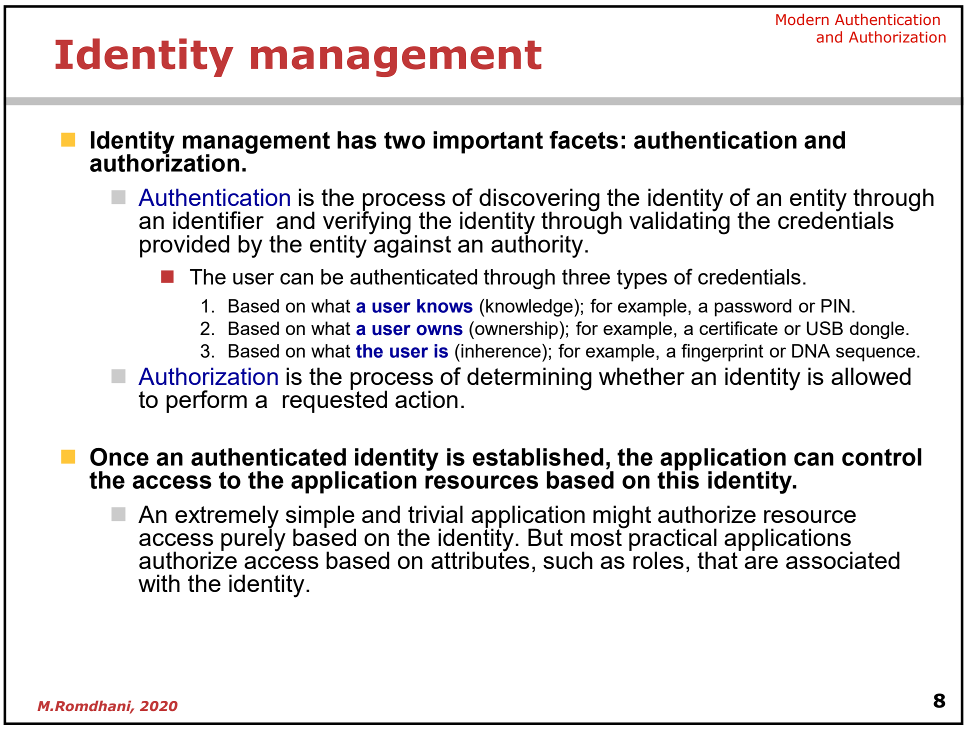
Problem Statement

- SSO & Federation
- Using "Authentication & Authorization" "across different systems" needs "Standards"



Introducing claims based security

7



Identity management

- **Identity management has two important facets: authentication and authorization.**
  - Authentication is the process of discovering the identity of an entity through an identifier and verifying the identity through validating the credentials provided by the entity against an authority.
    - The user can be authenticated through three types of credentials.
      1. Based on what **a user knows** (knowledge); for example, a password or PIN.
      2. Based on what **a user owns** (ownership); for example, a certificate or USB dongle.
      3. Based on what **the user is** (inherence); for example, a fingerprint or DNA sequence.
  - Authorization is the process of determining whether an identity is allowed to perform a requested action.
- **Once an authenticated identity is established, the application can control the access to the application resources based on this identity.**
  - An extremely simple and trivial application might authorize resource access purely based on the identity. But most practical applications authorize access based on attributes, such as roles, that are associated with the identity.

8

Role-Based Security

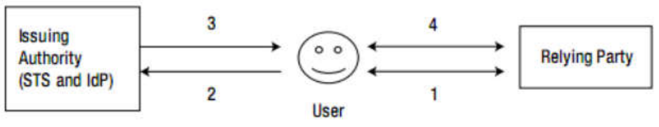
- **Role-based security is the most commonly used security model in the business or enterprise applications. The major benefit of using a role-based security model is the ease of security administration.**
  - The access rights are not given to an individual user, but to an abstraction called **a role**. A user gets assigned to one or more roles, through which the user gets access rights.
  - With this model, the security administration becomes a matter of managing the roles (typically far fewer than users) by assigning and unassigning roles to the users.
- **Role-based security has been around for a long time in the .NET Framework, starting with version 1.0. Identity (GenericIdentity) and Principal(GenericPrinciple) are the two abstractions provided by the .NET Framework for implementing role-based security.**
- **Limitations of Role-based security:**
  - “Is User X a member of Role Y?”
    - Broad permissions
  - “Is User X a member of Role Y for Item Z?”
    - Permissions in the context of an item (Impossible with role-based security)
  - Maintenance concerns (Not application security concerns)
    - Fine for simple apps with simple security “

What is a Claim ?

- **A claim in the world of authentication and authorization can be defined as a statement about an entity, typically a user. A claim can be very fine grained. The following list shows examples of claims.**
    - This user’s name is Tom.
    - Tom’s e-mail is tom@yahoo.com
    - Tom lives in New York.
    - Tom can delete users.
  - These statements are a lot more expressive than just putting a user in a specific role, such as admin, manager, IT etc.
- **The underpinning of claims-based architecture is trust.**
  - If I present to an application the claims created by an entity that the application trusts, then the application goes by the trust and accepts the claim. In this case, the application relies on the other entity.
    - This kind of application is known as a Relying Party (RP) application.
    - The entity that the RP application relies on is called the issuing authority. The endpoint of an issuing authority that accepts requests for tokens and issues the same is called a Security Token Service (STS) or Identity Provider

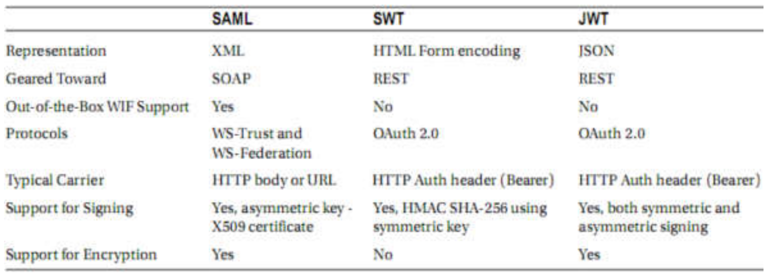
Workflow of Claim-based Security

- **The sequence of steps that happen in a Claims-based security is as follows:**
  1. The user requests an action. The RP asks for a token.
  2. The user presents the credentials to the issuing authority that the RP application trusts.
  3. The issuing authority issues a signed token with claims, after authenticating the user's credentials.
  4. The user presents the token to the RP application. The application validates the token signature, extracts the claims, and—based on the claims—either accepts or denies the request.



Token formats

- **There are three standard token formats:**
  - Security Assertion Markup Language (SAML), Simple Web Token (SWT), and JSON Web Token (JWT)



	SAML	SWT	JWT
Representation	XML	HTML Form encoding	JSON
Geared Toward	SOAP	REST	REST
Out-of-the-Box WIF Support	Yes	No	No
Protocols	WS-Trust and WS-Federation	OAuth 2.0	OAuth 2.0
Typical Carrier	HTTP body or URL	HTTP Auth header (Bearer)	HTTP Auth header (Bearer)
Support for Signing	Yes, asymmetric key - X509 certificate	Yes, HMAC SHA-256 using symmetric key	Yes, both symmetric and asymmetric signing
Support for Encryption	Yes	No	Yes

Introducing OAuth 2

13

OAuth fundamentals:
Authorization Code Grant, Implicit Grant and Client Credential Grant

14

OAuth2 Roles

- **The Resource Owner: The User**
  - The resource owner is the person who is giving access to some portion of their account
- **The Client: The Application**
  - The client is the application that is attempting to get access to the user's account. It needs to get permission from the user before it can do so.
- **The Resource Server: The API**
  - The resource server is the API server used to access the user's information.
- **The Authorization Server**
  - This is the server that presents the interface where the user approves or denies the request. In smaller implementations, this may be the same server as the API server, but larger scale deployments will often build this as a separate component.

What are we trying to do with OAuth2?

- **A “user” (resource owner),**
  - who has an account on “**Auth Server**” &
  - who has a resource on “**Resource Server**”
  - Delegates access privilege to a “**Client**” (app)
    - using a token generated by the “**Auth Server**”
    - To access the Resource on “**Resource Server**”
- **OAuth2 – Main Value Add**
  - Resource Server has **no** information regarding the credentials of the Resource owner
  - But can still give him or his clients access

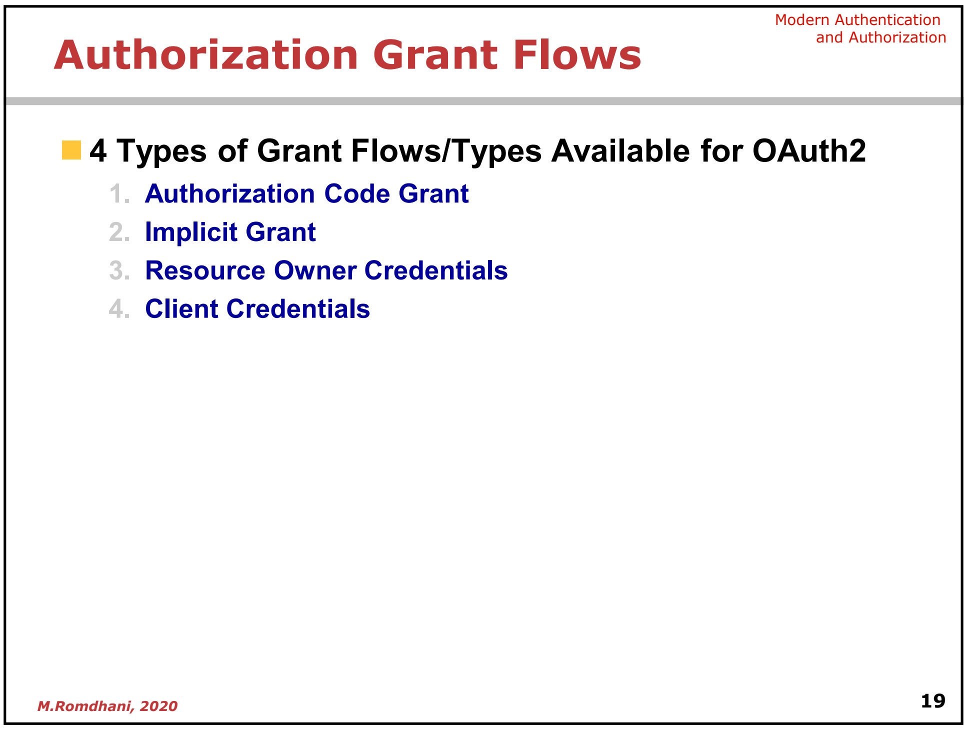
When do you need OAuth2?

1. **AuthServer :**
   - Your Service wants to implement Identity Management and expose APIs for other services to use your identity
2. **Client :**
   - Your Service wants SSO with Google, Office365, etc or a Proprietary Auth Server
3. **Resource Server:**
   - Your Service wants to expose Resources but would like authentication to happen with another service (internal or external)
   - Your big Service wants to split into multiple Services (aka micro-services) and would like SSO & Authorization checks between micro-services.

Setup of OAuth2 [one time setup]

- **Client (App) Registration**
  - One time only, for every Client App.
  - Usually a registration form in the developer or API section of AuthServer website

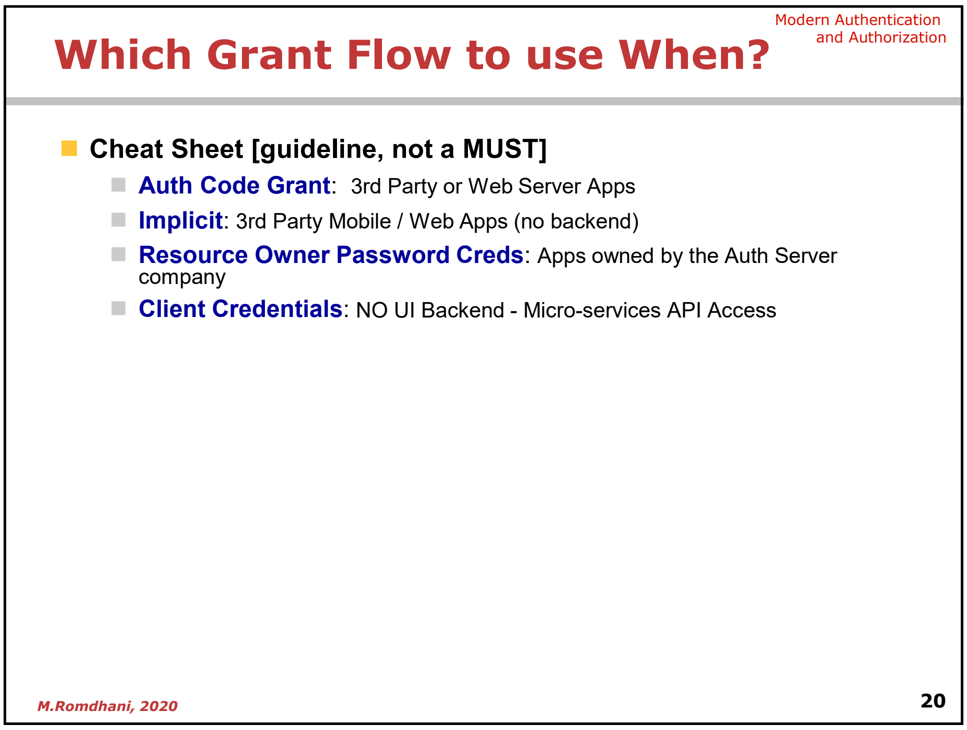
  - Client App will provide following to Auth Server (or more)
    1. **App Name** [shows resource owner]
    2. **App Website URI**
    3. **Redirect URI**
  - Auth Server will provide following to Client
    1. **Client Id** [publicly exposed]
    2. **Client Secret** [should be protected]



Authorization Grant Flows

- 4 Types of Grant Flows/Types Available for OAuth2
  1. **Authorization Code Grant**
  2. **Implicit Grant**
  3. **Resource Owner Credentials**
  4. **Client Credentials**

19



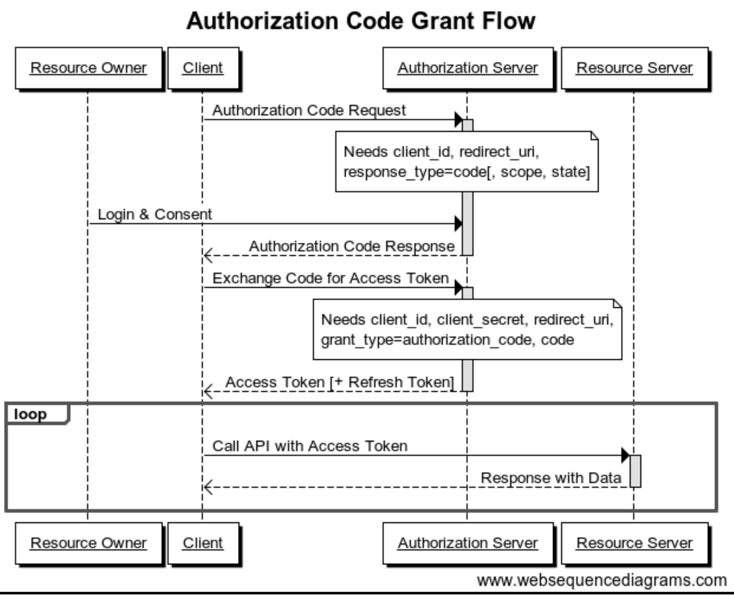
Which Grant Flow to use When?

- **Cheat Sheet [guideline, not a MUST]**
  - **Auth Code Grant**: 3rd Party or Web Server Apps
  - **Implicit**: 3rd Party Mobile / Web Apps (no backend)
  - **Resource Owner Password Creds**: Apps owned by the Auth Server company
  - **Client Credentials**: NO UI Backend - Micro-services API Access

20

Modern Authentication
and Authorization

Flow #1: Auth Code Grant Flow



Authorization Code Grant Flow
Resource Owner
Client
Authorization Server
Resource Server
Authorization Code Request
Needs client_id, redirect_uri, response_type=code[, scope, state]
Login & Consent
Authorization Code Response
Exchange Code for Access Token
Needs client_id, client_secret, redirect_uri, grant_type=authorization_code, code
Access Token [+ Refresh Token]
loop
Call API with Access Token
Response with Data
Resource Owner
Client
Authorization Server
Resource Server
www.websequencediagrams.com

M.Romd

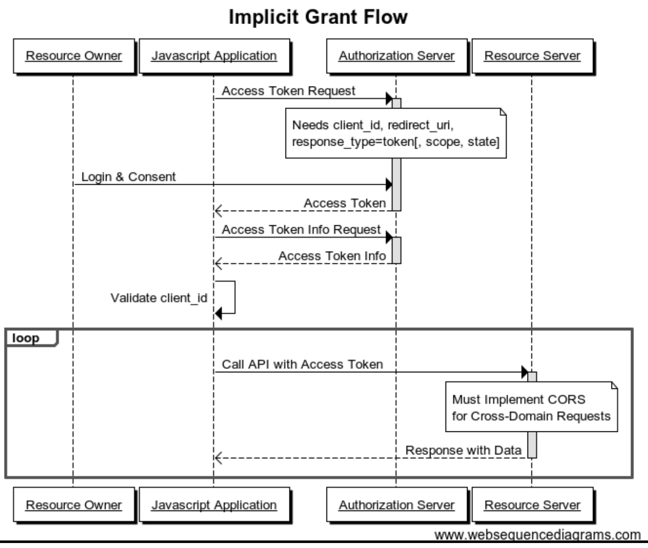
21

21

Modern Authentication
and Authorization

Flow #2: Implicit Grant

- No Auth Code – directly get a token
  - Basically, we don't trust client to keep client-secret secure.



M.Romdhani, 202

22

22

Flow #2: Implicit Grant

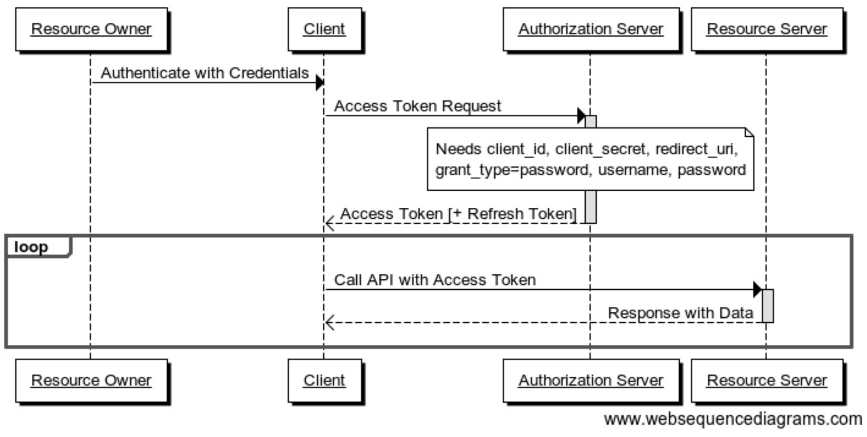
- **Flow 2: Implicit Grant**
  - **Did you notice the flow ?**
    - “has one less leg” – no auth code
    - “has no client secret”
    - “has no refresh token”
  - **Why is Implicit Flow so small in comparison to Code Grant ?**
    - For JavaScript Front ends – cannot keep a secret (client-secret or refresh-token)
    - Used for Limited time user sessions
    - Requires **Cross Origin Resource Sharing (CORS)**
    - Client has to prevent **CSRF (Cross site Request forgery)**

      So, in a sense it maybe less secure for the Resource Server

Flow #3: Resource Owner Password Credentials

- **Only for really trusted clients [owned by Auth/Resource server]**
  - For eq: Mobile Apps of a web service

Resource Owner Password Credentials Grant Flow



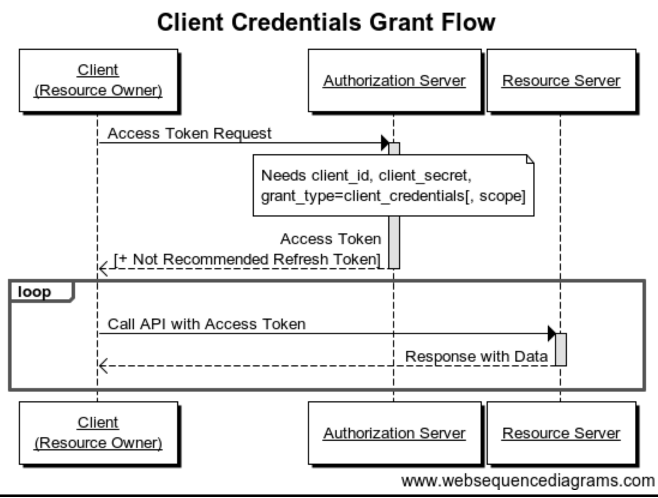
Resource Owner
Client
Authorization Server
Resource Server
Authenticate with Credentials
Access Token Request
Needs client_id, client_secret, redirect_uri, grant_type=password, username, password
Access Token [+ Refresh Token]
loop
Call API with Access Token
Response with Data
www.websequencediagrams.com

Flow #3: Resource Owner Password Credentials

- **When it should be used?**
  - With this type of authorization, the credentials (and thus the password) are sent to the client and then to the authorization server. It is therefore imperative that there is absolute trust between these two entities.
  - It is mainly used when the client has been developed by the same authority as the authorization server. For example, we could imagine a website named example.com seeking access to protected resources of its own subdomain api.example.com. The user would not be surprised to type his login/password on the site example.com since his account was created on it.
- **Example:**
  - Resource Owner: you having an account on acme.com website of the Acme company
  - Resource Server: Acme company exposing its API at api.acme.com
  - Client: acme.com website from Acme company
  - Authorization Server: an Acme server

Flow #4 : Client Credentials

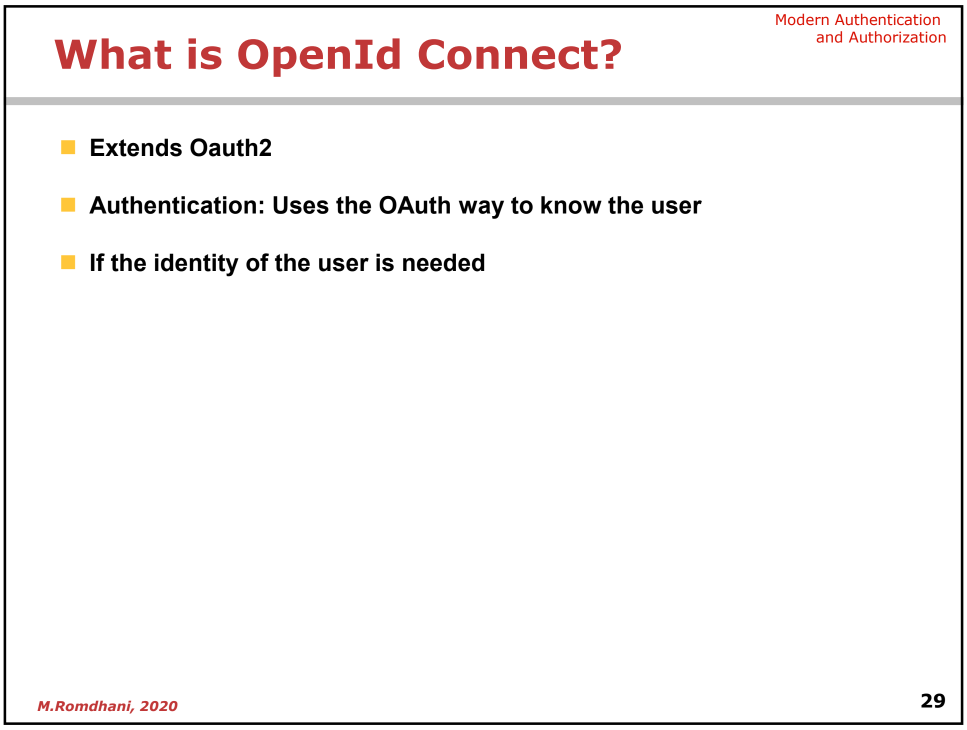
- **Micro-services in trusted network belonging to the same website as resource server**
  - Data access not related to a user


Client Credentials Grant Flow
Client (Resource Owner)
Authorization Server
Resource Server
Access Token Request
Needs client_id, client_secret, grant_type=client_credentials[, scope]
Access Token
[+ Not Recommended Refresh Token]
loop
Call API with Access Token
Response with Data
Client (Resource Owner)
Authorization Server
Resource Server
www.websequencediagrams.com

Flow #4 : Client Credentials

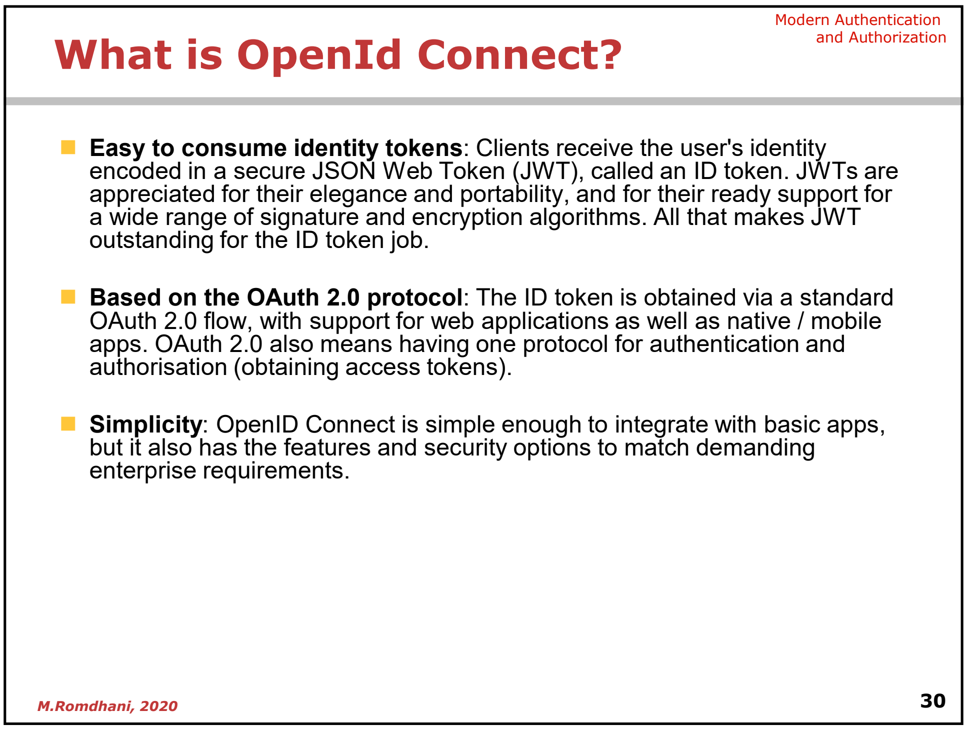
- **When it should be used?**
  - This type of authorization is used when the client is himself the resource owner. There is no authorization to obtain from the end-user.
- **Example:**
  - Resource Owner: any website
  - Resource Server: Google Cloud Storage
  - Client: the resource owner
  - Authorization Server: a Google server

OpenID Connect: Adding sign-in to OAuth2



What is OpenId Connect?

- **Extends Oauth2**
- **Authentication: Uses the OAuth way to know the user**
- **If the identity of the user is needed**

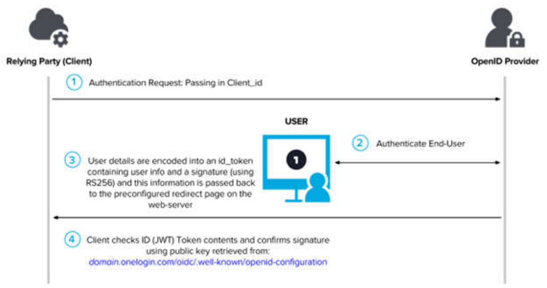


What is OpenId Connect?

- **Easy to consume identity tokens**: Clients receive the user's identity encoded in a secure JSON Web Token (JWT), called an ID token. JWTs are appreciated for their elegance and portability, and for their ready support for a wide range of signature and encryption algorithms. All that makes JWT outstanding for the ID token job.
- **Based on the OAuth 2.0 protocol**: The ID token is obtained via a standard OAuth 2.0 flow, with support for web applications as well as native / mobile apps. OAuth 2.0 also means having one protocol for authentication and authorisation (obtaining access tokens).
- **Simplicity**: OpenID Connect is simple enough to integrate with basic apps, but it also has the features and security options to match demanding enterprise requirements.

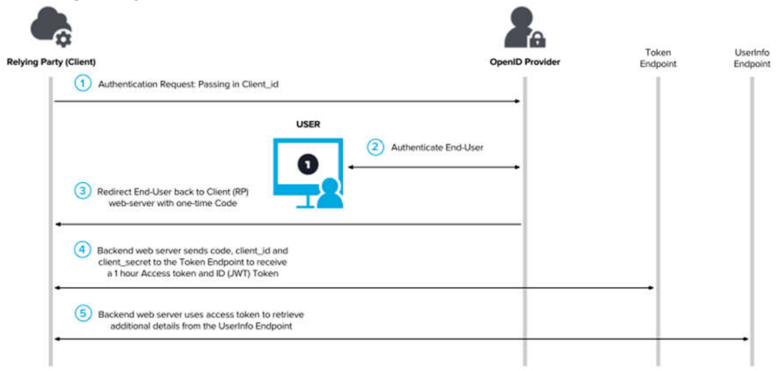
Implicit Flow

- The Implicit flow is required for apps and websites that have no back end logic on the web server, and everything that is passed between the app or site and the IdP can be viewed using browser development tools.
- In the Implicit flow, a public/private key (JSON Web Key or JWK) scheme is used to encrypt or sign user details. When you register your client app with the IdP (OneLogin), you will receive a client ID and a client secret. In an Implicit flow, the client secret should never be exposed.


Relying Party (Client)
OpenID Provider
1 Authentication Request: Passing in Client_id
USER
2 Authenticate End-User
3 User details are encoded into an id_token containing user info and a signature (using RS256) and this information is passed back to the preconfigured redirect page on the web-server
4 Client checks ID (JWT) Token contents and confirms signature using public key retrieved from: domain.onelogin.com/oidc/.well-known/openid-configuration

Authorization Cde Flow

- This flow is an option for apps that have web-server logic that enables back-end communication with the IdP (OneLogin). It functions like a traditional three-legged OAuth flow and results in a traditional OAuth access token being returned in secret to the web application via calls made on the back end. In this flow, rather than transmit the user details, the provider sends a special, one-time-use code that can be exchanged by the back-end web service for an OAuth access token.


Relying Party (Client)
OpenID Provider
Token Endpoint
UserInfo Endpoint
1 Authentication Request: Passing in Client_id
USER
2 Authenticate End-User
3 Redirect End-User back to Client (RP) web-server with one-time Code
4 Backend web server sends code, client_id and client_secret to the Token Endpoint to receive a 1 hour Access token and ID (JWT) Token
5 Backend web server uses access token to retrieve additional details from the UserInfo Endpoint

The Identity Token

- **The ID token resembles the concept of an identity card, in a standard JWT format, signed by the OpenID Provider (OP). To obtain one the client needs to send the user to their OP with an authentication request.**
- **Features of the ID token:**
  - Asserts the identity of the user, called subject in OpenID (sub).
  - Specifies the issuing authority (iss).
  - Is generated for a particular audience, i.e. client (aud).
  - May contain a nonce (nonce).
  - May specify when (auth_time) and how, in terms of strength (acr), the user was authenticated.
  - Has an issue (iat) and expiration time (exp).
  - May include additional requested details about the subject, such as name and email address.
  - Is digitally signed, so it can be verified by the intended recipients.
  - May optionally be encrypted for confidentiality.

The Identity Token

- **The ID token statements, or claims, are packaged in a simple JSON object:**

```
{
  "sub"       : "alice",
  "iss"       : "https://openid.c2id.com",
  "aud"       : "client-12345",
  "nonce"     : "n-0S6_WzA2Mj",
  "auth_time" : 1311280969,
  "acr"       : "c2id.loa.hisec",
  "iat"       : 1311280970,
  "exp"       : 1311281970
}
```

  - The ID token header, claims JSON and signature are encoded into a base 64 URL-safe string, for easy passing arround, for example as URL parameter.

```
eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzcyI6ICJodHRw
Oi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5NzYxMDAxIiw
KICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZfV3pBMk1qIi
wKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5NzAKfQ.ggW8hZ
1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6qJp6IcmD3HP9
9Obi1PRs-cwh3LO-p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJNqeGpe-gccM
g4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7TpdQyHE5lcMiKP
XfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoSK5hoDalrcvR
YLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4XUVrWOLrLl0
nx7RkKU8NXNHq-rvKMzqg
```

OIDC Flows

- **The type of OpenID Connect flow you should use has a lot to do with the type of client you're using and how well it can keep a secret.**
  - For example, if you're using a JavaScript application, where anything and everything can be looked at by someone using browser development tools, and there's no 'back end' logic in the web server that can do things away from the prying eyes of users… You must use the **Implicit flow** for OpenID Connect.
- **Now, if you're using a more traditional application, where some information is passed around on the front end (and anyone can peek at it) but it also has back end code that can talk to the provider in secret, you can use the Authorization Code Flow (although you don't have to)**
  - And there's also a **Hybrid Flow**, but at the end of the day, that's just a combination of both flows.

	Authorization Code	Implicit	Hybrid
All tokens returned from authorization endpoint	✗	✓	✗
All tokens returned from token endpoint	✓	✗	✗
Tokens sent via user agent	✗	✓	✓
Client can be authenticated (e.g. using client secret)	✓	✗	✓
Can use refresh tokens	✓	✗	✓
Communication in one round trip	✗	✓	✗

EU Login (aka ECAS)

ECAS / EU Login

- **ECAS is the European Commission Authentication Service that enables web applications to authenticate centrally with a common strong password, offering more security than the current LDAP password.**
  - It offers also single sign-on between applications using it.
  - **ECAS** is based upon the protocol and implementations developed by Yale University for **CAS** (Central Authentication Service).
- **Besides the password, ECAS provides a variety of authentication mechanisms: SMS challenge, Mobile App for iOS, Android and Windows Phone, hardware token, software token, eID, social networks (currently Facebook, Twitter and Google), etc.**
  - In 2016, ECAS has been rebranded EU Login.
- **ECAS For...**
  - **End User** — You are a user of some of the ECAS protected applications
  - **Casual bystander** — You are just curious about ECAS
  - **Developer** — You are a developer and you want your application protected by ECAS
  - **Local Service Desk** — You are member of the Local Service Desk
  - **ECAS Team** — You are a member of the ECAS team

What does ECAS / EU Login offer ?

- **Authentication with one of the available EU Login Authentication Methods**
- **Coarse-grained authorization**
- **Propagation of user security context, by the mean of Proxy Tickets**
- **Transaction signature**
- **Two servers with failover: https://ecas.cc.cec.eu.int:7002/cas**
- **A mockup server and a mockup client**
- **Clients and documentation for various technologies (mainly Oracle WebLogic and ColdFusion, but also PHP, Tomcat and JBoss)**
- **A tool for generating self signed certificates**
- **Attribute-based authorization**
- **Claims-Based Authentication**
- **Authentication using the SAML protocol**

Claims-Based Authentication

- EU Login offers support for Claims-based Authentication, where the identity of a user is represented by one or more claims contained in a security token.
- Security tokens are issued by a Security Token Service (STS) and consumed by an Application Service.
- If you are used to SAML terminology, a Security Token Service can be seen as an Identity Provider (IdP) and an Application Service as a Relying Party (RP) or, in other terms, a Service Provider (SP).
- Since version 7.2 EU Login can act as both Identity and Service Provider.

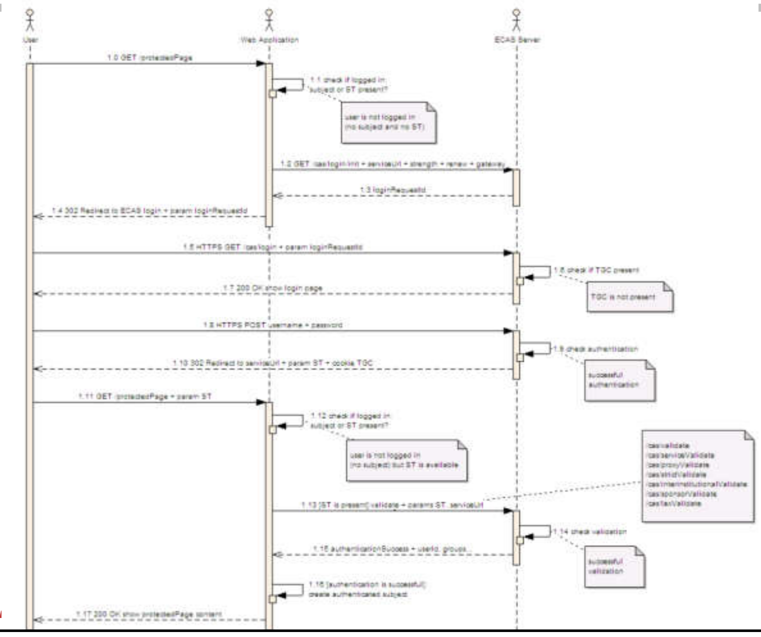
EU Login as Identity Provider

- **Register your application as Service Provider in EU Login**
  - To add your application as Service Provider in the EU Login Server configuration, please open an SMT ticket for the DIGIT NUPS IAM BO team, providing the following information:
    - the federation metadata of your application;
    - the claims your application would like to receive (see the "Supported claims" below);
    - one or more authentication strenght(s) accepted by your application.
    - Strengths currently compatible are: PASSWORD, PASSWORD_SMS, PASSWORD_TOKEN, PASSWORD_SOFTWARE_TOKEN, PASSWORD_MOBILE_APP and STORK;
- **Example of Use case : SharePoint 2010 (the Service Provider) wishes to obtain user identities from EU Login (the Identity Provider) using Claims-Based Authentication.**

OpenID Connect Token Types

- **OpenID Connect permits to obtain a variety of tokens which have different purposes, different sensitivity and different lifetime.**
  - **The ID Token**
    - The ID Token is merely a signed response (JWT) coming from the Identity Provider and telling who the user is.
    - It has an audience, which is the only application allowed to obtain it.
    - It has a unique ID, to prevent replay attacks.
    - It has a validity period, after which it is expired.
  - **The Access Token**
    - The Access Token gives access to a protected resource.
    - It can only be used once and is only valid for a very short time.
    - The aim of the Access Token is to be passed to another application to be granted access to it.
    - Access Tokens can be either "opaque" or JWT tokens.
  - **The Refresh Token**
    - The Refresh Token is kept inside an application to allow it to obtain Access Tokens giving access to remote services.
    - The Refresh Token is very sensitive and allows impersonating the end-user in all the services trusting the current application as a client.
    - The Refresh Token must never be passed to a different application than the one it was issued for.

ECAS/EU Login Generic Flow


User
Web Application
ECAS Server
1.0 GET /protectedPage
1.1 check if logged in: subject or ST present?
user is not logged in (no subject and no ST)
1.2 GET /cas/login/init + serviceUrl + strength + renew + gateway
1.3 loginRequestId
1.4 302 Redirect to ECAS login + param loginRequestId
1.5 HTTPS GET /cas/login + param loginRequestId
1.6 check if TGC present
TGC is not present
1.7 200 OK show login page
1.8 HTTPS POST username + password
1.9 check authentication
successful authentication
1.10 302 Redirect to serviceUrl + param ST + cookie TGC
1.11 GET /protectedPage + param ST
1.12 check if logged in: subject or ST present?
user is not logged in (no subject) but ST is available
/cas/validate
/cas/serviceValidate
/cas/proxyValidate
/cas/strictValidate
/cas/interinstitutionalValidate
/cas/sponsorValidate
/cas/laxValidate
1.13 [ST is present] validate + params ST, serviceUrl
1.14 check validation
successful validation
1.15 authenticationSuccess + userId, groups...
1.16 [authentication is successful] create authenticated subject
1.17 200 OK show protectedPage content

EU Login as Identity Provider

- **Register your application as Service Provider in EU Login**
  - To add your application as Service Provider in the EU Login Server configuration, please open an SMT ticket for the DIGIT NUPS IAM BO team, providing the following information:
    - the federation metadata of your application;
    - the claims your application would like to receive (see the "Supported claims" below);
    - one or more authentication strenght(s) accepted by your application.
    - Strengths currently compatible are: PASSWORD, PASSWORD_SMS, PASSWORD_TOKEN, PASSWORD_SOFTWARE_TOKEN, PASSWORD_MOBILE_APP and STORK;
- **Example of Use case : SharePoint 2010 (the Service Provider) wishes to obtain user identities from EU Login (the Identity Provider) using Claims-Based Authentication.**

ECAS Client Integration

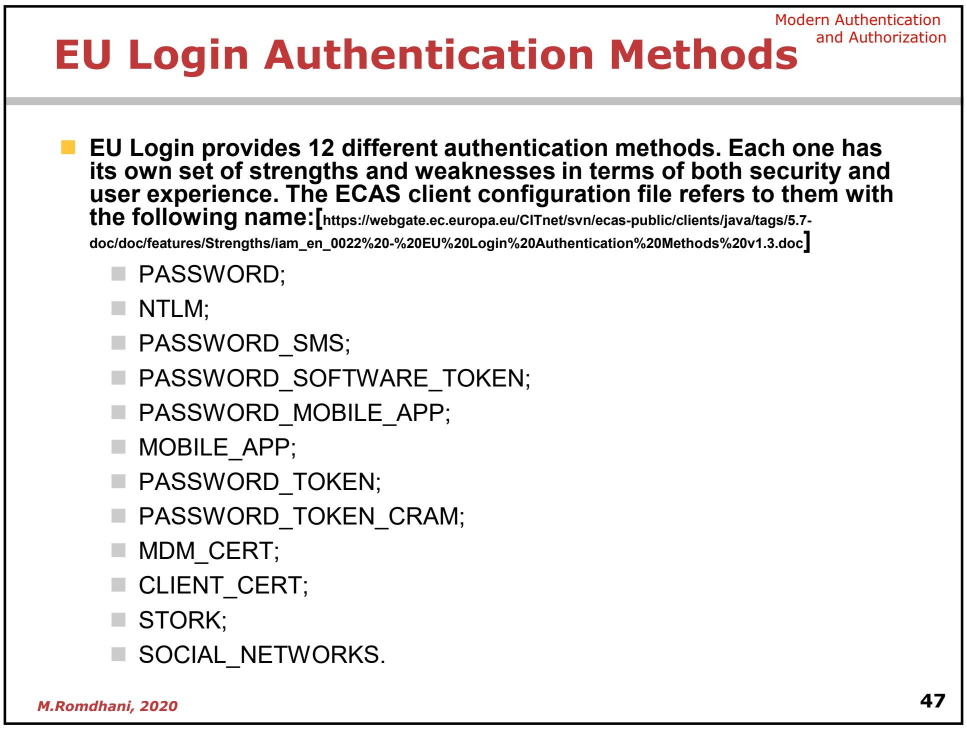
- **ECAS Client is available for diffreent technologies**
  - ECAS for JAX-WS Web services under WebLogic Server
  - ECAS Client for Java EE 6 with JBoss AS 10+ (WildFly 10+)
  - ECAS PHP Client (Simple lightwieght client)
  - ECAS PHP Client 2.x
  - ECAS Client For Apache Tomcat 6.0, 7.0 and 8.0

OpenID Connect for CAS users

- **As of version 6, EU Login supports an additional protocol to interoperate with protected applications: OpenID Connect.**
- **Similarities**
  - For traditional Web applications (i.e. applications with a back-end executing logic), OpenID Connect provides an authorization "code" flow, which is an authentication protocol using the same steps as the CAS protocol.
  - The end-user is redirected to the Identity Provider, authenticates there and is redirected back with an additional parameter in the URL.
  - This parameter is called the "code", it is the equivalent of the CAS Service Ticket.
  - The "*code*" has to be validated at the *token endpoint*, which is the equivalent of the CAS ticket-validation URL, where Service Ticket are validated.
  - The response to the validation of the code is called an ID Token, which is a JSON Web Token (JWT) and consists in the user ID and the user attributes (called "*claims*") with a signature.
  - So a JSON format is used instead of an XML format for CAS.

OpenID Connect for CAS users

- **Differences**
  - For OpenID Connect, every client application MUST be registered to the Identity Provider before it can use OpenID Connect. This constraint does not exist with CAS. Why is it so?
    - It is because in OpenID Connect, every client application must be assigned a *client ID* (and possibly a *client secret*) which are to be sent in every interaction with the server to authenticate the calling application.
    - With CAS, the identity of the calling application is deduced from the service URL which is used in the different interactions (e.g. authentication and ticket validation).
  - In OpenID Connect, the protected application must authenticate itself by providing its client ID and client secret when validating the "code".
  - Another difference between CAS and OpenID Connect is that when an error occurs with CAS, it is often displayed on the CAS server directly (for example: "*You were redirected by the application with an invalid strength* "BASIC") whilst for OpenID Connect most errors are sent back to the calling application.
    - This means that all OpenID Connect client applications must be able to handle errors returned to them and possibly display meaningful corresponding error messages to their end-users.
  - For CAS, any error which could be handled centrally was processed so and there was no need to duplicate error-handling in every client.



EU Login Authentication Methods

- **EU Login provides 12 different authentication methods. Each one has its own set of strengths and weaknesses in terms of both security and user experience. The ECAS client configuration file refers to them with the following name:[**https://webgate.ec.europa.eu/CITnet/svn/ecas-public/clients/java/tags/5.7-doc/doc/features/Strengths/iam_en_0022%20-%20EU%20Login%20Authentication%20Methods%20v1.3.doc**]**
  - PASSWORD;
  - NTLM;
  - PASSWORD_SMS;
  - PASSWORD_SOFTWARE_TOKEN;
  - PASSWORD_MOBILE_APP;
  - MOBILE_APP;
  - PASSWORD_TOKEN;
  - PASSWORD_TOKEN_CRAM;
  - MDM_CERT;
  - CLIENT_CERT;
  - STORK;
  - SOCIAL_NETWORKS.