OAUTH 2.0

OAuth 2.0 - It is Authorization delegation protocol and it has nothing to do with authentication.
We don't need to enter our user id & password in a 3rd party app.
In some cases we may see authentication using OAuth but that's not OAuth 2.0 that is OIDC.
OpenID Connect (OIDC) - is an authentication standard built on top of OAuth 2.0.

Authentication VS Authorization

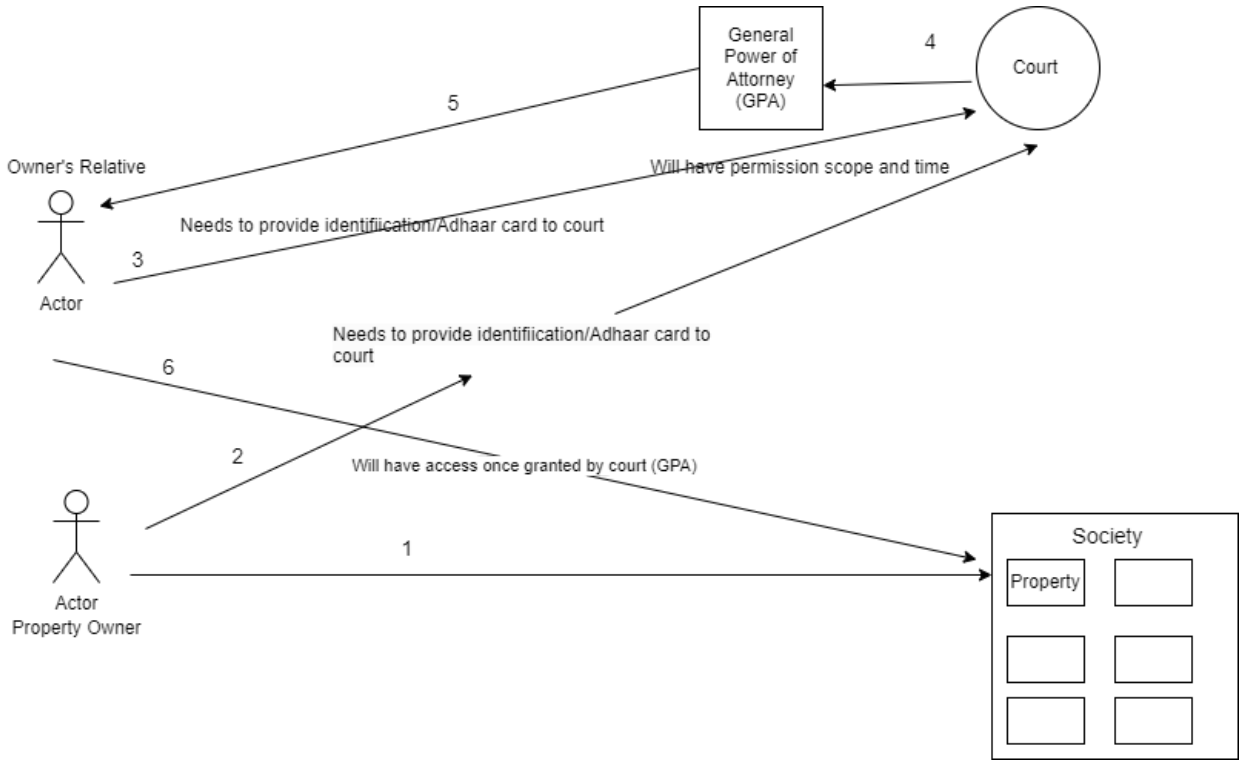
Authentication is identification. (Eg - We provide ID to provide the authenticity to the company.But if the ID is validated it doesn't mean we will have access to everything in the company.)
Authorization means the scope of the identity. What permission does the user have.

Grant type - They are methods through which applications can gain Access Tokens and by which you grant limited access to your resources to another entity without exposing credentials.

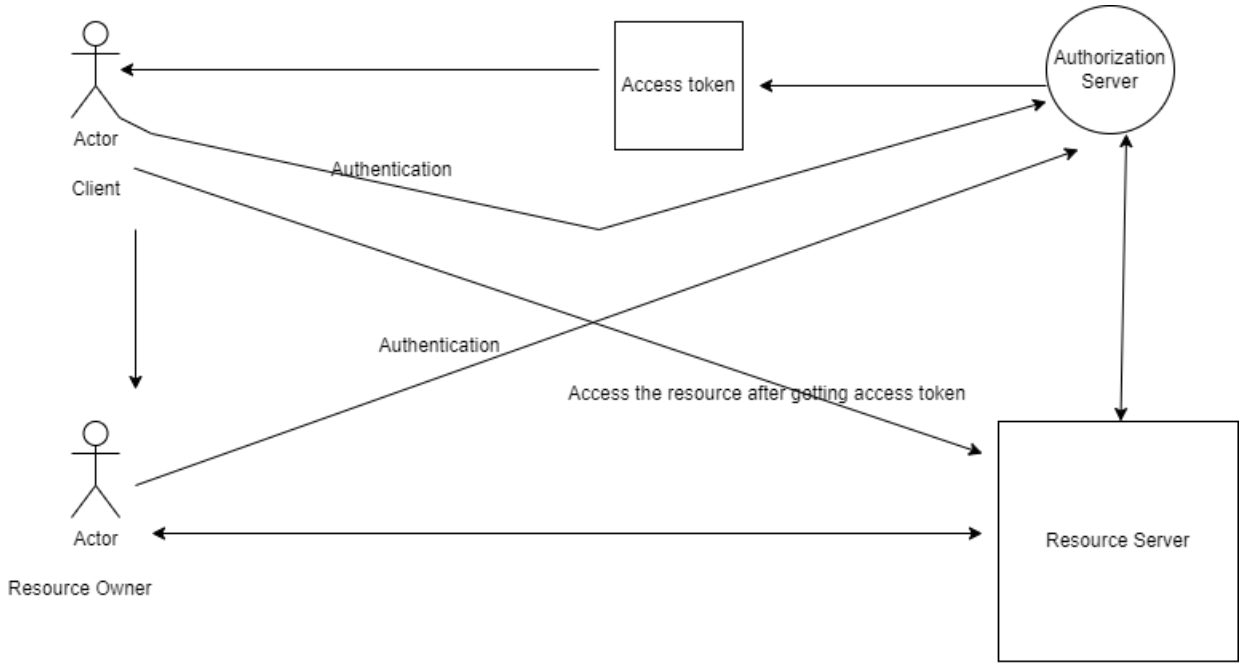
Use case--

Say you have a property somewhere in India.
You have to travel to some different country which means you will not be available to make that deal right to sign the paper and to sell that property right.So what will happen in this case?
we can delegate our authorization right to one of our relatives/friends, he will work on this particular task or particular thing on your behalf.
And that is exactly his delegation of authorization.
This happens in India using the General Power of Attorney. (A person can give another person a complete general right or power to act lawfully, with respect to his property or bank accounts, or tax payments, or a registration work to sue a third party etc)

General power of Attorney will have the scope/permission and the time in which he can make that deal or access the papers.



Now the same thing happens in case of OAuth 2.0



Important thing to note here is we dont authenticate to the client, we authenticate to Authorization server which holds our resource

Authorization code grant type

STEP -1

*Before step 1, draw.io has already registered itself with the Authorization server.
Authorization server will be able to authenticate it seeing its clientId& secret.

response_type=code - This tells the authorization server that the application is initiating the authorization code flow.
client_id - The public identifier for the application, obtained when the developer first registered the application.
redirect_uri - Tells the authorization server where to send the user back to after they approve the request.
scope - One or more space-separated strings indicating which permissions the application is requesting. The specific OAuth API you're using will define the scopes that it supports.
state - The application generates a random string and includes it in the request. It should then check that the same value is returned after the user authorizes the app. This is used to prevent CSRF attacks.
Assume state as a correlation Id which we have in mule.

STEP-2

Point to note here, user is login to authorization server(google in this case) and not the 3rd party app(draw.io)

STEP-3

Consent will be provided to the Authorization server by authentication.

STEP-4

Auth code is generated and redirected to the browser using redirect_uri,if step -2 is successful.
If the user approves the request, the authorization server will redirect the browser back to the redirect_uri specified by the application, adding a code and state to the query string.

STEP-5

Now that the application has the authorization code, it can use that to get an access token.

The application makes a POST request to the service's token endpoint with the following parameters:

grant_type=authorization_code - This tells the token endpoint that the application is using the Authorization Code grant type.
code - The application includes the authorization code it was given in the redirect.
redirect_uri - The same redirect URI that was used when requesting the code. Some APIs don't require this parameter, so you'll need to double check the documentation of the particular API you're accessing.
client_id - The application's client ID.
client_secret - The application's client secret. This ensures that the request to get the access token is made only from the application, and not from a potential attacker that may have intercepted the authorization code.

STEP-6

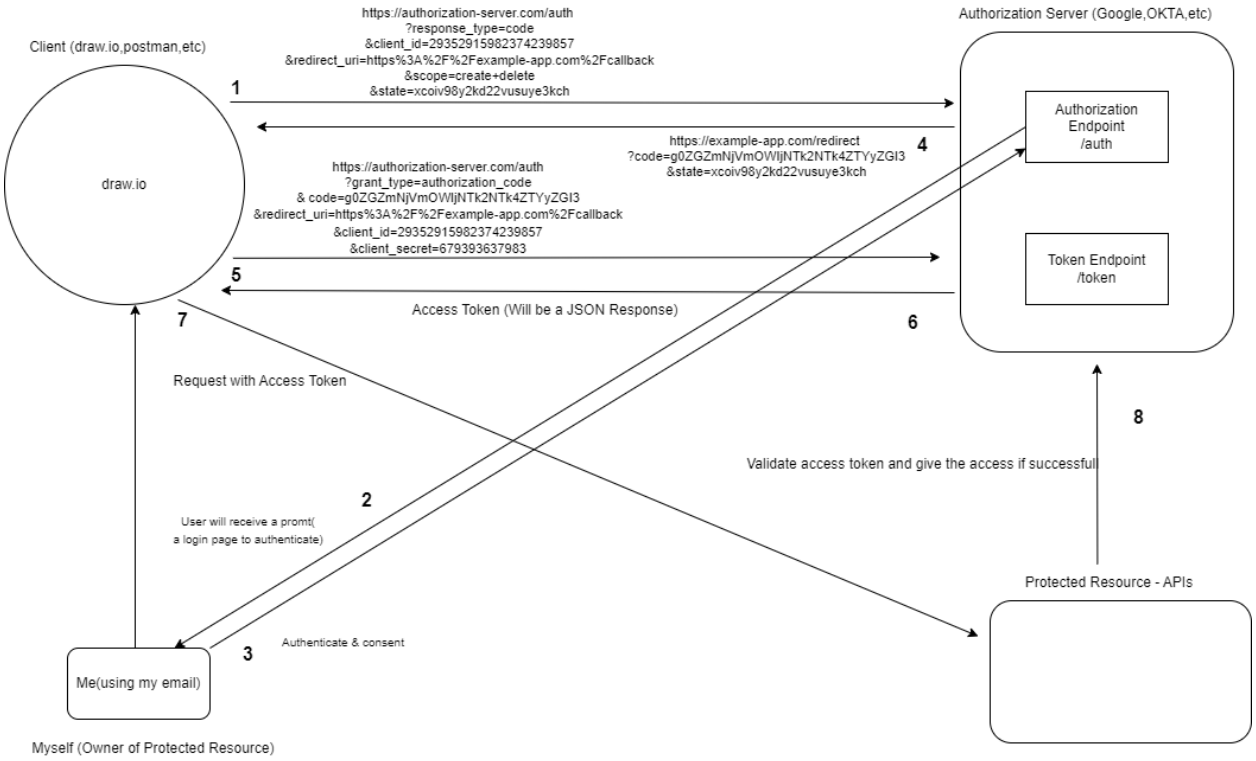
The token endpoint will verify all the parameters in the request, ensuring the code hasn't expired and that the client ID and secret match. If everything checks out, it will generate an access token and return it in the response!
Access token will be sent to redirect uri.

STEP-7

Client make request to protected resource with the access token

STEP-8

Protected resources validate it with authorization and provide access on successful validation.



REF =
https://developer.okta.com/blog/2018/04/10/oauth-authorization-code-grant-type#what-is-an-oauth-20-grant-type

*In the above flow, step 8 is a step which will be eliminated.
The elimination of the above step is done using JWT, which is given by the authorization server to client.When then used by client to get access to protected resource.

JWT

JSON web token (JWT), pronounced "jot", is an open standard (RFC 7519) that defines a compact and **self-contained** way for securely transmitting information between parties as a JSON object. Again, JWT is a standard, meaning that all JWTs are tokens, but not all tokens are JWTs.

Because of its relatively small size, a JWT can be sent through a URL, through a POST parameter, or inside an HTTP header, and it is transmitted quickly. A JWT contains all the required information about an entity to avoid querying a database more than once. The recipient of a JWT also does not need to call a server to validate the token.

Typically, a private key, or secret, is used by the issuer to sign the JWT. The receiver of the JWT will verify the signature to ensure that the token hasn't been altered after it was signed by the issuer. It is difficult for unauthenticated sources to guess the signing key and attempt to change the claims within the JWT.

Structure of JWT

- Header
- Payload
- Signature

*Note - Header & Payload are encoded not encrypted and anyone can see it so any sensitive information is not passed in claims in payload.
Signature is encrypted.
Signature provides integrity to jot.

Q)How is a signature created?

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

We can use HMAC or RSA.

Q)How does the protected/resource server know that the JWT is not manipulated and is sent by a genuine authorization server?
Ans)The third part of a JWT like hhhh.pppp.ssss is the signature. The signature is performed with a server private key over the header and payload (hhhh.pppp), and protects the content. If an attacker alters the content or the signature, the server will detect it verifying the signature and will reject the authentication.

The server knows that a JWT token sent by the client is valid by verifying its cryptographic signature using the public key of the issuer (authorization server). If the signature is valid, the server trusts that the token has not been tampered with and can proceed to process the token's claims.

CLIENT CREDENTIAL GRANT TYPE -

Here there is no resource owner and the client authenticates itself with the authorization server.In other words, the client credentials grant type is used by client applications to obtain an access token beyond the context of a user, for example, in machine-to-machine environments.

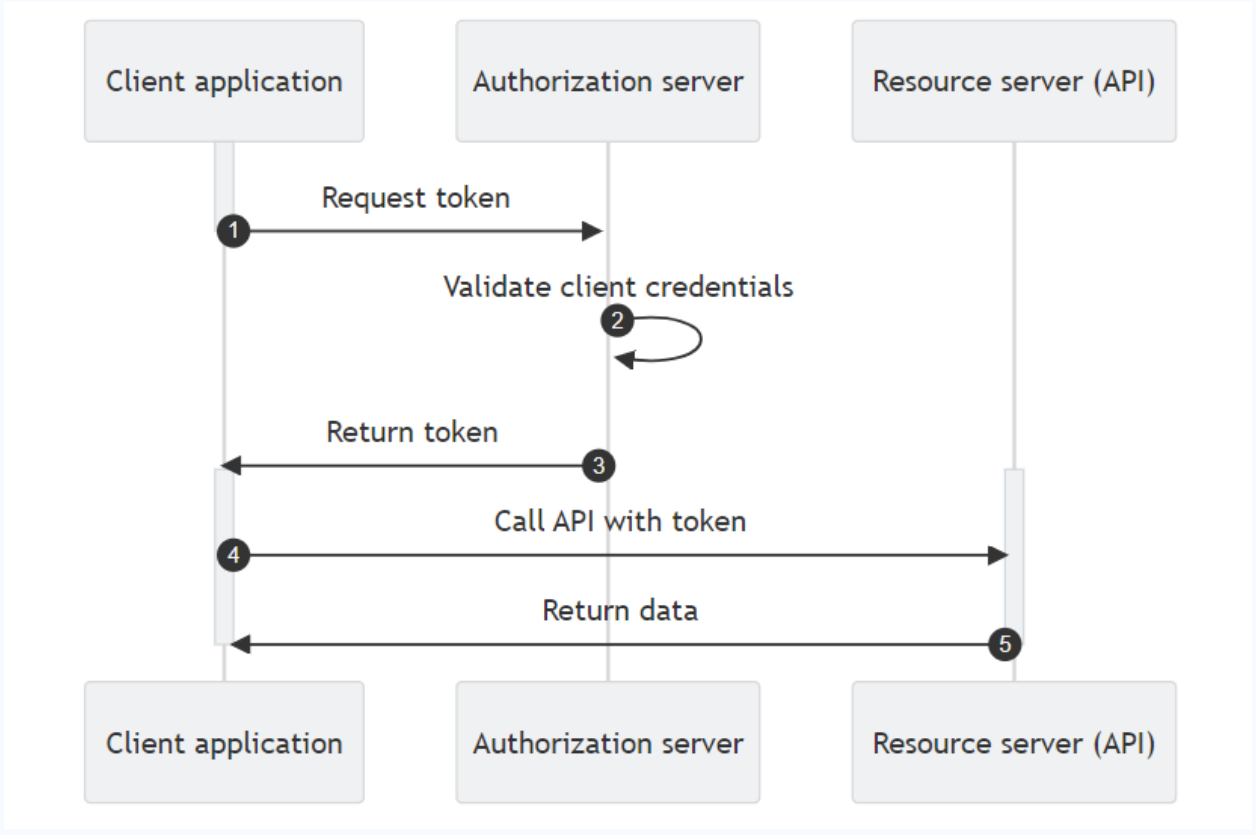
Client send 3 parameter to /token endpoint in Authorization server
`"grant_type=client_credentials&client_id=$CLIENT_ID&client_secret=$CLIENT_SECRET"`

Authorization server validates the client credentials received in the request.

Authorization server returns the token.

The client requests protected resources from the resource server and submits the token it received in the previous step.

The resource server validates the token and responds with the requested resources.



IMPLICIT GRANT TYPE -

Not recommended due to its security vulnerabilities.
The implicit grant type was created for public clients, like mobile applications or pure JavaScript front-end applications. It does not include client authentication, but, instead, the client receives access tokens as the result of authorization. The implicit grant type does not support granting refresh tokens.

Implicit Flow Threats

As there is no client authentication present in the implicit grant flow, the process relies on the presence of the resource owner and the registration of the redirection URI. Access tokens are embedded in the redirection URI and can become exposed to the resource owner or other applications that are used on the same device as the client.