Web Security and Access Management-2

**5 Stolen JWTs**

So far, we have discussed how JWTs are a secure way of exchanging information authentication.

Although JWT is a robust mechanism, it is still prone to attacks. In this lesson, we will discuss what happens if a JWT is stolen. We will also discuss how a hacker can make changes in a token and mislead us in believing that it is a valid token.

**What would happen if JWT is stolen**

If a hacker somehow gets access to our JWT, then there are two issues that we face:

**1. Hacker can view sensitive information available in the token**

As we discussed earlier, a JWT string is just base64 encoding of the header, payload, and signature. If a hacker gets access to the token, then they can decode it and see the information within the token. Thus, it is advisable for the tokens to not contain any sensitive information like passwords.

**2. The token can be used to access the application**

If your JWT is stolen or compromised, then the attacker has full access to your account. The attacker can send requests to applications, pretending to be you, and can make potentially harmful changes. One good thing is that the tokens expire after some time, so the hacker will only be able to use the token for a limited amount of time.

**Where to store your JWT**

As we know, a token is sent by the server which is stored on the browser for future requests. Now the question is, where should the token be stored on the browser? We have a couple of options:

**1. Web Storage (local storage or session storage)**

If tokens are stored inside local storage, they are accessible by any script inside your page and thus vulnerable to cross-site scripting (XSS) attacks. In XSS attacks, the hackers can inject a malicious JavaScript code on your page. When this code runs, it can access the content of your local storage. Therefore, it is advisable to not store any sensitive information in the web storage.

**2. Cookies**

When cookies are used with the HttpOnly flag, they are good candidates for storing JWTs as they are immune to XSS attacks. Additionally, we can use the Secure cookie flag to guarantee that the cookie is only sent over HTTPS.

However, cookies are also not completely safe as they are prone to CSRF attacks, as we have already discussed in previous lessons. Most modern web frameworks provide mechanisms to prevent CSRF attacks so we can consider storing our tokens in cookies.

**How to invalidate a JWT**

If our JWT is somehow compromised, then we need some mechanism to invalidate our token. In session-based authentication, we can just close the browser and the session is destroyed. Unfortunately, it is not that easy in case of token-based authentication.

One option that we have is that the server should change its secret key. However, this will invalidate the tokens for all the users. This approach should be used if the application owner thinks that the JWTs of a large number of users are stolen.

Another option is maintaining a blacklist of all the invalid tokens. If a client suspects that the token is stolen, then they can logout from the browser. On doing this, the token for that user will be deleted from the browser storage and will be added to the blacklist present on the server. When the hacker sends a request with the stolen JWT, the server will find it in the blacklist and throw an unauthorized error.

**6 Cryptographic Key Management**

Until now, we have not discussed a very important aspect of JWT, the key management.

Let us look at some questions that can be raised regarding keys.

1. Will the secret key always remain the same, or will it be changed after regular intervals?
2. If the secret key is changed, then what would happen to the tokens signed by older keys?
3. If we are using asymmetric signing and the private key is changed, how will we share the new public keys with all the applications?

We will answer each of these questions one by one.

**Will the secret keys rotate?**

Definitely yes! It is not safe or advisable to use the same key for a long time. If someone accidentally gets ahold of our applications key, then the attacker can send malicious requests to our server and we might not even know. If we rotate our keys regularly then we decrease the window of unauthorized access that an attacker gets.

Now the question is, how often should we change the keys? There is no definite answer to this, as it depends on the application. Some might change them every hour and others might keep the same key for a month. It depends on how many tokens you generate and how vulnerable your application is to attacks.

**What would happen to old JWT once a key is changed?**

Changing the secret key is easy, the hard part is validating the tokens that were signed by the previous key. We can’t just invalidate the old tokens every time we change the key. There is a way to handle this problem. What if while generating the token, we also mention the key that was used for signing inside the header of the token? When we receive this token, we can check the header to see what key was used to sign it.

I know what you are wondering: if we write the key in the token itself, then an attacker can easily read our token and will get to know about the key. Here comes the trick. We will not write the actual key in the token but only an identifier. We will maintain a list of keyIds and corresponding keys in our memory or DB. When we receive a token, we will fetch the keyId from the token. Then we will look up our memory or DB for the actual key.

The JWT header supports a kid parameter to hold a key identifier. It can take any value, as it is just used to identify the actual key.



**What would happen if the private key is changed in asymmetric signing?**

As we know that in the case of asymmetric signing, the private key is used to sign the token and it can be verified by all the applications that have the public key. If we keep rotating the private key, then the public key will also change. How will an application get to know which public key to use?

We can associate each public key with a keyId and provide this keyId in the kid parameter of the token header. But in that case, each application will have to maintain a set of public ids for lookup. This is not a feasible option.

We have a JSON Web Key specification that offers a way to represent cryptographic keys in a JSON format. A JWK can be included in a JWT token as a way to distribute the public key. We have two ways in which we can include the public keys in our token header:

**1. Directly embedding the public key**

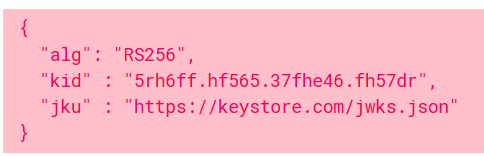
As the name suggests, public keys are public. So even if we add them into the token, it is not a security threat, as everyone can easily access the public key.

However, we don’t add the public key directly into the token. We can convert the public key into JSON format using any of the available libraries. A JSON object that represents a cryptographic key is called JWK.

A JWK can be included in the token header using jwk parameter.

**2. Embedding a URL which contains the key**

We can maintain the complete list of all the public cases at a particular location. Then in the token header, we can provide the URL of this location along with the key identifier. There are two claims available which help us in doing this. The first is kid, which we have discussed earlier also. It will contain the key identifier. The second claim is jku. It contains the URL of the location where all the keys are stored.



**7 Hacking JSON Web Tokens**

At this point, we have discussed the ins and outs of JWTs. We have seen how they are generated, validated, and how their keys are managed. We have also discussed what would happen if an attacker steals our JWT. But there is one thing which we have not discussed yet: is it possible for an attacker to create a JWT (without knowing your secret key or private key) and making you believe that this is a valid token?

In other words, is it possible for an attacker to change the data within a token, and have it still be validated by our server? Unfortunately, there are some ways through which an attacker can do this. Some of these issues have been caught already and fixed and some require extra caution from the token generator.

We will discuss each of these methods below:

**1) Brute Force Approach**

In symmetric signing, we use a secret key to sign the token. If an attacker gets our secret key, the attacker can change the data in the token, sign it again using the secret key, and send it with the request.

If an attacker has our valid JWT then the attacker can brute force various symmetric keys and compare the signature result to the known-valid signature. If there is a match, then the attacker has discovered the symmetric key and can modify and forge JWTs at will. There are plenty of libraries for doing this.

To save ourselves from a brute force attack, we should carefully select our secret key. It should not be too easy to guess.

**2) None Algorithm**

In the **alg** claim, we provide the algorithm that is used to sign or encrypt the token. Earlier this claim was allowed to take None as value. If a token has None value in **alg** claim, then it means that this token need not be validated.

Any attacker can create a token with **alg** claim as None and get access to our resources. This issue was fixed in 2015 but there still might be some libraries that allow None value in **alg** claim.

**3) Modify the algorithm RS256 to HS256**

We already know that HS256 uses a secret key to sign and validate the token. We also know that RS256 uses a private key to sign the token and public key to validate it. Now if an attacker has access to our token (which is signed using RS256) then the attacker can change the token data using the following steps:

1. Change the algorithm in **alg** parameter from RS256 to HS256.
2. Make changes in the payload.
3. Sign the token using the public key (assuming the attacker has access to the public key). Please note here that the public key cannot be used for signing. But, since the algorithm is changed to HS256, the public key will act as a secret key.

When this token will be received by the application, it will see that the algorithm is HS256. It will use the public key as a secret key and the malicious user will get access to the application.

**A4 OAuth**

**1 OAuth Introduction**

Before we dive into OAuth, I would first like to discuss a problem.

**Problem Statement:**

Consider we want to use a new app called **PicsArt**, which allows us to beautify our images. We just need to upload our images and this app gives us some options to edit our images.

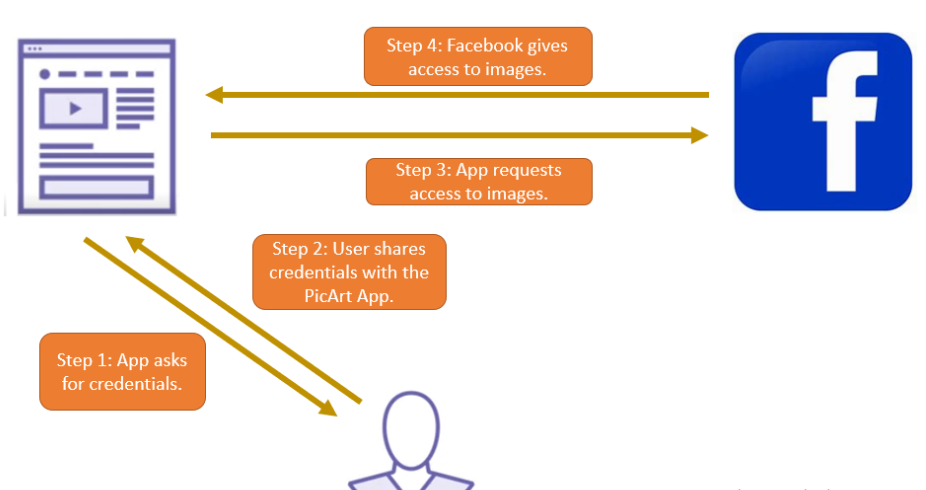
This app provides us a few methods to upload images, such as from the phone gallery or a direct upload from our social media accounts like Facebook or Instagram.

The problem is, the PicsArt app needs access to our Facebook account to access our images.

Here are a few methods to solve this problem:

In the first method, we can share the credentials with the client app which it can use to access our images

1. The PicsArt app will ask us to provide our Facebook credentials.
2. The app will use those credentials to access our images from Facebook.

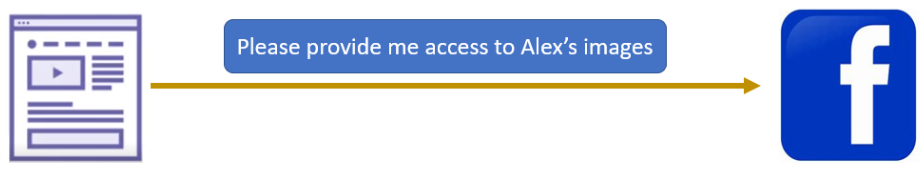
****

This is the most insecure way to give access to our resources, as we can’t trust any third-party app. There is no way for us to guarantee that the app will only access our images and will not post anything on Facebook. Also, what if we don’t want this app to access our images anymore? Will we change our credentials then?

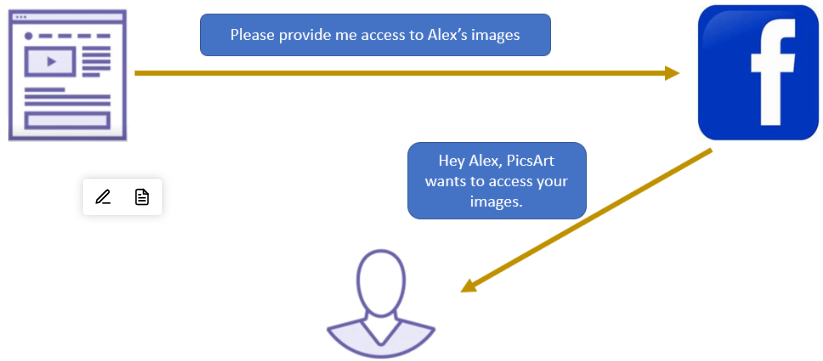
The second method solves this problem in a much secure way and forms the basis for OAuth.

Here is what the flow will look like:Consider a user Alex is using the PicsArt app.

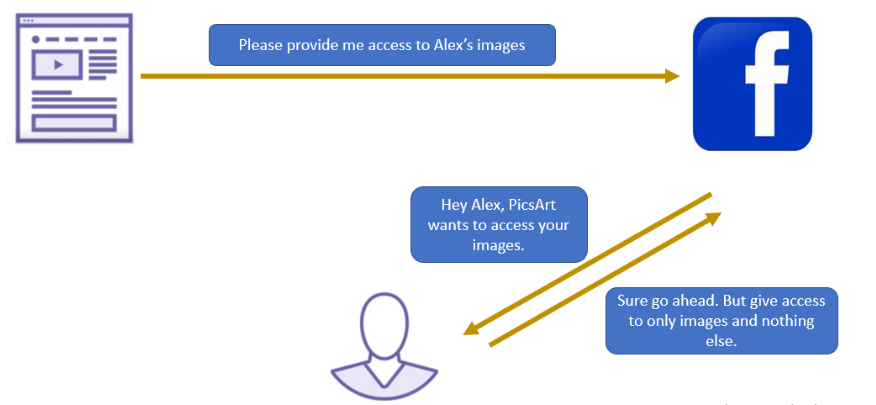
1. The PicsArt app will request Facebook to provide access to Alex’s images.



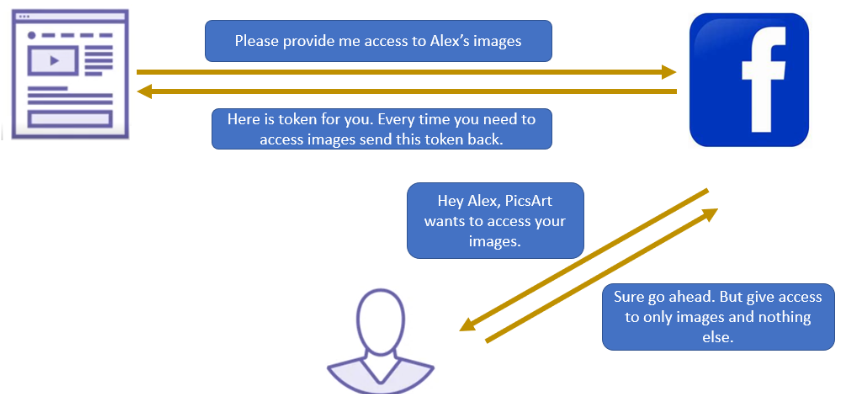
2. Facebook will confirm with Alex if it can provide access to his images to the PicsArt app.



3. Alex will allow Facebook to provide image access to PicsArt.



4. Facebook will share a token with the PicsArt app. Each time it needs to access the images, it can send the token to Facebook. This is a JWT token, which we discussed in the last chapter.



**What is OAuth 2.0?**

OAuth 2.0 is the industry-standard protocol for authorization. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

As the definition suggests, OAuth is a protocol that simplifies the process of providing authorization to clients to access secured resources.

**Ex:**

In other words, OAuth is an authorization framework that allows a client app (PicsArt App) to retrieve information from another system using a token which is valid for a limited time. The application users authorize the client app to retrieve information on their behalf.

Some developers have some confusion about whether OAuth is used for authentication or authorization or both. The answer is that OAuth is specifically meant for authorization. It has an extension called OpenId Connect which can be used for authentication. But OAuth should only be used for authorization.

The second example that we discussed above can be implemented through OAuth. OAuth defines various specs and flows using which clients can access the secured resources.

**Why shouldn’t OAuth be used for authentication?**

During authentication, a client app will need some user information. The client app can send the token to the resource server (Facebook in our example) to get the user information, but it is a bad idea because there is no standard way to send the user information back to the client. If applications use OAuth for authentication, then every implementation will be different, which will be a problem. This is the major problem using OAuth for authentication.

On the other hand, OpenId Connect defines a standard way to return user information. It defines a UserInfo endpoint which can be used to access user information.

**2 OAuth Terminologies**

Before we look into the exact details about how OAuth works and the various flows of OAuth, let’s go through all the OAuth related terminology. We will refer to the example that we discussed in the previous lesson to understand this technology.

**Resource owner**

The resource owner is the owner of the resource that is being accessed. When you log in to PicsArt App using your Facebook account, you are granting access to PicsArt to access your images. In this case, you are the resource owner.

**Client**

The client is an application that accesses protected resources on behalf of the resource owner. The client could be hosted on a server, desktop, mobile, or other devices. In our example, the PicsArt app is the client.

**Resource server**

The server that is hosting the protected resources and is capable of accepting and responding to requests by clients using access tokens. In our example, Facebook is the resource server.

**Authorization server**

The server which issues access tokens after successfully authenticating a client and resource owner is called authorization server. In our example, the Facebook server was issuing the access token. Normally, there is a separate server that does this task.

**Authorization grant**

An authorization grant is a credential representing the resource owner’s authorization (to access its protected resources) used by the client to obtain an access token. The OAuth specification defines four grant types, which we will discuss in the upcoming lessons.

**Authorization code**

In our example, we showed that the Facebook App shared a token with the client. In some OAuth flows, the authorization server does not give the access token directly. It first issues an authorization grant. The client then sends this grant with the client secret (more on this later) to the authorization server. After this, the authorization server gives access token to the client.

**Access token**

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

**Scope**

Scope defines the permissions of a token. It defines what resources can be accessed using a given access token. In our example, the client app wants to access only images, so the images are scope.

**3 Authorization Code Grant Type**

**What is grant type?**

In OAuth 2.0, the term grant type refers to the way an application gets an access token. Each grant type is optimized for a particular use, whether that’s a web app, a native app, a device without the ability to launch a web browser, or server-to-server applications.

In this lesson, we will look at the Authorization Code grant type.

**Authorization Code grant type**

The Authorization Code grant type is the most commonly used OAuth 2.0 grant type. It is used by both web apps and native apps to get an access token from the authorization server once the user has authorized. The Authorization Code flow is most suitable for websites and mobile apps that have a backend.

This type has the extra step of exchanging the authorization code for the access token. The exchange of authorization code for the access token takes place in the back channel. Due to this feature, it provides an additional layer of security.

**Authorization Code grant type Working**

Now we will look at the detailed working of the Authorization Code grant type.

**Step 1 => Authorization request**

In the first step, the client app (PicsArt) redirects the resource owner (the user) to the authorization server’s authorization endpoint. The app sends some query parameters which help the authorization server in identifying the client app and its intent.

The query parameters sent with the request are:

* **response\_type:** This parameter defines what is the type of response that is expected. In this flow it will be code.
* **client\_id:** This parameter defines the id of the client that needs access to the resource. In our example, it will be the client id of the PicsArt app.

**Note**:

You might be wondering: where does this client id come from? Every client app first needs to register with an authorization server. When a client gets registered with an authorization server, it is provided a unique client\_id and client\_secret, which it uses to identify itself to the authorization server.

* **redirect\_uri:** This is the URI where the authorization server redirect to once it has finished interacting with the resource owner.
* **scope**: This parameter defines the resources to which access is being requested. This is not a mandatory parameter, and if it is not provided the authorization server provides access to default resources already defined for this client.
* **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.

The complete request looks like this:

https://authorization.server.dummy.com/authorize

?response\_type=code

&client\_id=12345

&redirect\_uri=https://client.dummy.com/callback

&scope=images\_read

&state=abcde

The client opens this URL in a browser. The authorization server will present them with a prompt asking if they would like to authorize this application’s request.

**Step 2 => Authorization response**

If the user provides consent to the authorization server, then the authorization server redirects the browser to the **redirect\_uri** which was provided in the request. The authorization server sends the following parameters in the response:

* **code**: This is the authorization code generated by the authorization server. This code is relatively short-lived and is bound to **client\_id, resource\_owner, scopes** and **redirect\_uri.**
* **state**: This is the same string that was passed with the request.

The response URL looks like:

https://authorization.server.dummy.com/callback

?code=hhdf6hsbhjG66hgtgfGGHJGCHJ

&state=abcde

**Step 3 => Token request**

Once the client has received the authorization code, it can exchange the code with the access token by sending a post request to the token endpoint.

Following parameters are sent with the request to get the token:

* **grant\_type**: For this flow, the value will be **authorization\_code**. This tells the token endpoint that the client is using Authorization Code grant type.
* **code**: This is the code that was received in Authorization Response.
* **client\_id**: The client id of the client.
* **client\_secret**: This is the client secret. This field is to verify that the request is sent by a validly registered client and not by an attacker posing as the client.

POST /token/endpoint HTTP/1.1

Host: authserver.dummy.com

grant\_type=authorization\_code

&code=hhdf6hsbhjG66hgtgfGGHJGCHJ

&client\_id=12345

&client\_secret=gh5Gdkj743HFG45udbfGfs

**Step 4 => Token response**

The token endpoint verifies all the parameters in the request. It verifies that the code has not expired and the client id and client secret matches. If everything looks good, then the access token is returned.

HTTP/1.1 200 OK

Content-Type: application/json

{

"access\_token":"YT3774ghsghdj6t4GJT5hd",

"token\_type":"bearer",

"expires\_in":3600,

"refresh\_token":"YT768475hjsdbhdgby6434hdh",

"scope":"images\_read"

}

You might be thinking, “Why are there two steps to get the access token? Why do we need to first get the authorization code and then exchange it with the access token?”

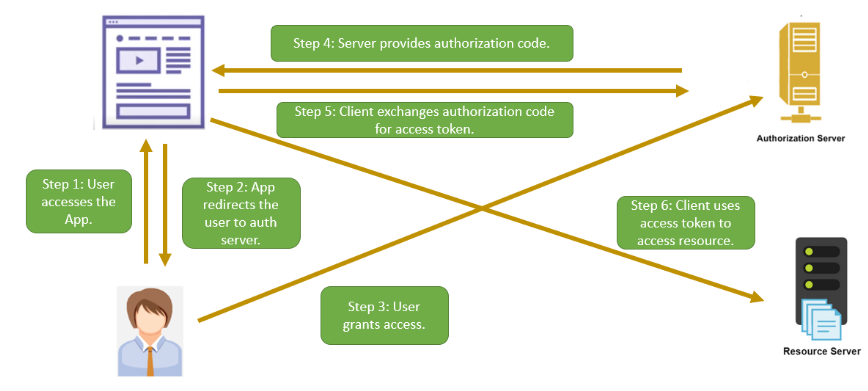
To understand this you must first know about two concepts:

* **Front channel**: Less secure browser/mobile app to the server channel.
* **Back channel**: Highly secure server to server communication channel.

It is not safe to share the client secret and get the access token on the front channel.

Therefore, we first fetch the authorization code using the front channel and then request the access token using the back channel.

Below is the pictorial representation of this flow.



**4 Implicit Grant Type**

**Implicit grant flow**

The Implicit grant type is designed for single-page JavaScript apps that do not have a backend. In the previous lesson, we discussed the Authorization Code grant flow, in which the client app used the client\_secret and authorization code to get the access code.

The problem with JavaScript apps (without a backend) is that they have no way to store client secrets. Storing the client secret in the JavaScript code is not as safe, because anyone can access it. Therefore, we use the Implicit flow for these apps. In Implicit flow, the authorization server directly returns the access token instead of returning the code.

This flow type should be used only if there is no alternative option because it is not safe. The exchange of token happens at the front end and an attacker can access the token.

**Implicit grant type working**

Now we will look at the detailed working of implicit grant type.

**Step 1 => Authorization Request**

This step is similar to the first step in the previous flow. The only difference is that the resource\_type parameter will contain a token as value.

The query parameters sent with the request are:

1. response\_type: Since we are requesting for the access token directly, this parameter will contain a token.
2. client\_id: The client id of the app.
3. redirect\_uri: This is the uri to which the authorization server redirects once it has finished interacting with the resource owner.
4. scope: The optional parameter defining the resources being requested.
5. state: This parameter is optional in this flow as well.

The complete request looks like this:

https://authorization.server.dummy.com/authorize

?response\_type=token

&client\_id=12345

&redirect\_uri=https://client.dummy.com/callback

&scope=images.read

&state=abcde

Step 2 => Authorization Response#

Once the user authenticates and provides permissions to the authorization server, it will redirect the browser to **redirect\_url** specified by the client app in the request. The authorization server will then add a token and state to the fragment part of the URL.

Please note that the client will receive an access token, whereas in the previous flow it was receiving authorization code.

The token is returned in the URL fragment instead of in the query string. The reason for this is to ensure that the app will be able to access the value from the URL.

Here is what the response looks like:

https://authorization.server.dummy.com/callback

#access\_token=hhdf6hsbhjG66hgtgfGGHJGCHJ

&token\_type=bearer

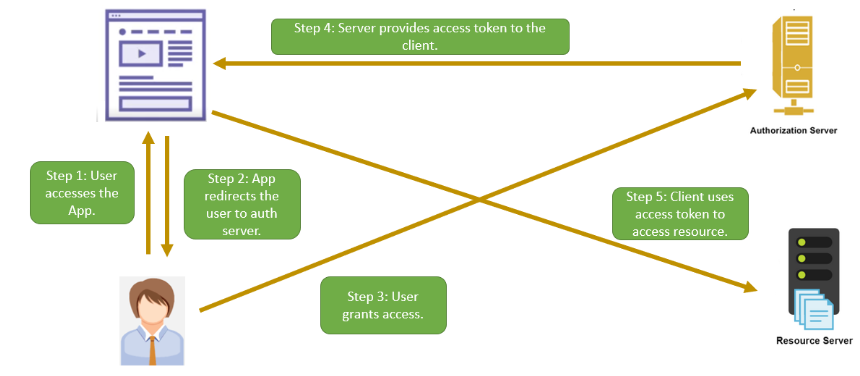
&expires\_in=500

&state=abcde

The expiration time of this token is kept a little less because this flow is not safe.

**Note**: This grant does not return a refresh token because the browser has no means of keeping it private.

Here is a pictorial representation of this flow:



**5 Client Credentials Grant Type**

**Client Credentials Grant Type**

This grant type is used for machine to machine authorization. There is no user involved in this flow. Suppose we have an application that follows the microservices architecture. The application is divided into small parts and each part is deployed on a separate server.

If one internal server needs to access some data from the other server, then they can use the client credentials grant type.

**Client Credentials grant type working**

Now, we will look at the detailed working of Client Credentials grant type.

**Step 1 => Token request**

In this flow, there is a direct token request. Since no user is involved, the client directly sends an HTTP Post request to the authorization server.

The query parameters sent with the request are:

* grant\_type: Since we are requesting the access token using the client credentials, this parameter will contain client\_credentials.
* client\_id: The client id of the app.
* client\_secret: The client secret of the app.
* scope: The optional parameter defining the resources being requested.

The complete request looks like this:

POST /token/endpoint HTTP/1.1

Host: authserver.dummy.com

grant\_type=client\_credentials

&client\_id=12345

&client\_secret=gh5Gdkj743HFG45udbfGfs

&scope=images\_read

**Step 2 => Token response**

If the client credentials are valid, a token is returned by the authorization server.

The response from the server is as shown below:

HTTP/1.1 200 OK

Content-Type: application/json

{

"access\_token":"YT3774ghsghdj6t4GJT5hd",

"token\_type":"bearer",

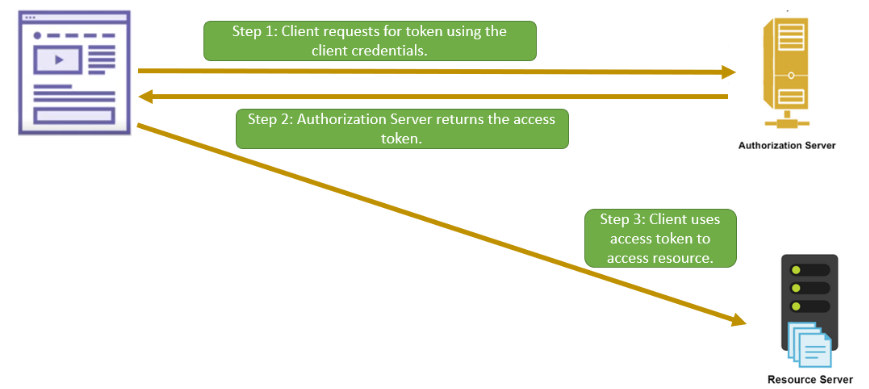
"expires\_in":3600,

"refresh\_token":"YT768475hjsdbhdgby6434hdh",

"scope":"images\_read"

}

Here is a pictorial representation of how this flow works.



**6 Resource Owner Credentials Grant**

The Resource Owner password credentials grant type is used in cases where the resource owner trusts the client and is ready to share its credentials with the client. The authorization server should take special care when enabling this grant type and only allow it when other flows are not viable.

This flow was introduced to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token. Today, there is no case in which this flow should be used, as it is very insecure.

**Note**:

In HTTP Basic authentication, the server requests the client to present a username and password combination as part of the HTTP Basic challenge-response mechanism. With HTTP Basic Authentication, the client’s username and password are concatenated, base64-encoded, and passed in the Authorization HTTP header. The server can then authenticate this user against a user profile stored in the server’s local repository, a database, or an LDAP directory.

The Resource Owner password credentials grant type is used in cases where the resource owner trusts the client and is ready to share its credentials with the client. The authorization server should take special care when enabling this grant type and only allow it when other flows are not viable.

This flow was introduced to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token. Today, there is no case in which this flow should be used, as it is very insecure.

In HTTP Basic authentication, the server requests the client to present a username and password combination as part of the HTTP Basic challenge-response mechanism. With HTTP Basic Authentication, the client’s username and password are concatenated, base64-encoded, and passed in the Authorization HTTP header. The server can then authenticate this user against a user profile stored in the server’s local repository, a database, or an LDAP directory.

**Resource Owner Credentials grant working**

**Step 1 => Token request**

In this flow, the client collects the credentials from the user and sends a POST request to the authorization server.

The query parameters sent with the request are:

1. response\_type: The value for this flow is “password”.
2. client\_id: The client id.
3. client\_secret: The client secret.
4. username: The username of the user.
5. password: The user password.
6. scope: Defines the resources to be accessed.

The requests sent looks as below:

POST /token/endpoint HTTP/1.1

Host: authserver.dummy.com

grant\_type=password

&client\_id=12345

&client\_secret=gh5Gdkj743HFG45udbfGfs

&username=Jone@xyz.com

&password=abcde

&scope=images\_read

**Step 2 => Token response**

If the client credentials and user credentials are valid, then a token is returned by the authorization server.

The response from the server is as shown below:

HTTP/1.1 200 OK

Content-Type: application/json

{

"access\_token":"YT3774ghsghdj6t4GJT5hd",

"token\_type":"bearer",

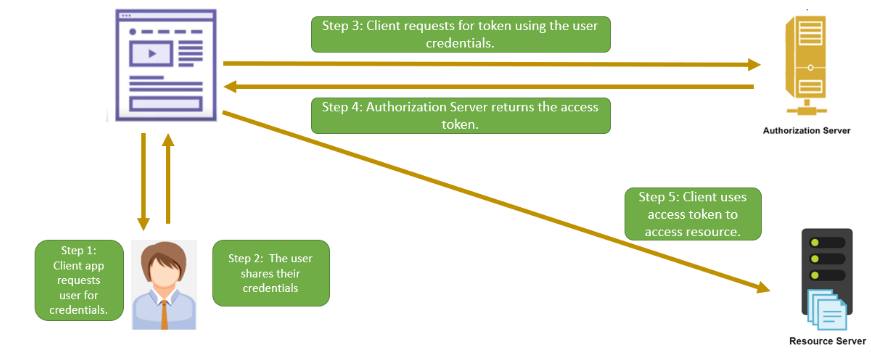
"expires\_in":3600,

"refresh\_token":"YT768475hjsdbhdgby6434hdh",

"scope":"images\_read"

}

Here is the pictorial representation of this flow.



**7 Refresh Token Grant**

**Refresh Token Grant**

As we have seen earlier, the access token is valid only for a certain time frame and it expires after the frame has ended. The reason for this is to ensure security because if the access token gets stolen, the attacker can use it until it expires (which takes a long time) or it is blacklisted on the server.

If the client asks the user to authenticate and authorize every time the access token expires then it will be very frustrating. To avoid this, the client app can use a refresh token. A refresh token is a token that can be used to get the access token when it expires. This should be kept highly confidential, because if an attacker gains access to refresh token then the attacker gets unlimited access to the resources.

There are different settings for refresh tokens which are defined by the authorization server. Some tokens are single-use only and others can be used multiple times. Also, the refresh token expires after some time, but this time is much longer than the expiration time of the access token.

Refresh tokens are usually subject to strict storage requirements to ensure they are not leaked. They can also be blacklisted by the authorization server.

**Refresh token grant working**

**Step 1 => Refresh token request**

If the client already has a refresh token, then it can exchange the refresh token for a new access token by making the HTTP POST request to the authorization server.

The following parameters will be sent with the request:

1. grant\_type: The value for this flow is refresh\_token.
2. refresh\_token: The refresh token.
3. client\_id: The client id.
4. client\_secret: The client secret.
5. The complete request looks like:

POST /oauth/v2/accessToken HTTP/1.1

Host: authserver.dummy.com

Content-Type: application/x-www-form-urlencoded

grant\_type=refresh\_token

&refresh\_token=k3hjJE5x1lZh-zjU-02w8EJW6l2jnuP8F1uXMgkm8nzjPfnaJR

&client\_id=12345

&client\_secret=hhdHFgggdFGfhd

**Step 2 => Refresh token response**

If the client credentials are valid and the refresh token has not expired, then an access token is returned.

The response from the authorization server is as shown below:

HTTP/1.1 200 OK

Content-Type: application/json

{

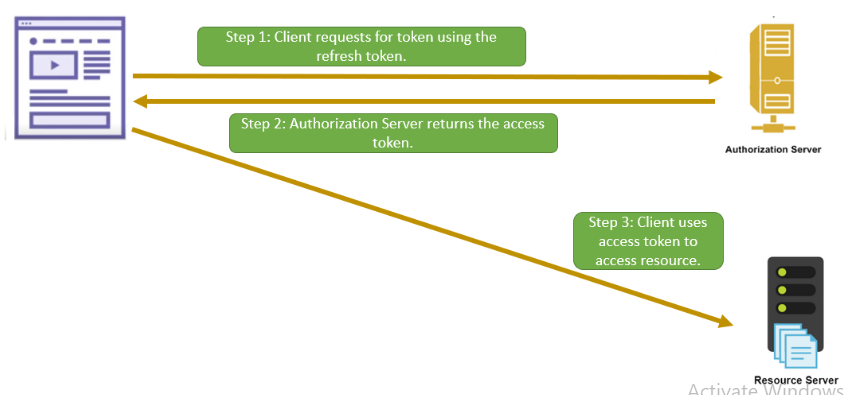
"access\_token": "dgfYTGFVygPtyqytVfyGFtyF",

"expires\_in": 3600,

"refresh\_token": "GHcdkHJjcbhhBHJGbhjfgkHghB",

"refresh\_token\_expires\_in": 500000

}



Which flows allow refresh token grants?#

Out of all the four flows that we discussed earlier, two do not allow refresh token grants:

Implicit grant type - In this grant type, the client is not able to store the client\_secret. We can’t expect that it will be able to store the refresh token, which is highly confidential. Therefore, this flow doesn’t support refresh token grant flow.

Client credentials grant type - In this grant type there is no user, so the need for refresh token doesn’t arise. The client can just send its client id and client secret to get the access token.

**A5 OpenID Connect**

**1 OpenID Connect Introduction**

OAuth 2.0 is designed only for authorization. It is used for granting access to data and features from one application to another. In OAuth, the client is given a token which it uses to access the data on the resource server, but it doesn’t get to know anything about the user. OAuth was used for authentication as well, but since it was not designed for authentication it was extended further to support authentication.

**OpenID Connect** is an extension of OAuth. It is a thin layer above OAuth which adds support for authentication.

You may have seen that when you try to login to an app, then the app can prompt you to authenticate using your Facebook or Google account. In this case, the app is probably using OpenID Connect. OpenID Connect allows a range of clients, including web-based, mobile, and JavaScript clients, to request and receive information about authenticated sessions and end-users.

**Local user authentication**

Let’s suppose we are not using OpenID Connect or any Identity Provider (discussed in the next section) for authentication. In that case, our application will have to maintain a local database in which we will store user information and credentials. This seems like an easy solution but there are a few problems:

If our organization has lots of applications, then maintaining a database for each application is a tedious task. It requires resources and staff to manage.

Users find the task of signing up as cumbersome. First, they need to remember so many passwords, and second, it is likely that they’ll use the same password everywhere. If that password gets compromised, all the applications that the user uses with that password will also become compromised.

**Local user authentication**

Let’s suppose we are not using OpenID Connect or any Identity Provider (discussed in the next section) for authentication. In that case, our application will have to maintain a local database in which we will store user information and credentials. This seems like an easy solution but there are a few problems:

If our organization has lots of applications, then maintaining a database for each application is a tedious task. It requires resources and staff to manage.

Users find the task of signing up as cumbersome. First, they need to remember so many passwords, and second, it is likely that they’ll use the same password everywhere. If that password gets compromised, all the applications that the user uses with that password will also become compromised.

**Identity Providers**

To solve the problem of local authentication, Identity Providers came in the picture. As the name suggests, Identity Providers take care of your authentication needs while you focus on your main business. They provide the identity of the user so that organizations can directly onboard the user without asking for any details.

It is a win-win situation for both the user and the organization. The organization is not required to store the personal data of its user, and the users are saved from creating an account each time they need to use some application.

**What is OpenID Connect?**

There should be a standard way for client apps to communicate with the Identity Providers. OpenID Connect is one of those standards that defines how a client and Identity Provider should interact.

The OpenID had two earlier versions, **OpenID 1.0** and **OpenID 2.0**. Its third iteration is called OpenID Connect. OpenID Connect is the most widely used, due to its simplicity and usability. Since the information is passed in the form of JWT, OpenID connect can be used across different implementations.

**Participants in OpenID Connect**

1. **End User**: End User refers to the entity for which the client is requesting identity information. This is called Resource Owner in OAuth.
2. **Relying Party**: This is the party that relies on the authorization server to provide the identity of the End User. This is called client in OAuth.
3. **Identity Provider**: This is a server that provides identity information about the End User. This is called Authorization Server in OAuth.

**2 OpenId Connect Terminologies**

**Identity token**

While discussing **OAuth**, we discussed the **authorization code** and **access token**. In the case of **OpenId** Connect, there is one more token that we can request. This token is called the identity token, which encodes the user’s authentication information.

In contrast to access tokens, which are only intended to be understood by the resource server, ID tokens are intended to be understood by the client application. The ID token contains the **user information in JSON format**. The JSON is wrapped into a JWT.

When a client receives the identity token, it should validate it first. The client must validate the following fields:

1. iss - Client must validate that the issuer of this token is the Authorization Server.
2. aud - Client must validate that the token is meant for the client itself.
3. exp - Client must validate that the token is not expired.

Here is some sample user information in the form of JSON present in an identity token.

{

"iss": "https://server.example.com",

"sub": "24400320",

"aud": "s6BhdRkqt3",

"nonce": "n-0S6\_WzA2Mj",

"exp": 1311281970,

"iat": 1311280970,

"auth\_time": 1311280969,

"acr": "urn:mace:incommon:iap:silver"

}

Let’s look at what these values mean:

**iat** - The iat claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. Its value must be a number containing a NumericDate value.

**auth\_time** - Time when the End-User authentication occurred. Its value is a JSON number representing the number of seconds from 1970–01–01T0:0:0Z as measured in UTC until the date/time.

**nonce** - ID token requests may come with a nonce request parameter to protect from replay attacks. When the request parameter is included, the server will embed a nonce claim in the issued ID token with the same value of the request parameter.

The Identity token contains only basic information about the user. To get the complete user information, the client must send the access token (please note access token should be sent not identity token) to UserInfo endpoint.

**Scope**

The Identity Provider may have a lot of information about a user. Some of this information can be shared freely with the client apps and some information might be shared under special circumstances.

The client app must also tell the Identity Provider about what user information it is looking for. This information can be provided through the scope field. There are four types of scopes defined in OpenID Connect. Each of these scopes defines certain attributes.

When a particular scope is requested, then all the attributes defined under that scope are returned.

|  |  |
| --- | --- |
| **Scope** | **Purpose** |
| email | Fetch the user’s email information. |
| phone | Fetch the user’s phone details. |
| profile | Fetch the user’s default information. |
| address | Fetch the user’s address. |

Please note that there is one more scope value, i.e. openid. This value is mandatory if the client app needs an identity token in the response.

**Claims**

Simply put, claims are **name/value** pairs that contain information about a user. For example, a family\_name claim is used to send the family name of a user. The benefit of claims is that each client knows what claims to look for to get particular user information. The client app can either request the category of a claim by providing the scope field or it can request individual claims using the optional claims request parameter.

Below is the list of claims which are divided amongst scopes.

|  |  |
| --- | --- |
| **Scope** | **Claim** |
| email | email, email\_verified |
| phone | phone\_number, phone\_number\_verified |
| profile | name, family\_name, given\_name, middle\_name, nickname, preferred\_username, profile picture, website, gender, birthdate, zoneinfo, locale, updated\_at |
| address | address |

**Endpoints**

The authorization server defines some endpoints that are used by the client to request some data.

The core endpoints are defined below:

**1. Authorization endpoint**

We have discussed this endpoint in OAuth as well. This endpoint returns the authorization code after the user authenticates and provides their consent to the authorization server (called the Identity Provider in case of OpenID).

**2. Token endpoint**

This endpoint is used to exchange an authorization token with an access token and identity token. The identity token contains some basic user information. More on this later.

In the case of OAuth flow, we used to only get the access token. Now, we are getting an additional identity token which contains basic user information. To get this additional token, some changes are made in the request. We will discuss this in the next lesson.

**3. UserInfo endpoint**

The UserInfo endpoint is used by the client to request user profile information that was previously granted access to. To access this information, the client needs to send an access token with the request. The user fields are returned in the form of JSON which may or may not be encapsulated in a JWT.

Following is a sample response from this endpoint:

HTTP/1.1 200 OK

Content-Type: application/json

{

"sub" : "Joe",

"email" : "Joe@dummy.net",

"email\_verified" : true,

"name" : "Joe Frank"

}

**3 Authorization Code Flow for Authentication**

**The Authorization code flow** for OpenID Connect is similar to the Authorization Code Flow that we discussed in the OAuth 2.0 chapter. The only difference is the change in the value of the scope field. It must contain openid as one of the values, followed by other scope values based on what type of user data the client wants.

**There are two questions that can be raised:**

1. What would happen if the client does not provide an openid in the scope field while sending a request to the authorization server?

The answer is that in this case, the flow will work as a normal authorization flow.

The client app will not get access to the user information as it will not receive the

identity token.

1. Can user information be fetched from the UserInfo endpoint by sending the access token in the request even if openid was not provided in the scope field when an access token was requested?

The answer is NO. When we send a request to the token endpoint to fetch the access token, then we must send openid in the scope field. We must also send other scope values like email or address if we want to get this information. The access token that is returned is based on the scope values that were sent with the request. When we hit the UserInfo endpoint, then only that user information is returned which the access token is authorized to get.

**Note:**

Let’s say that while sending a request to the token endpoint, the scope value is “openid email”. The client sends this request and gets an access token. If the client sends this access token to the UserInfo endpoint, it will get only email information. It will not get an address or any other information.

**Authorization code flow types**

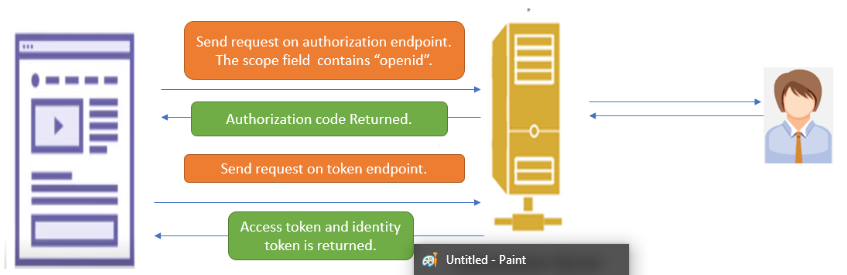
There is only one type of Authorization flow. In this flow, the response\_type is sent as code.

What tokens the client will get is based on what is being sent in the scope attribute.

1 If the scope contains openid.



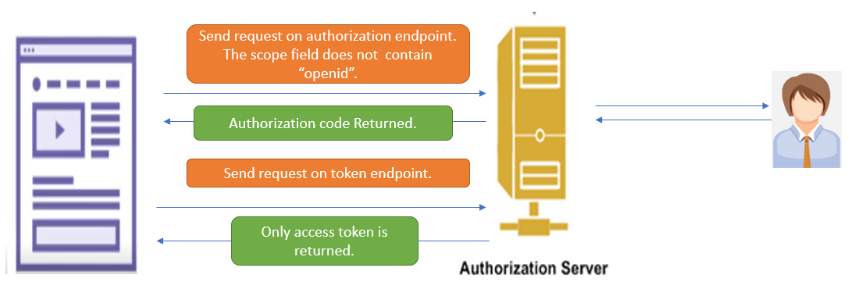
Here is the flow for this case.



2 If the scope does not contain openid.



Here is the flow for this case.



**4 Implicit Code Flow for Authentication**

This flow is also similar to the Implicit grant type discussed in the OAuth chapter. This flow is used for single-page JavaScript apps or those apps which do not have a backend.

In Implicit flow, the response\_type field can either take token or id\_token or token\_id\_token as value. This leads to some interesting cases depending upon what is provided in the scope field.

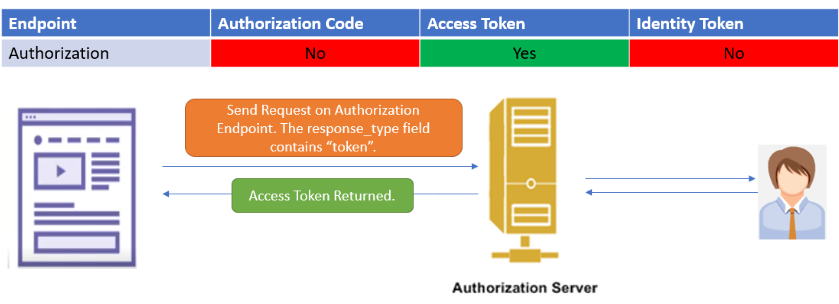
I**mplicit flow types**

There are different implicit flows for different values of the response\_type and scope field.

**1. response\_type=token**

When the value of response\_type is token, then only the access token is returned. Even if openid is included in the scope request parameter, an ID token is not issued.

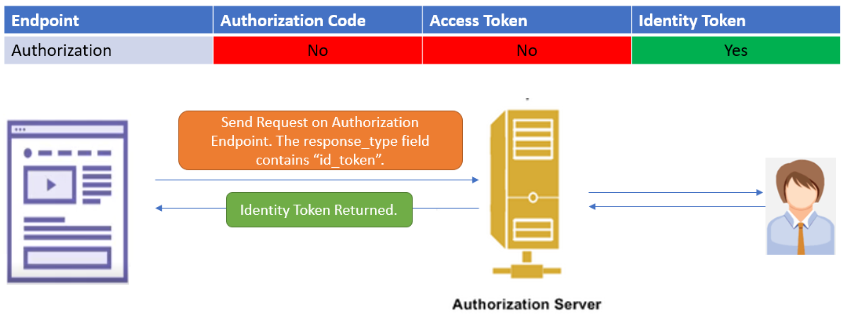
This flow uses the authorization endpoint only. It does not use the token endpoint.



**2. response\_type=id\_token**

When the value of response\_type is id\_token, then only the identity token is returned. The scope field must contain openid.

This flow uses the authorization endpoint only. It does not use the token endpoint.

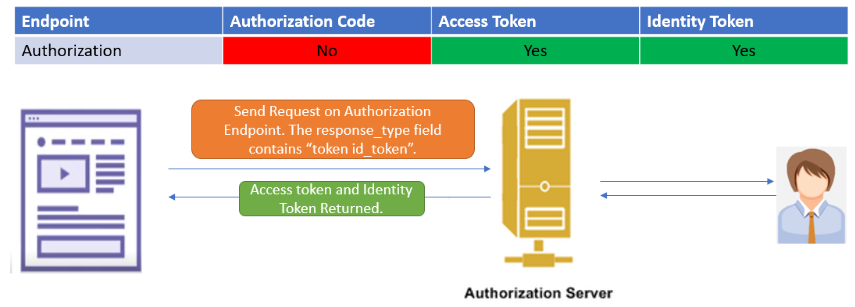


**3. response\_type="token id\_token"**

When the value of response\_type is token id\_token, then both access token and identity token is returned. The value of the scope field must contain openid, otherwise the identity token will not be returned.

When an access token is issued together with an ID token from the authorization endpoint, the hash value of the access token calculated in a certain way has to be embedded in the ID token.

This flow uses the authorization endpoint only. It does not use the token endpoint.



**5 Hybrid Code Flow for Authentication**

As the name suggests, this flow is a mix of Authorization code flow and Implicit code flow.

In **Authorization flow**, we first get authorization token from authorization endpoint and then get the access token and identity token from the token endpoint. This takes some time as two server calls are needed.

In the **implicit flow**, we get the access token and identity token from the authorization endpoint. This is faster but is not secure.

In the **hybrid flow**, the client gets immediate access to the identity token from the authorization endpoint itself. The client also gets the authorization code from the authorization endpoint. Later, it fetches the access token from the token endpoint which can be used to get further user info.

**Hybrid flow type**

In hybrid flows, the response\_type field should contain code as one of the values and token or id\_token or token+id\_token as the other value.

**1. response\_type="code id\_token"**

When the value of response\_type is code id\_token, an authorization code and an ID token are issued from the authorization endpoint, and an access token and ID token are issued from the token endpoint.



Both the authorization endpoint and the token endpoint issue an ID token, but the contents of the ID tokens are not always the same. It is possible that the ID token issued by the authorization server has fewer claims due to security reasons.

When an authorization code is issued together with an ID token from the authorization endpoint, the hash value of the authorization code calculated in a certain way has to be embedded in the ID token.

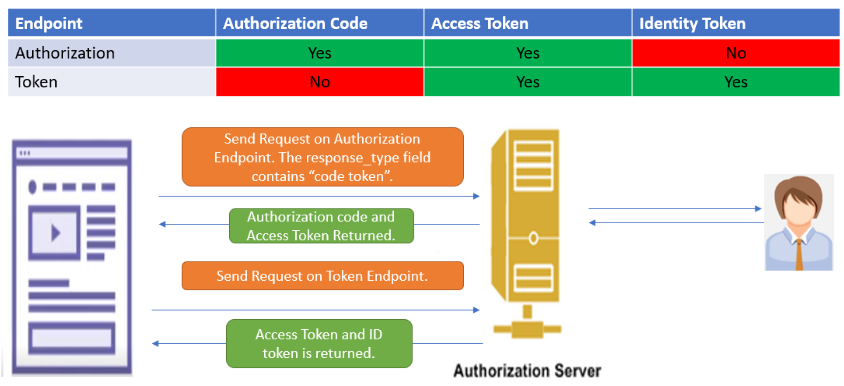
**2. response\_type=“code token”**

When the value of response\_type is code token, an authorization code and an access token are issued from the authorization endpoint, and an access token is issued from the token endpoint.

The token endpoint also issues the ID token if openid is included in the scope request parameter.

The access token issued by authorization endpoint and token endpoint may or may not be the same. As per OpenID Connect specification, different Access Tokens might be returned due to the different security characteristics of the two endpoints and different lifetimes and the access to resources.

If scope contains openId



**3. response\_type=“code token id\_token”**

When the value of response\_type is code id\_token token, an authorization code, an access token, and an ID token are issued from the authorization endpoint, and an access token and an ID token are issued from the token endpoint.

When an ID token is issued from the authorization endpoint, the hash value of the access token has to be embedded in the ID token if an access token is also issued. Similarly, the hash value of the authorization code has to be embedded in the ID token if an authorization code is also issued.

