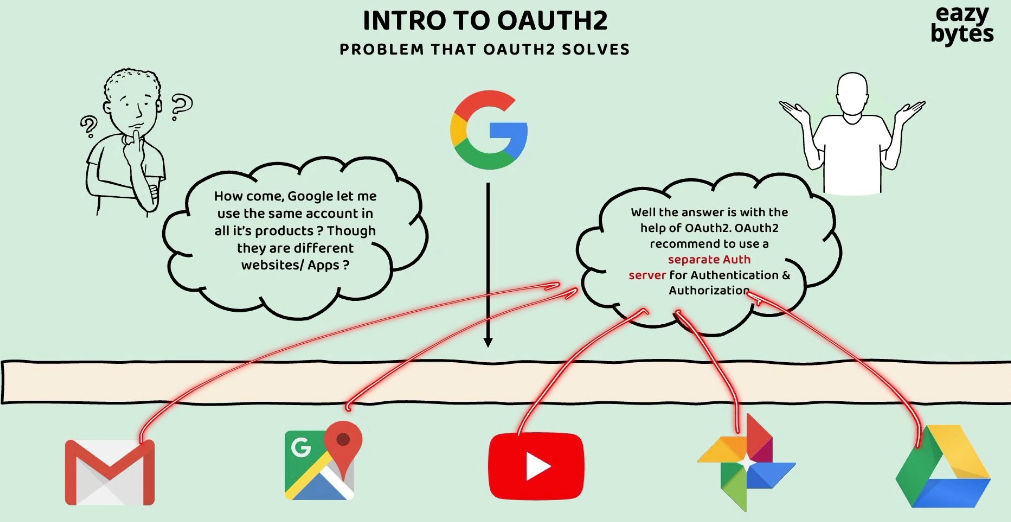
**12. OAUTH2 and openid connect**

what are the most common problems that theis OAuth2 framework

is trying to solve inside the industry.



We all use this various Google products. Google has primarily Gmail as a product.

They also have Google Maps, YouTube, Google Photo, Google Drive.

So similarly, there are many other applications which are developed and maintained by Google or Alphabet.

All these applications of Google, they have their own separate code base.

All these web applications are mobile applications. They're going to be deployed

in different, different servers. But have you anytime wondered how all these Google products, they allow the same credentials as an input?

We as an end user, whenever we want to use any Google product, we need to create our own account with the help of Gmail. So the same Gmail credentials you can use

to authenticate in all the remaining applications.

For example, if I want to use YouTube to see the videos or to upload some videos,

I can log in into these YouTube mobile application or web application with the Gmail credentials that I have created previously. The same applies for the remaining Google products.

So how this is possible is, behind the scenes, Google as an organization,

it is following the OAuth2 standard or OAuth2 framework.

So whenever someone is following the OAuth2, it is going to recommend them

to use a separate authorization server which is responsible for storing the end user credentials, and the same Auth server is also responsible for end user authentication and authorization.

So initially, when an end user register into the Gmail account, a separate Auth server

will handle the registration process and will store the end user credentials.

The same Auth server will be used by all the applications inside Google

to authenticate the user and to perform authorization of the end user.

This way, we as an end users of Google products, we don't have to create

different, different accounts in different products and we don't have to maintain different credentials for different products of Google.

So the responsibility of this Auth server is, it is going to perform the authentication of the end user and it is going to issue the tokens during the login operation.

The same token can be used with the other applications as well.

For example, inside one of my browser tab, think like, I have logged in into my Gmail account. With the help of Auth server, I'm going to get an token. Now, this token will be saved somewhere inside my browser. Maybe after a few hours or maybe after few days if I try to open one of the different product of Google like Maps or YouTube or Photos or Drive, it is not going to ask me the credentials again, it is simply going to use the same token that is available inside my browser.

And with this token, the authentication and authorization is going to happen

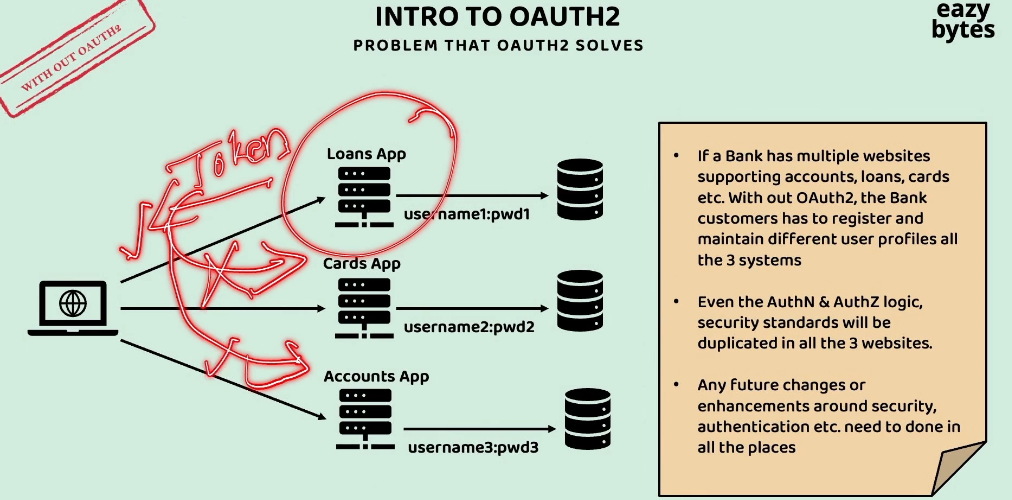
without me entering any credentials.

So this is possible because all these web applications are mobile applications. They're pointing to the same Auth server which issued the token originally.

So this is a very first major problem that OAuth2 is solving inside the industry.

Always OAuth2 framework is going to recommend to maintain separate authentication server inside an organization.

Let's try to take one more scenario and try to understand what are the advantages that we get if you try to maintain a separate Auth server inside an organization.



Think like, there is a bank application which has multiple websites supporting accounts, loans, and cards.

If these bank application, if they're not following the OAuth2 framework,

then the bank customers, they have to register and maintain different user profiles

in all the three systems.

Inside each of these applications, if they have their own authentication and authorization logic, then the end user will end up creating different credentials for different applications, and all these credentials are going to be stored in different storage systems.

So here, clearly, these people, they're not following the OAuth2 framework

inside the bank organization. With these, we are going to have multiple challenges.

The very first challenge or drawback is the security-related logic

like authentication and authorization, it is getting duplicated in multiple places.

So in future, if there is a requirement change around the authentication and authorization, the same has to be done in all these three places.

The next challenge is, think like, this loan application has issued a token

during the login process to the client application.

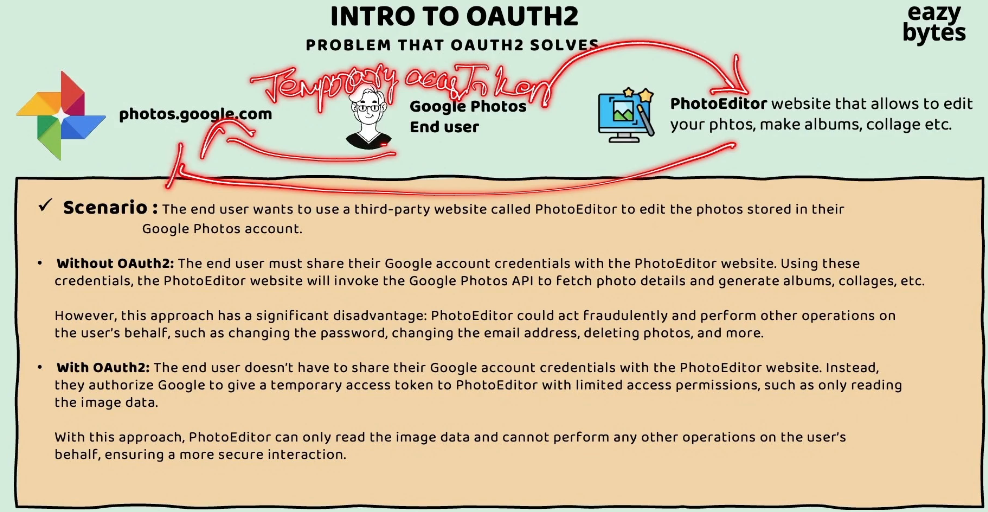
So the same token is not going to make any sense to the other applications.

That's why they're going to reject that token. They always accept their own credentials, and they're going to issue their own tokens during the authentication and authorization process.

Always try to separate your security-related logic into an Auth server

so that your individual applications, they are going to simply have the logic

related to the business logic.



The next common problem that OAuth2 framework try to solve inside the industry is,

Delegated Authentication or Authorization.

Think like you are an end user who maintain all your photos inside the Google Photos, and there is a third-party app with the name PhotoEditor.

So this PhotoEditor third-party app, it is going to allow you to edit your photos

with the help of artificial intelligence, and it is also going to help you

to create albums out of your photos. It is also going to help you to create collage of your photos.

Since this third-party app is providing all these features, you decided to use this third-party app.

So let's try to imagine this scenario

without OAuth2 and with OAuth2.

In a very common basic scenario, what you're going to do?

You're going to download the photo from your Google Photo.

The same you're going to upload into the PhotoEditor for performing some edits.

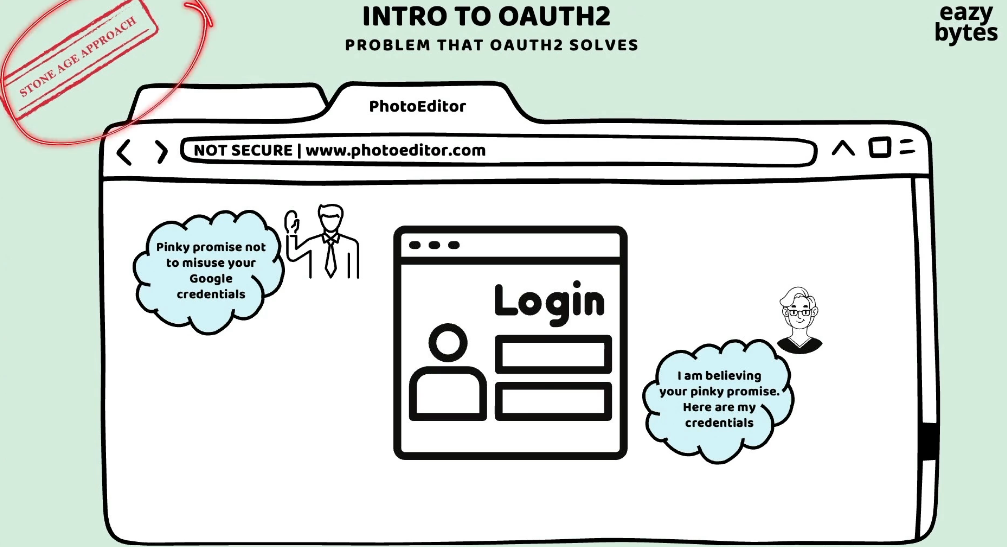
So the same process you need to do for all the photos.

So this is going to be very cumbersome process.

So to make your life easy, PhotoEditor, they supports integration with the Google Photos.

So this integration with the OAuth2, it is going to be supported in two different styles.

One is without OAuth2 and the other one is with OAuth2 framework.



So let's try to understand what is going to happen inside the without OAuth2 scenario.

In this scenario, the PhotoEditor third-party app, it is going to ask the end user

to provide his Google Photo credentials.

Though sharing the Google credentials directly to this website is a risky option,

still this end user want to share his credentials because he liked the features

provided by this website a lot. They're going to use the credentials of this end user,

and they're going to read the photos from the Google Photos website for editing.

So this used to be a stone age approach where people used to follow these

before OAuth2 introduced.

So whatever we discussed, this is a very happy scenario.

But there is a significant disadvantage in this approach, which is, the PhotoEditor application, they may do some fraudulent activities by performing other operations on user's behalf. They can change the password or they can change the email address because they know the actual credentials of the end user.

Now, let's try to imagine the same scenario with the OAuth2.

So with OAuth2 what is going to happen is, the end user will not share his Google credentials to the third-party application. Instead, he's going to let Google Photos

to issue an temporary access token. Using this temporary access token, the PhotoEditor app, they can only read the photos from the end user Google Photos account. With the access token that is issued by the Google Photos

on behalf of the end user, it is only useful to read the images or to load the images.

Apart from reading, any other operations will straight away rejected by the Google photos because the access token issued to the third-party app

does not have enough privileges.

So whenever we are using OAuth2, what is happening?

The end user delegating the authentication and authorization

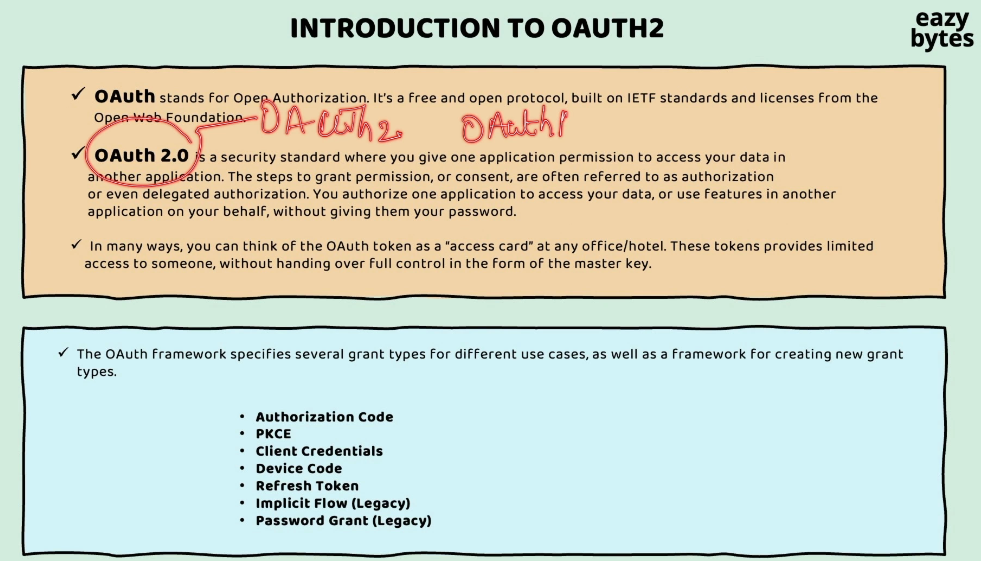
to the PhotoEditor app to read his photos from the Google Photos.

And whenever the end user feel like he no more needed the services of this website,

he can easily revoke the permission that he has initially issued to this website

by logging into the Google account.

Introduction to OAUTH2



OAuth 2 is a free and open source protocol that is built by the open source community. When I say OAuth 2, it is a standard, or it is a protocol,

or it is a specification that every organization has to follow

whenever they're trying to implement authentication

and authorization inside their organization.

Whenever an access token is issued to a client application or a third-party application, with the help of that access token, they should be able to read that data of the end user. So with the help of this access token, they should be able to perform only a limited functionality.

You need to follow one of the grant types that are supported by the OAuth 2 framework.

For example, if an end user is involved during the authentication and authorization process, then we can use either of the Authorization Code or PKCE grant type flow

based upon the type of our application.

If your application is built using the JavaScript frameworks, like, Angular, React, or mobile applications, in such scenarios, you need to use PKCE.

Otherwise, you can safely use Authorization Code.

The next grant type flow that we have here is Client Credentials.

So these Client Credentials, we need to use whenever two different backend applications or two different APIs, they're trying to communicate with each other.

So in these scenarios, the end user is not involved.

whenever two different devices or IoT applications, if they're trying to communicate with each other, we need to use the Device Code grant type flow.

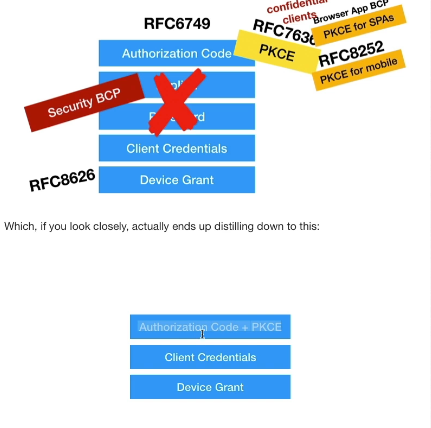
Similarly, there is a Refresh Token grant type flow that can be used in the scenarios whenever an access token is expired and if you're looking to get a new access token.

Implicit flow is very similar to the Authorization Code grant type flow, but it's a older version of Authorization Code grant type flow, which has some drawbacks.

That's why this Implicit flow is deprecated right now.

This Password grant type flow is also a flavor of Authorization Code grant type flow,

and this is also deprecated due to its drawbacks.

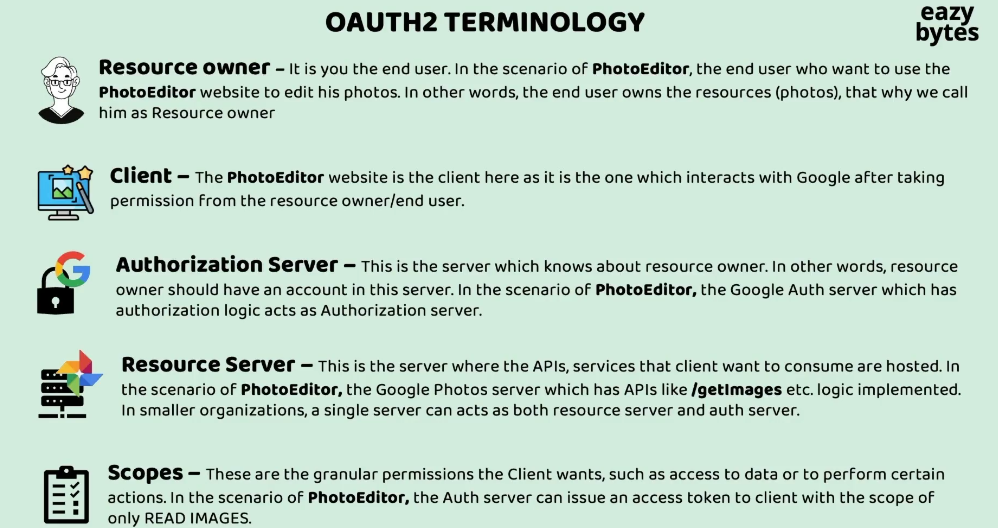


As part of OAuth 2.1, the open source community, they're trying to reduce the number of grant type flows to Authorization Code plus PKCE and Client Credentials and Device grant.

With that, we are only going to have three grant type flows in the future when this 2.1 is officially released.

So they are recommending to use the Authorization Code with PKCE instead of using them separately.

OAuth2 terminologies or jargons



So in the scenario of PhotoEditor, the PhotoEditor application,

it is going to invoke the /getimages API, which is exposed by the Google Photos resource server.

So when client application asks for the resources, obviously, the resource server will expect an access token before providing the response.

So how to get the access token, the client first has to reach out to the authorization server to get the access token. Once the access token is received, using the same access token, it can reach out to the Resource Server to fetch the resources.

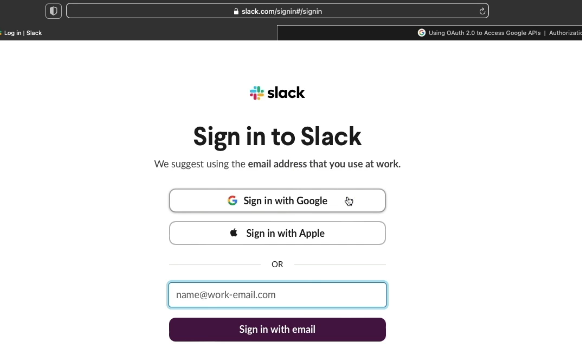
In smaller organizations, there is a good possibility that a single server can act

as both resource server and auth server.

**So just like how we have authorities and roles concept inside the Spring Security framework, very similarly, inside the OAuth 2 standards, we call them with the name or with the jargon scopes.**

Demo of OAuth2 Sample flow

Apart from this regular process of account creation or account login, this slack website is also supporting social login.



So what is a social login?

Social login is a process where we can quickly sign in into a third party application by entering our social credentials from Google or Apple or Facebook, Twitter, GitHub, LinkedIn. So since all these Google, Apple, or any other social organizations, they have their own authorization servers.

So here, what Slack is doing is it is giving a flexibility to the end user if you are in a hurry and not able to create your own account from scratch by entering your first name, last name, email, and date of birth, gender, all other details, you can simply use the social login option.

So whenever we use social login option, what is going to happen?

My profile details which are stored inside the Google auth server, they are going to be fetched and given to the Slack.

So Slack behind the scenes, by taking my profile details from Google or Apple,

it is going to create an account for me very quickly.

Initially, OAuth 2.0 is invented for delegated authorization where we can delegate authorization to your third party app to read our own data in some other application.

But inside the industry, people became super smart, and they started using for authentication as well.

So here, the Slack website, they're trying to use the OAuth 2.0 framework for authentication. In future, whenever I want to log in into this application,

I just have to use the same option, which is sign in with Google.

And with that, my authentication is going to be super quick.

Here, Slack has a integration with the Google Auth server.

Behind the scenes, what Slack is going to do is before it try to enable this button on this website, they will approach to the Google team and express their interest in using social login in the Slack website, and Google team, what they're going to say, they'll say, "Um fine, you just have to create your own details inside my Auth server so that the integration between Auth server and Slack is going to work seamlessly."

So if you go to the official documentation of OAuth 2.0 from Google inside this website, you'll be able to see what are the steps that any organization has to follow if they want to use Google as an auth server inside their websites.

So first, the organization like Slack, they have to register themselves to get the client ID and their client secret.

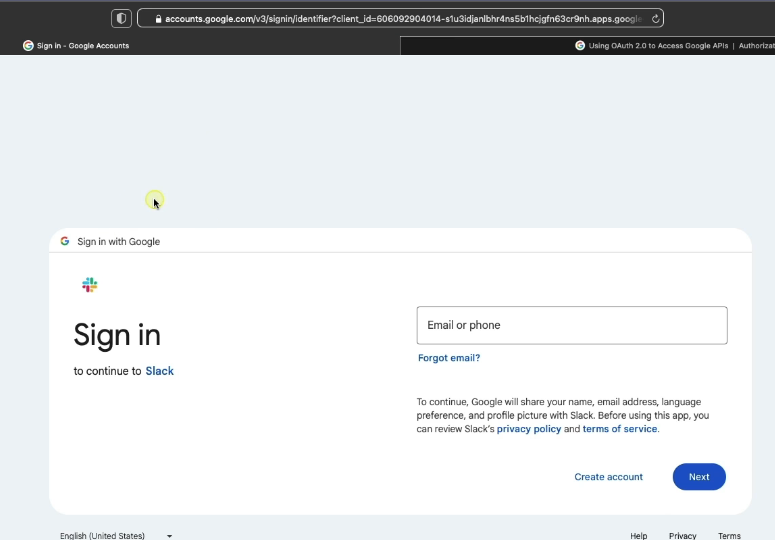
**So these credentials are different from the end user credentials.**

These client ID and client secret, they're going to be used by the Google Auth server

to identify which client application is trying to invoke the authentication or authorization process. Here, the client is going to be Slack, so Sack is going to get its own ID and secret, which it has to use while it is trying to invoke the authentication process here.

So here, if I click on the sign in with the Google, it is going to redirect me to the Google login page. You can see the domain, accounts.google.com,

and here there is a client ID which, so this client ID represents Slack.

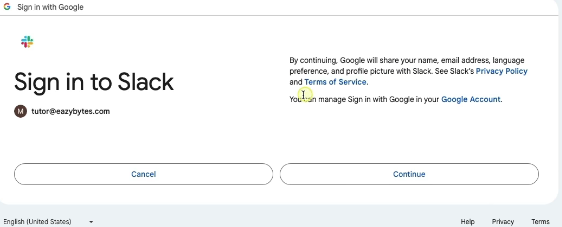


So here, I can enter my Gmail username and password. As soon as I click Next,

you'll be able to see I'm getting a consent page that is being asked by Google, saying that by continuing Google will share my name, email address, language preference, profile picture with the Slack.

So here, as soon as I click continue, behind the scenes an access token will be issued by Auth server. The same will be sent back to the Slack,

and Slack will land me onto their website.

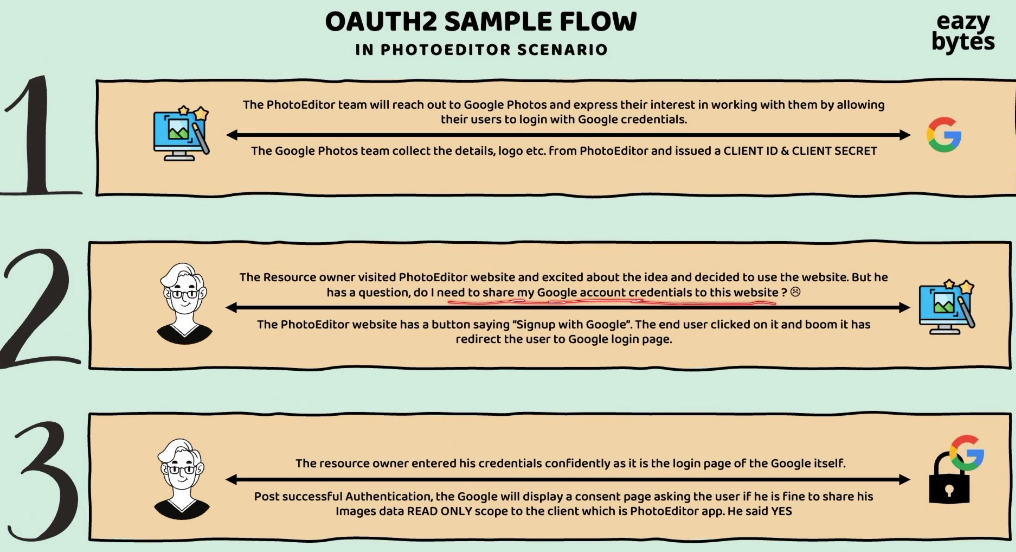


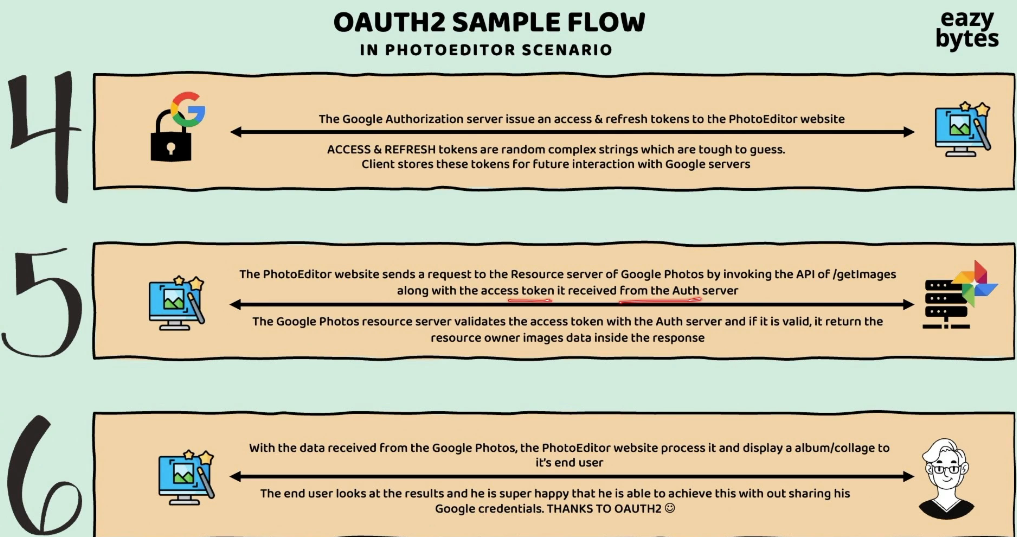
But as a software developer, we will not see these kind of social logins inside real applications.

For example, if you are building a web application or mobile application for a bank

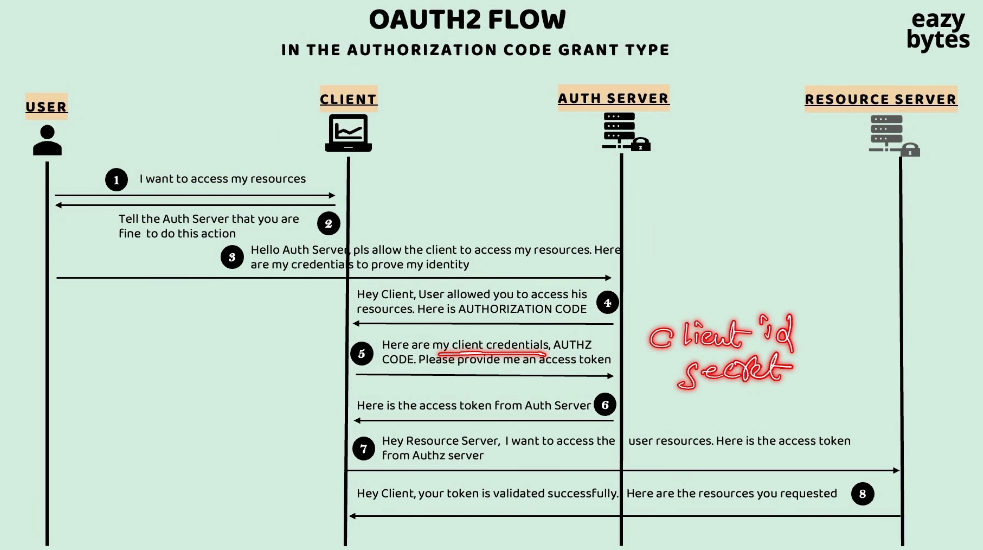
or for an insurance company, there is a very good chance that they may not use the social login option.

Instead, they are going to set up their own auth server using which they'll be able to issue the access tokens to the third party applications or to the applications belonging to the same organization.





 Deep dive on Authorization code grant type flow in OAUTH2



There'll be four different components or parties involved. The very first party is end user. End user, in other words, we can also call as Resource Owner.

The second component is Client application followed by Auth Server, followed by Resource Server. So this Client application and Resource Server can belong to different organizations, or they can belong to the same organization.

Regardless of their relationship, this authorization code grant type flow,

it's going to work very similarly.

Steps;

Whenever an end user requested a resource to a Client application.

The Client application will redirect the end user to the login page of the Auth Server.

The end user is going to prove his identity by entering username and password

inside the login page of the Auth Server.

Once credentials are validated, the Auth Server will return an authorization code. So this is not an access token. This is a temporary code that is going to be issued by the Auth Server to the client application.

So as part of the step three, the end user proved his identity. As a next step 5,

the client also has to prove its identity by sharing its client secret.

Along with the client credentials (client\_id and client\_secret), it should also send the same Authorization Code that it has received as part of step four.

So by providing all these details, it is going to request the auth server to issue an access token.

If all the details are valid, as part of the step six, the Auth Server, it is going to issue an access token to the client application.

So now using this access token, client is going to call the Resource Server

saying that I want to access so-and-so end user resources and this is the access token for your reference.

The Resource Server is going to validate the access token is valid or not. If the access token is valid, it is going to send the actual resources requested

by the client application as a response.

How my Resource Server is going to validate the access token issued by the Auth Server?

Because they are two different components, and I don't see any communication

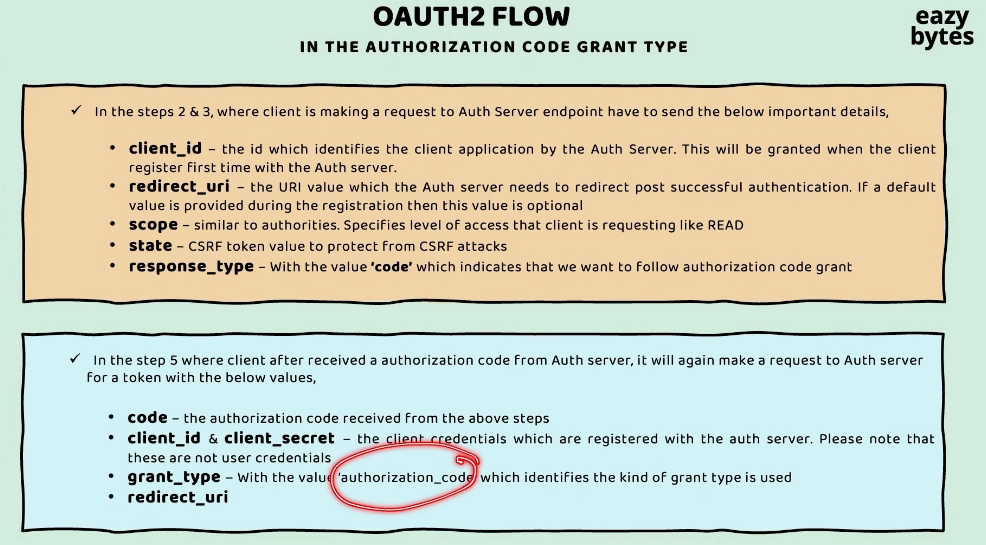
between these two components.

So this can be a question.

If you can recall our JWT tokens discussion. Inside the JWT tokens, there is a digital signature concept. Using the digital signature concept, one can validate if the token is valid or not by themself without the need of reaching out to the token-issuing component. The same is going to happen here as well.

The Resource Server, it don't have to talk with the Auth Server.

Since the access token also is going to come in a JWT format, it is going to validate it very quickly behind the scenes without interacting with the Auth Server.



how my Auth Server will know to which page of the client application it has to land

during the Authorization Code flow and during the access token flow?

As part of the step two and three where the end user identity is going to be verified by the Auth Server.

Client application send the client\_id only, but not the client\_secret. So using this client\_id, the Auth Server can identify the details of the client application,

and the same details, it is going to show you on the login page of the auth server saying that so-and-so client is trying to access your resources.

Are you fine with that?

So this kind of consent the Auth Server is going to show based upon this client\_id

that it received as part of step two and three.

Apart from client\_id, the client application, it is also going to send other details like redirect\_uri, scope, state, and response\_type.

The purpose of redirect\_uri is very simple. Inside these request parameter only,

the client application is going to mention what is the URI value that the Auth Server needs to redirect post to end user's successful authentication.

And inside the scope, the client application is also going to mention what are the authorities or what are the level of access that the client is looking inside the Resource Server.

For example, if the client is looking for the READ **scope**, the same it is going to be mentioned inside the scope details.

So these scope details will also be displayed to the end user inside the login page

saying that so-and-so client is trying to request so-and-so scope details, are you fine?

After the scope we have state request parameter. So inside the state request parameter, we are going to send a randomly generated **CSRF** token value.

So this is to protect from the CSRF attacks.

If the end user identity is verified, the client application is expecting an Authorization Code. That's why we need to mention these **code** as a value inside the response\_type during the step two and three.

So in step four, we know what is going to happen.

The Auth Server, it is going to redirect the response to the redirect\_uri with the Authorization Code. Once the Authorization Code is received by the client application, what it is going to do is, as part of the step five, it is going to prove its own identity.

So that's why this time inside the request, it is going to send both the client\_id and client\_secret. Along with these client\_id and client\_secret, it is going to send the same Authorization Code value that it has received as part of step four.

And this time, it should also mention grant\_type inside the request.

The grant**\_**type should be Authorization Code, because we are expecting

an access token belongs to an end user.

So that's why we need to mention this valid value, which is Authorization Code.

Apart from grant\_type, it is also going to send the redirect\_uri, which is going to have the URI details that Auth Server can be used to redirect the response after issuing an access token.

So whatever CSRF token value that we have mentioned initially under this state,

it is going to play a very important role to avoid the CSRF attacks.

So based upon our discussion, you may have a question, which is, why in the Authorization Code grant type flow, client is making requests two times?

The very first time it is trying to make a request to get the Authorization Code,

followed by another request to get an access token.

So this can be your question.

To answer your question, this grant type flow is designed this way for better security.

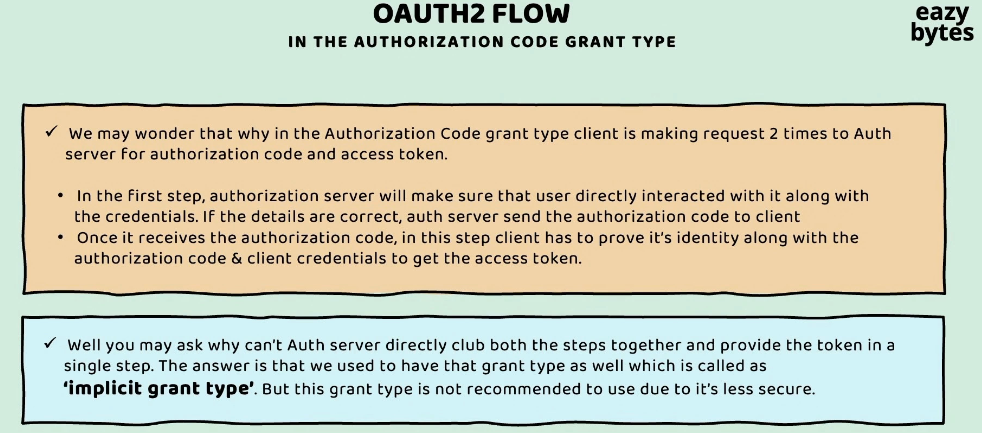
So in the very first step, the authorization server will make sure only the end user directly interacted with it by asking the end user credentials inside the login page.

Once the end user identity is verified, it can't issue directly the access token,

because the Auth Server has to verify the client identity as well.

That's why it is going to make one more request where it is going to expect client\_secret as well. So if the client\_id, client\_secret, and Authorization Code, all of them are correct, then the Auth Server is going to issue an access token.

To answer your question in simple words, we can say inside the authorization code grant type flow, the client application has to call two times for better security reasons.



We have a grant type flow with the name implicit grant type.

As part of this implicit grant type, the client application, it is going to make only one request to the auth server, and within one request, it is going to get the access token directly.

There won't be any authorization code involved.

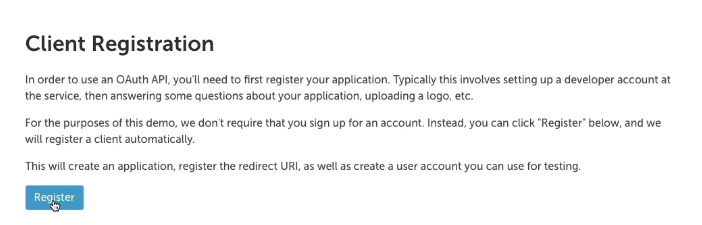
Inside a single step, the auth server is going to validate both end user identity and the client identity.

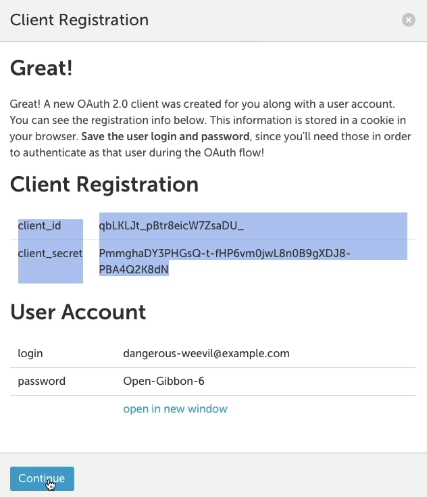
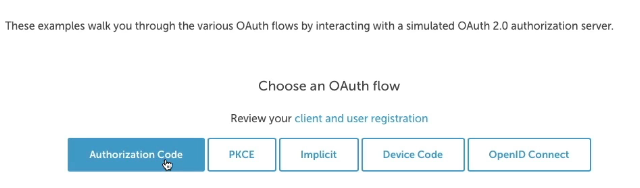
this grant type is deprecated right now due to its drawbacks since it is less secure.

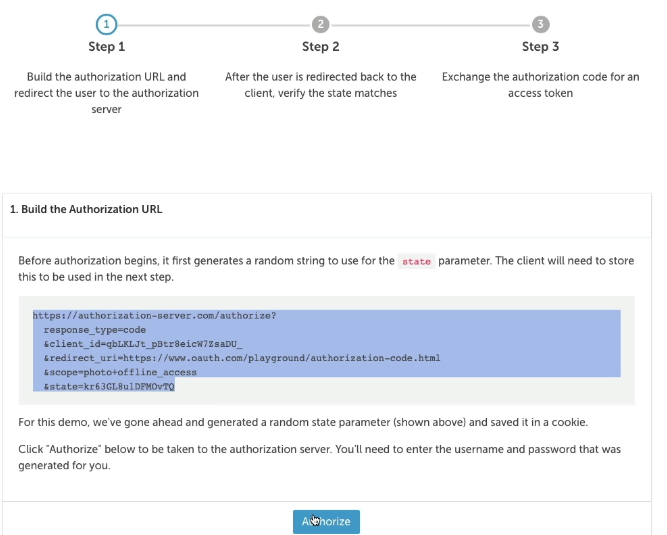
Demo of Authorization code grant type flow in OAUTH2

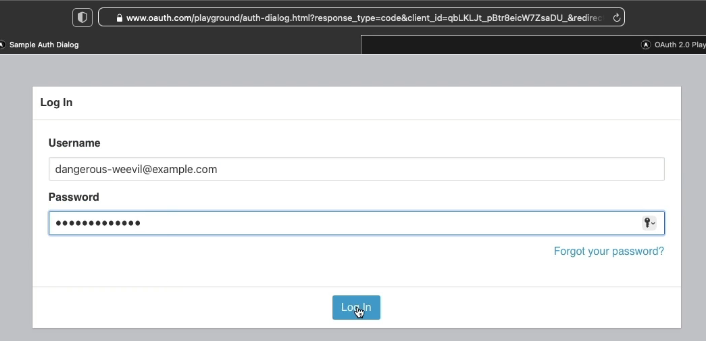
OAuth is an organization acquired by okta recently. So both these OAuth and okta, they have commercial products using which organizations can build their own auth server by following the OAuth standards.

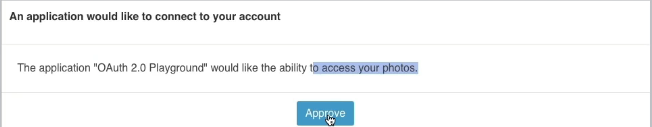










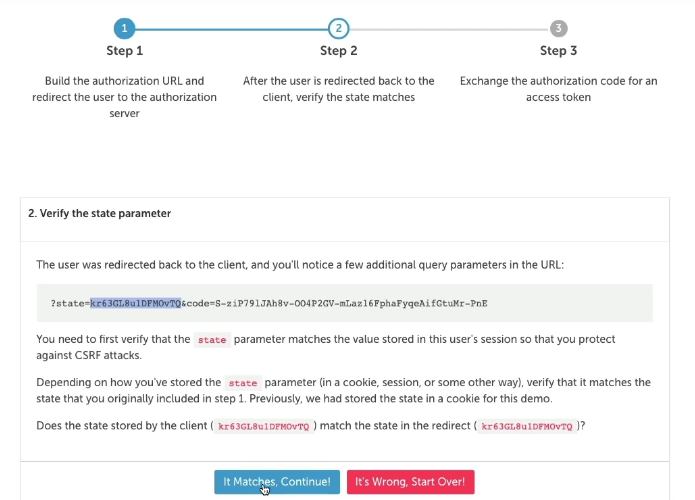
As soon as I click Approve, what is going to happen?

The auth server, it is going to issue the Authorization code.

So as part of the step two, the client application, it is going to make sure that the token value that it has received under the state is it the same as the initial value

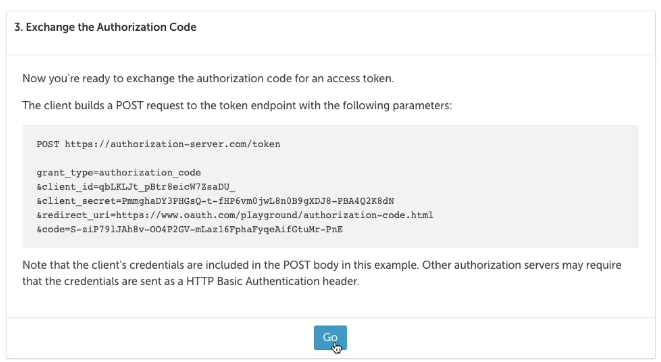
that it has sent as part of the step one.

So this is going to give and confirmation to the client that the authorization server only send these code details but not any other hacker.



So if the state token value is same,

we need to click on this It Matches, Continue.



And inside the third step, the client application is going to send a request

to get an access token. Inside this third step, it is going to make a post request.

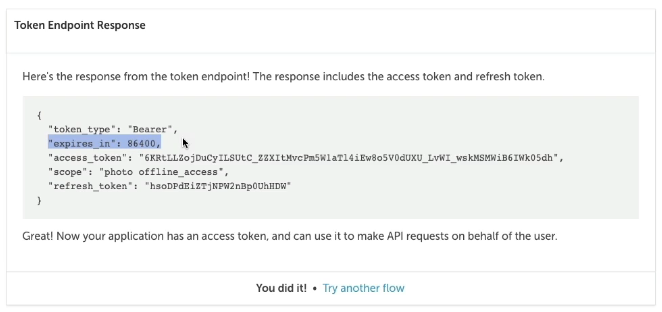
So previously, all the requests are get request.

As soon as I click on this Go button, behind the scenes the auth server, it is going to validate this client details, Authorization code.

If all of them are correct, it is going to issue the access token and redirect the end user to so and so redirect\_uri.

As a response you'll be able to see I'm getting an access token.

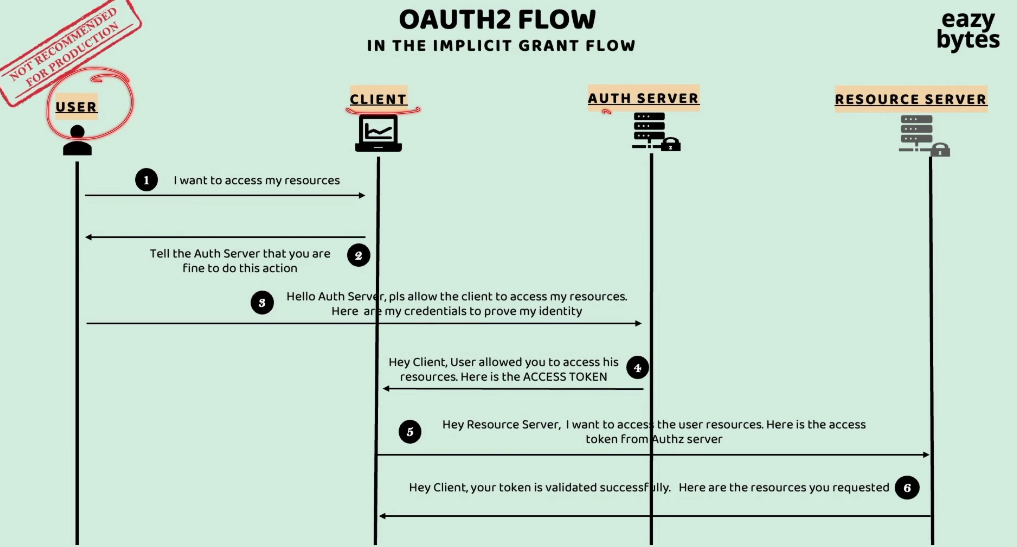
And that refresh token and this access token, it is going to expire in so and so seconds.



Deep dive & Demo of implicit grant flow in OAUTH2

this grant type flow, as of now, it is deprecated,

and soon, inside the 2.1 version, it is going to be completely removed.



If the end user credentials are valid, as part of the step four response, the auth server, it is directly going to issue the access token.

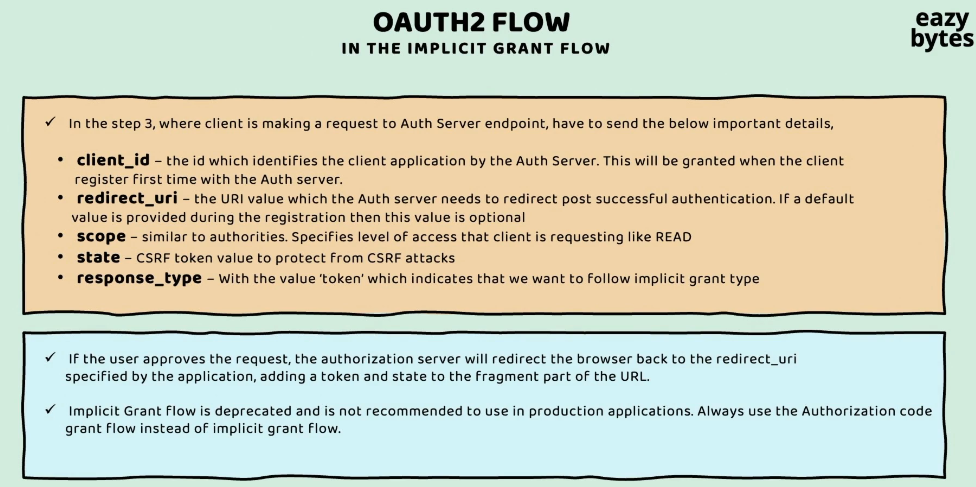
There is no concept of Authorization Code here.

Directly, the access token is going to be issued by the Auth Server to the client application. So once the access token is issued, the client application, it is going to make a request to the Resource Server to access the User resource

by sharing the access token inside the request.

If you try to understand the primary reason around why this is deprecated

is it is trying to share the access token in a single step, and with this, there are certain drawbacks.



Let me try to explain what are those drawbacks.

Since inside the step three, the client is going to make a request to the auth server using the GET request. it is only going to send the client\_id, redirect\_url,

scope, state, and response\_type. So the response\_type is going to be token,

and you know what is the purpose of scope, state, redirect\_url, and client\_id.

Inside this flow, there won't be any Client\_Secret involved,

so only the client\_id is enough. The reason why client secret is not involved is,

since this is a GET request, inside the GET request, there is no meaning of sending the client secret. That's why this flow does not support client secret

as part of the step three request.

And with that what is going to happen?

Anyone who knows the client\_id, they will be able to mimic as a client application with the auth server, and once the end user entered his credentials, they should be able to get the access token.

So since there is no way for the auth server inside this flow to validate the identity of the client application, this is marked as deprecated because hackers, they found too many ways to steal the access token inside this flow.

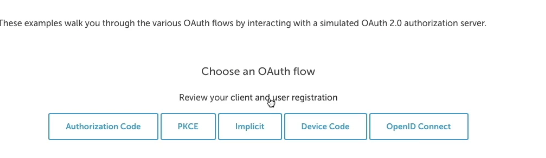
Apart from these drawback, the other serious drawback is when the auth server is trying to share the access token with the client application, the access token is going to be shared inside the request URL itself because, initially, the request went using GET, so to this GET, as a response, the auth server can only send using GET only since it is going to send the access token inside the GET URL itself, there's a very good chance that anyone who has access to your browser history, they'll be able to steal your access token.

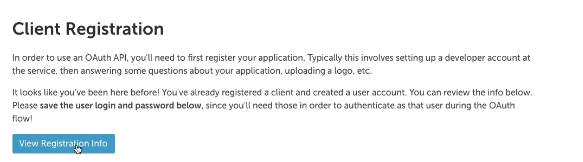
Think of a scenario, one of the auth server is issuing an access token with an expiration of seven days.

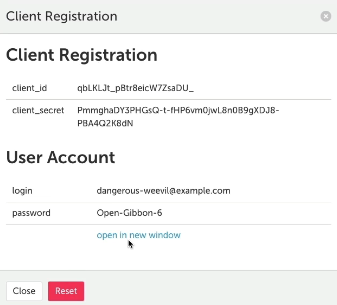
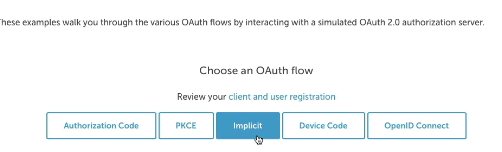
In this scenario, you might have used the access token on day one and you left the computer. If some other user has access to your browser history, they'll be able to easily know what is the access token that auth server sent initially.

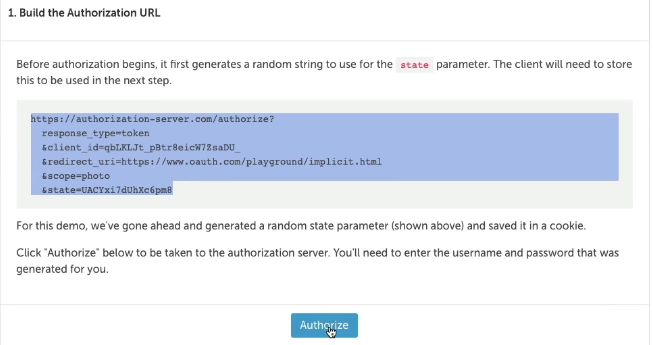
Using the same access token, there is a good chance that they may misuse it.

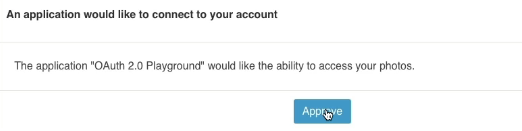
Demo

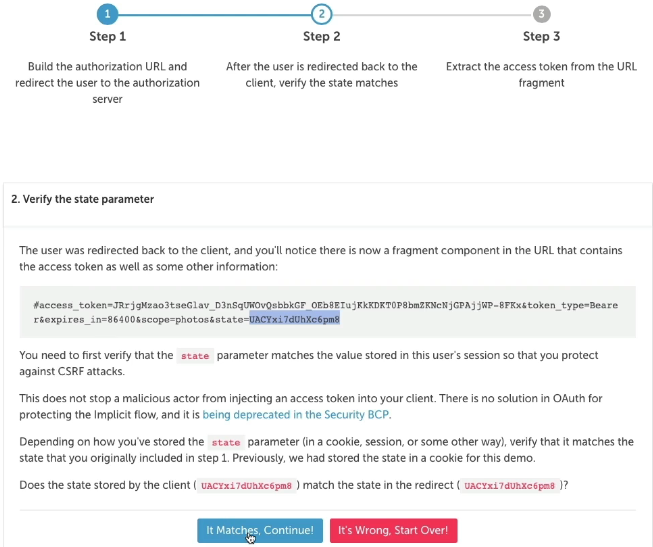


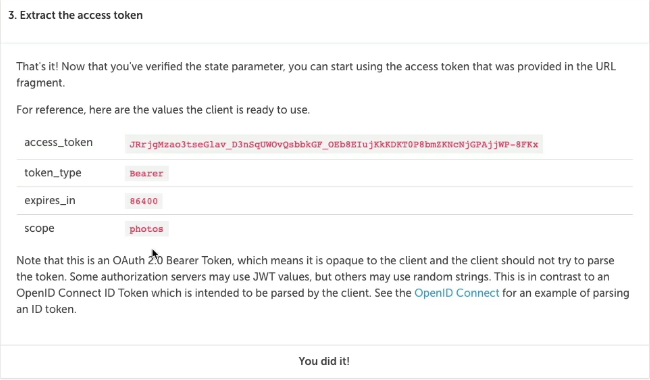








Deep dive & demo of Authorization code grant type with PKCE

So what is a full form of PKCE?

Proof Key for Code Exchange.

This is going to work very similar to authorization code grant type flow,

but it has some minor differences. In other words, we can call this PKCE

as another flavor of authorization code grant type flow.

I said as part of the step five, the client application, they have to share their client credentials. So as part of these client credentials, they need to pass their client\_id and the client\_secret.

But unfortunately, the client secret can't be saved by the public clients like JavaScript-based applications, single page applications which are built

based upon angular reactive framework mobile applications.

Because all the client code, it is going to be built with the help of plain JavaScript

and anyone can see the source code inside the browser or by downloading the mobile application installation file.

And with that, they'll be able to see the client secret very clearly.

That's why for public-facing clients, where they're going to build a code

with the help of JavaScript for such applications, instead of following the authorization code grant type flow, we can happily follow PKCE grant type flow.

Let me explain you how PKCE is going to solve this problem for the public-facing clients.

