**Part 2A: OAuth 2.0 Authorization Code Grant**

<https://medium.com/@shoaib.alam/part-3-oauth-2-0-client-credentials-grant-with-azure-ad-36ac0f9ab2b7>

The OAuth framework specifies several grant types for different use cases. The Authorization Code Grant is probably the most common grant type. It is used by both web and mobile apps to get an access token after a user authorises an app. OAuth doesn’t dictate the authentication mechanism, and the authorization server is free to choose any method such as a username/password, cryptographic certificates, security tokens, federated single-sign-on, or any other possibility. Thus, a wide variety of authentication techniques are available to the user for authentication process.

The main feature of this OAuth flow is its relatively high level of security. The user’s authentication passes directly between the user and the authorization server; it’s never seen by the client application. This protects the user from sharing their credentials with the client application (Password antipattern discussed in [Part 1](https://medium.com/@shoaib.alam/what-is-oauth-2-0-and-how-it-works-90a3ee752de1)). The authorization code grant is also called 3-legged OAuth because it enables the checking of the identity of the three actors involved in the flow: OAuth server, resource owner and the client.

Another interesting feature of this flow is the use of refresh token which allows granting access for a longer period without requiring the resource owner to re-authenticate. This is in fact transparent to the resource owner and only involves the client and the OAuth provider.

**How Authorization Code Flow works**

Authorization Code Flow is used in case the client doesn’t have any tokens from previous interactions. The resource owner will have to log in by providing the credentials. Alternatively, if the client has previously gained access and the refresh token is still valid, the client can use the refresh token to obtain a new access token.

As explained in [Part 1](https://medium.com/@shoaib.alam/what-is-oauth-2-0-and-how-it-works-90a3ee752de1), there are the following four actors in OAuth flow:

**Resource Owner or User**

The Resource Owner wants to use web or mobile app.

**Client**

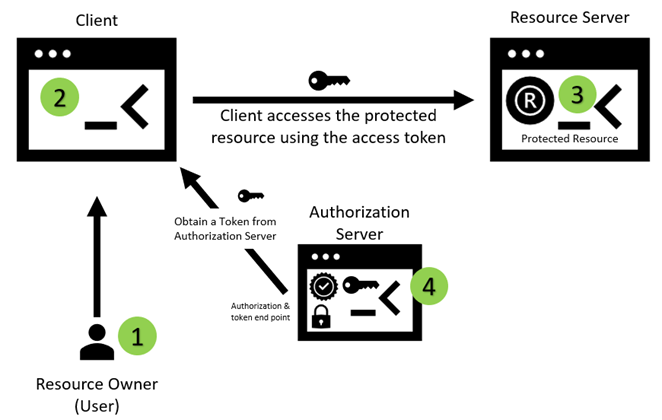
The web or mobile app takes the role of a client and tries to access the protected resources or APIs.

**Protected Resource**

The protected resources or APIs protected by resource provider or Server.

**Authorization Server**

The Authorization Server creates the tokens and codes before it sends them back to the client.

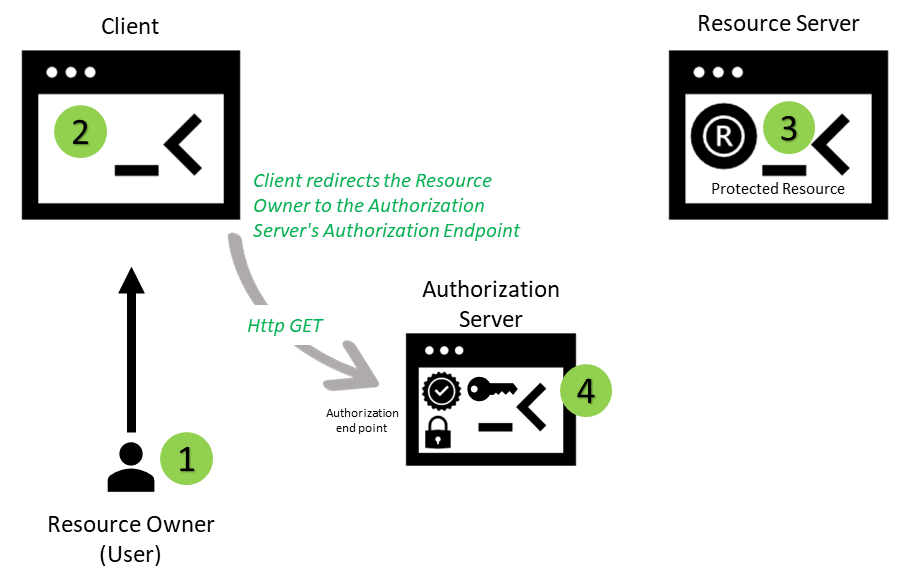


All these four actors participate in Authorization Code Grant flow. Now we start looking at different steps between OAuth actors to complete Authorization Code Grant flow. Let’s break this down it into individual steps:

**Step 1: Redirecting the Resource Owner to the Authorization Server to start the process.**

The resource owner (user) goes to the client application and indicates to the client to use a particular protected resource on his/her behalf. For instance, the user would tell the client to use a specific service to get their profile details. This get profile service is an API that the client knows how to process it. And the client also knows that it is protected by OAuth. When the client realizes that it needs to get a new OAuth access token, it redirects the resource owner to the authorization server making a request which indicates that the client is requesting an authority from resource owner to read his/her profile.

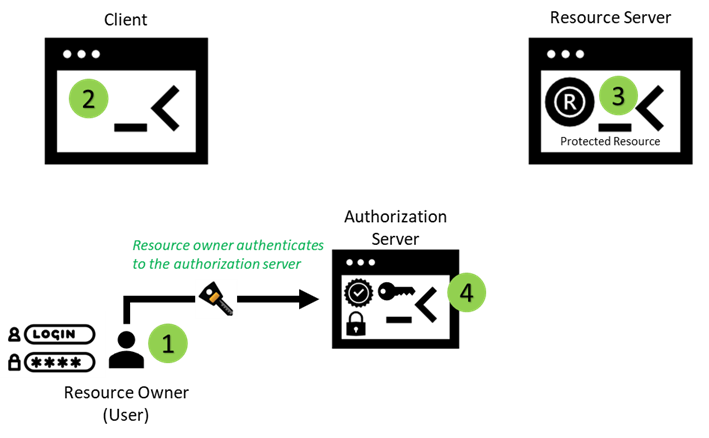
This redirect causes the browser to send an HTTP GET to the authorization server.



**Step 2a: The resource owner authenticates to authorization server.**

In the next step, the authorization server requires the user to authenticate. This step is essential in determining who is the resource owner and what rights they’re delegating to the client.

The user authentication process is directly between the user and the authorization server, and it’s never seen by the client application. This protects the user from having to share their credentials with the client application i.e., the antipattern that OAuth was invented to combat.

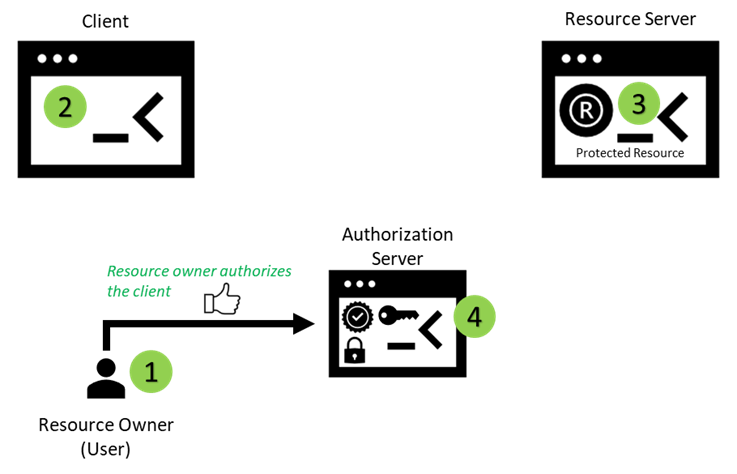


*Note: A wide variety of authentication techniques are available for the user to authenticate such as a username/password, cryptographic certificates, security tokens etc.*

**Step 2b: The resource owner authorizes the client.**

In this step, the resource owner chooses to delegate some portion of their authority to the client application. The client request can include an indication of what kind of access it’s looking for in the form of OAuth scopes. The authorization server can allow the user to deny some or all scopes or approve or deny the request as a whole.

Many authorization servers persist this authorization decision and won’t prompt the user for scopes in the subsequent requests for the same client. The user will still be redirected to the authorization endpoint, and will still need to be logged in, but the decision to delegate authority to the client have already been made previously.



**Step 3: The authorization server redirects the user with authorization code.**

The authorization server redirects the user back to the client. This redirect includes the special code as a query parameter. The value of this parameter is a one-time-use credential known as the authorization code, and it represents the result of the user’s authorization decision.

This redirect takes the form of an HTTP redirect to the client’s redirect\_uri. Once the app has been successfully authorized and granted an authorization code the authorization server sends the code to the redirect URI.

Get Shoaib Alam’s stories in your inbox

Join Medium for free to get updates from this writer.

Subscribe

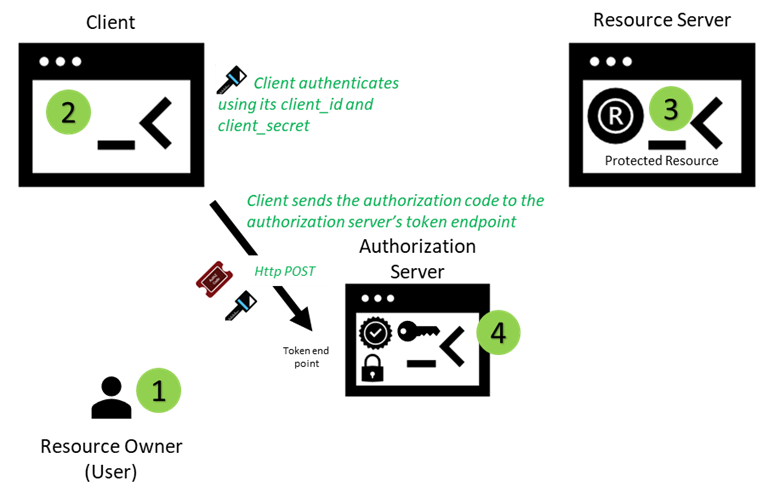
Redirect URI is registered as a part of an app registration process. It is important to register the correct location for redirect URI. OAuth redirect\_uri parameter value must match this redirect URI registered in the app. Which will be explain later when I register an app with Azure AD.



**Step 4: Client sends the request to the authorization server’s token endpoint.**

Now the client has the authorization code, it can send it back to the authorization server on its token endpoint to make a request for access token.

The client performs an HTTP POST with its parameters as a form-encoded HTTP body, passing its client\_id and client\_secret as a HTTP Basic authentication scheme in authorization header. This HTTP request is made directly between the client and the authorization server, without involving the browser or resource owner at all. This ensures that the client can authenticate itself directly without other components being able to see or manipulate the token request.

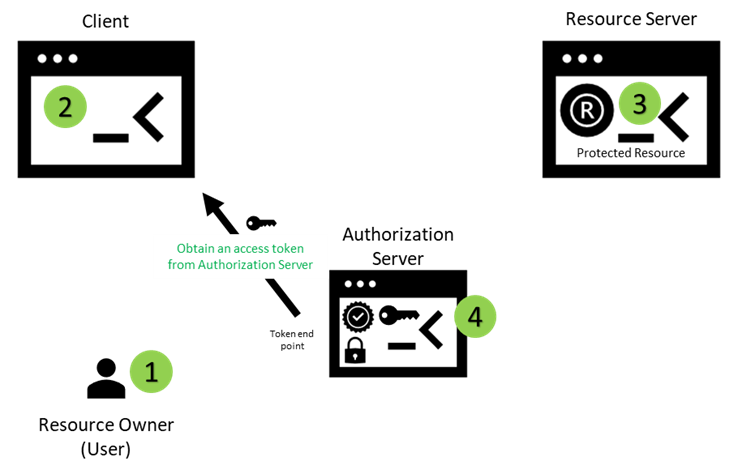


**Step 5: Authorization Server issues an access token.**

The authorization server performs several steps to ensure the request is legitimate.

First, it validates the client’s credentials i.e., client\_id and client\_secret as mentioned in step 4. Then, it reads the value of the code parameter which is an authorization code from the body and looks up any information it has about that authorization code, including which client made the initial authorization request, which user authorized it, and what it was authorized for.

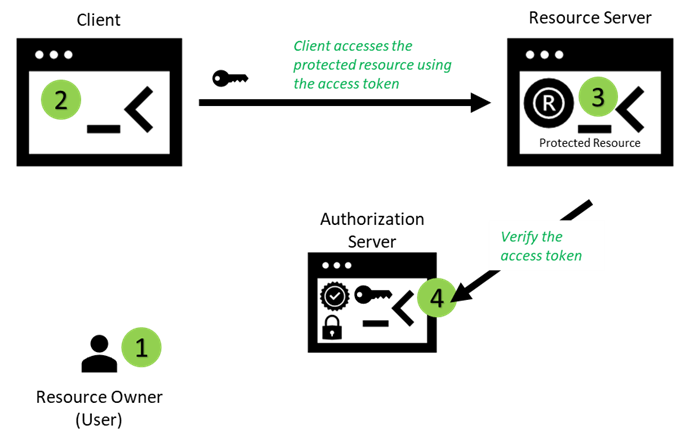
If the authorization code which is generated in step 3 is valid, has not been used previously, is not expired, and the client making this request is the same as the client that made the original request, the authorization server generates and returns a new access token for the client.



**Step 6: Client accesses the protected resource using the access token.**

Once client receive the token it can present it to the resource server. One of the options is to present this token as a Bearer token in Authorization header.

The resource server can then parse the token out of the header, determine whether it is expired or not, who authorized it and what it was authorized for. A resource server has several options for doing this validation of the token. In case of JWT the resource server can verify the token signature along with various claims inside the token.

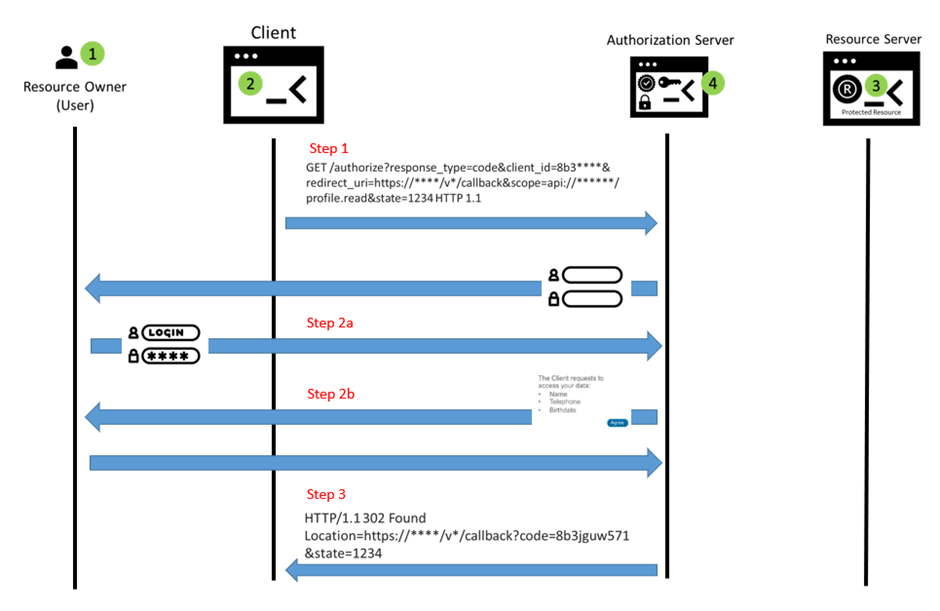


**Sequence Diagrams**

Following sequence diagrams covers a use case for a specific endpoint.

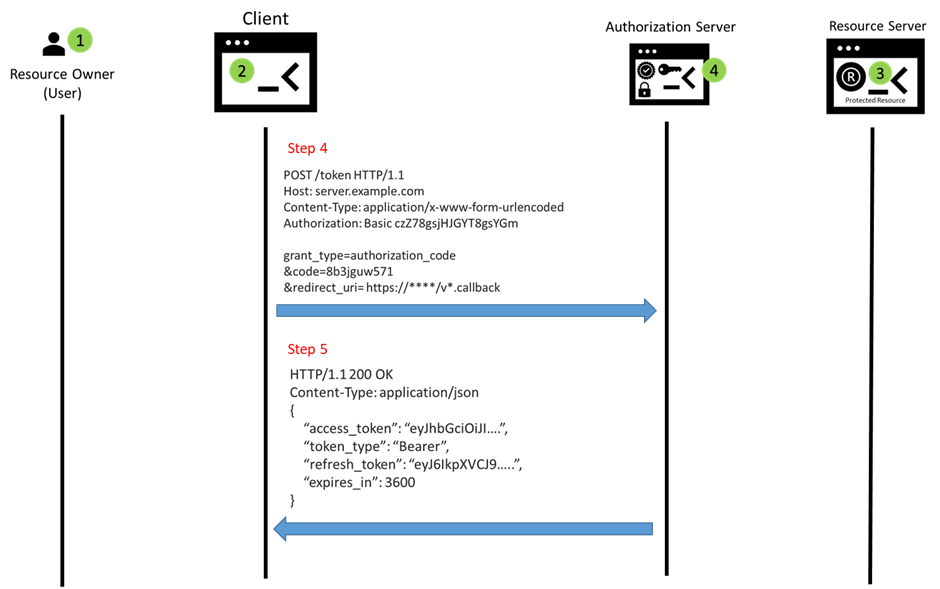
**Authorization End Point**

The result delivered by authorization endpoint is an authorization code.



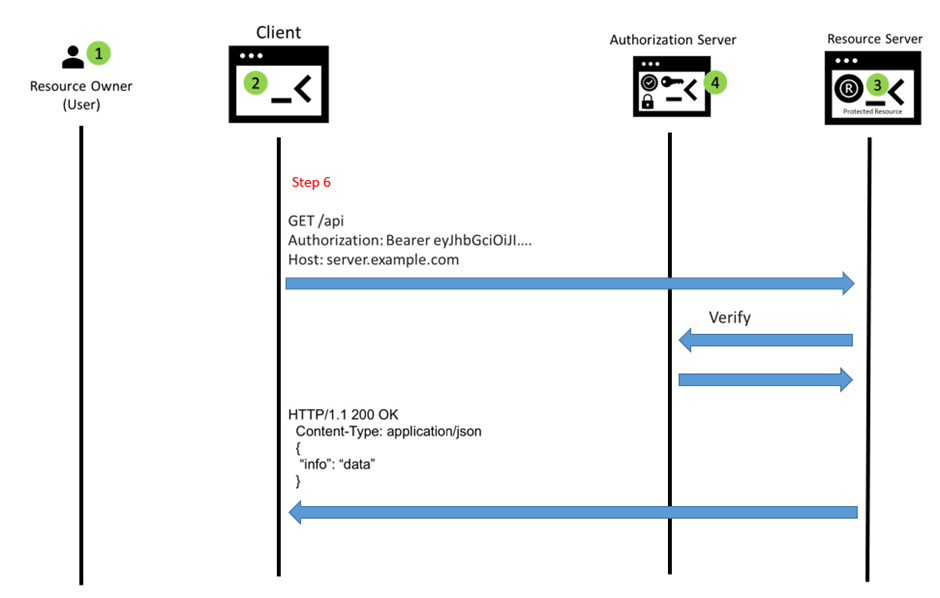
**Token End Point**

The result delivered by token endpoint is a JSON object containing access token and refresh token (optional).



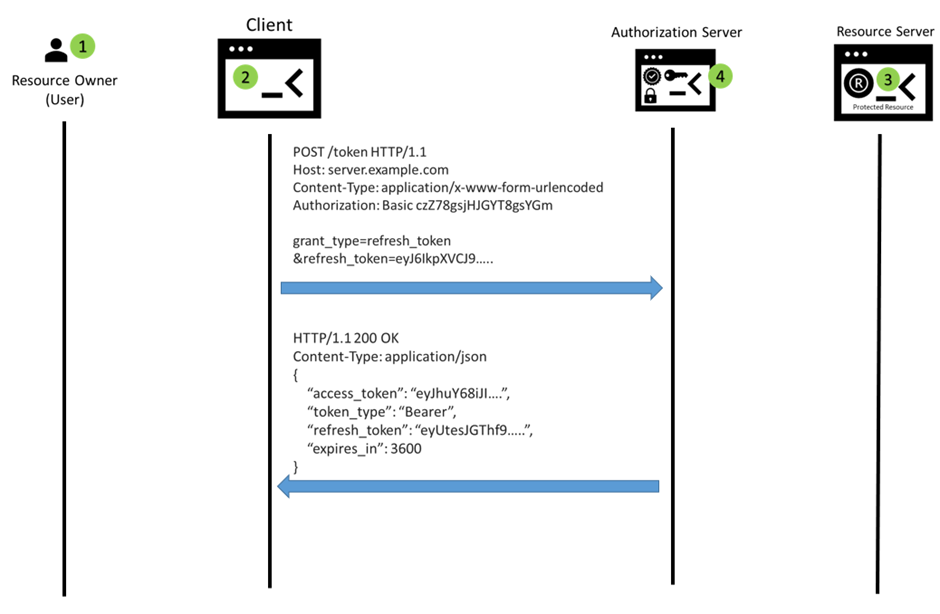
**Resource Access**

The resource server verifies the access token before granting access. If the access token is valid, the requested resource is returned. If the access token is invalid, in general the status code 401 (Unauthorized) is returned. However, if the access token is expired, the status code 403 (Forbidden) is returned.



**Refresh token**

Following sequence diagrams covers a use case for refresh token. A HTTP POST request to the token endpoint will generate a new access and refresh token.



In the next post I am going to use Azure Active Directory or Azure AD to explain the authorization code grant flow. Azure AD will act as an Authorization server.