Complete List of OAuth 2 Grants

By Ravgeet Dhillon

FusionAuth

Auth. Built for Devs, by Devs.

Introduction

Authorization is necessary to protect resources from malicious use. When the Internet Engineering Task Force (IETF) drafted internet protocols and rules, it also planned out different methods to protect and access resources on a server. These efforts led to OAuth 1.0 and later OAuth 2.0.

The OAuth 2.0 specification is an authorization framework containing a number of methods, or grants, by which a client application can get an access token. The access token can be presented to an API endpoint, which can then examine it to determine validity and permissions levels. Each grant type is designed for a particular use case.

OAuth 2.0 focuses on authorization. There are other protocols like OpenID Connect (OIDC) that focus on authentication. OIDC allows software to access login and profile information about the logged-in user.

This article will go through all the different OAuth 2 grant types and explain the flow for each so that you can determine which is the best fit and safely use it in your applications.

If you aren't familiar with terms like Authorization Server and Resource Server, you might be interested in What is OAuth.

Authorization Code Grant

The authorization code grant is used to sign into applications by using third-party authentication providers like Google, Facebook, and GitHub or your own OAuth server like FusionAuth. It is one of the most common methods used on the web to authorize and authenticate the Client to access protected data from the Resource Server.

Flow for Authorization
Code Grant

1. The Client redirects the user to the authorization endpoint on the Authorization Server with the following parameters in the query string:

- **response_type, with the value code**
- **client_id, with the Client's id**
- **redirect_uri, which is where the Client should be redirected after successful authentication at the Authorization Server. This parameter is optional, but without it, the user is redirected to a pre-registered redirect URI**
- **Scope, a space-delimited list of scopes**
- **State, with a CSRF token. This is optional but reccomended. Store the value of the CSRF token in the user's session so that it can be validated when they return**

2. These parameters are validated by the Authorization Server.

3. The user is asked to log in to the Authorization Server and approve the Client. If there are scopes provided, those must be approved.

4. If the user approves the Client, they are redirected from the Authorization Server back to the Client (specifically to the redirect URI) with the following parameters in the query string:

- **code, with the authoriztion code**
- **state, with the state parameter sent in the original request. The Client should compare this value with the one stored in the user's session to make sure the authorization code obtained is in response to requests made by this Client and not another client application**

5. The Client sends a POST request to the token endpoint on the Authorization Server. The Client also must include client credentials if it's a confidential client or it was issued client credentials. The Authorization Server will authenticate the Client if client credentials are included and verify that the refresh token was issued to the authenticated client. The client authentication process is described in detail in section 3.2.1 of RFC 6749. The POST request must include the following parameters:

- **grant_type, with the value of authorization_code.client_id, with the Client's id.**
- **redirect_uri, with the same redirect URI the user was redirected back to.**
- **code, with the authorization code from the query string.**

6. The Authorization Server responds with a JSON object containing these properties:

- **token_type, with the value Bearer**
- **expires_in, with an integer representing the TTL of the access token**
- **access_token, the access token itself**

scope and refresh_token may be returned as well.

There is also an extension called **PKCE** which prevents certain attacks if the client secret cannot be secured, such as in a single page application.

Implicit Grant

The implicit grant type is used by user-agent-specific clients like web browsers or email readers. Generally, it's used by single-page web applications that can't store client secret credentials because their application code and storage are publicly accessible.

This grant is no longer recommended for getting access tokens because it is less secure than the authorization flow, since it lacks client authentication. The implicit grant is only used in legacy applications. Therefore, no flow will be outlined.

Resource Owner Password Credentials Grant

This grant is based on the functionality of the username and password credentials of a resource owner (user) to authorize and access protected data from a Resource Server. This kind of grant works well for trusted first-party clients on both web and platform applications. It is also useful as a stepping stone when porting applications using custom auth solutions over to OAuth-based solutions.

Flow for Resource Owner Password Credentials Grant

1. The Client asks the user for authorization credentials (generally a username and password).

2. The Client sends a POST request with the following body parameters to the authorization endpoint on the Authorization Server:

- **grant_type, with the value password.**
- **scope, with a space-delimited list of requested scope permissions.**
- **username, with the user's username.**
- **password, with the user's password.**
- **Client credentials as described in the Authorization Code Grant section.**

3. The Authorization Server responds with a JSON object containing these prop-

- **token_type, with any string value based on the application's use case.**
- **expires_in, with an integer representing the TTL of the access token.**
- **access_token, which is the access token itself.**
- **refresh_token, which is a refresh token used to get a new access token when the original expires (optional).**

Client Credentials Grant

The Client Credentials grant type uses the Id and secret credentials of a Client to authorize and access protected data from a resource owner. This grant type is generally used for machine-to-machine authorization, in which a specific user's permission to access data isn't required. It can be used in a cron job that performs daily housekeeping tasks on the server.

Flow for Client Credentials Grant

1. The Client sends a POST request with the following body parameters to the authorization endpoint on the Authorization Server:

- **grant_type, with the value client_credentials.**
- **scope, with a space-delimited list of requested scope permissions**
- **Client credentials as described in the above Authorization Code Grant section.**

2. The Authorization Server will respond with a JSON object containing the following properties:

- **token_type, with any string value based on the application's use case.**
- **expires_in, with an integer representing the TTL of the access token.**
- **access_token, the access token itself.**

Refresh Token Grant

The Refresh Token grant type is used to gain a new access token from the Authorization Server by providing the refresh token to the token endpoint on the server.

Every access token expires after a specific period in time, and a refresh token is sent to the Authorization Server to get a new access token.

Flow for Refresh Token

1. The Client sends a POST request with the following body parameters to the authorization endpoint on the Authorization Server:

- **.grant_type, with the value refresh_token**
- **refresh_token, with the refresh token.**
- **scope, with a space-delimited list of requested scope permissions. This is optional; if it isn't sent, the original scopes will be used. You can request a reduced set of scopes instead.**
- **Client credentials as described in the Authorization Code Grant section.**

2. The Authorization Server responds with a JSON object containing the following properties:

- **token_type, with the value Bearer.**
- **expires_in, with an integer representing the TTL of the access token.**
- **access_token, which is the access token itself.**
- **refresh_token, which is a refresh token used to get a new access token when the original expires (optional).**

JWT Bearer Grant

The JSON Web Token Bearer or JWT grant type is used for issuing access tokens to the Client without requiring the Client to send any confidential secrets. This grant type is mostly used with trusted clients to get access to user resources without authorization. It allows secure calls to be made without sending the client credentials

Flow for JWT Bearer Grant

1. JWT requests require the signing of the JWT assertion using public-key cryptography.

2. The Client sends a POST request with the following body parameters to the Authorization Server:

- **grant_type, with the value urn:ietf:params:oauth:grant-type:jwt-bearer.**
- **assertion, which must contain a single JWT**
- **scope (optional), which contains a space separated list of scopes**
- **Client credentials as described in the Authorization Code Grant section.**

3. The Authorization Server responds with a JSON object containing the following properties:

- **token_type, with the value Bearer**
- **expires_in, with an integer representing the TTL of the access token**
- **access_token, which is the access token itself**

Device Code Grant

The Device Code grant type works well for devices that don't support an easy data entry method. This grant type is used by browserless or input-constrained devices in the device flow to exchange a previously obtained device code for an access token. It offers simple integration for developers.

Flow for Device Code Grant

1. An OAuth client makes a POST request to the authorization endpoint on the Authorization Server:

- **response_type, with the value device_code.**
- **client_id, which is the client's Id.**

2. The Authorization Server responds with a JSON payload:

- **verification_uri, which is the URL that the user navigates to on another device.**
- **user_code, which is the code the user enters once they've authorized with the Authorization Server.**
- **device_code, which is the unique Id assigned to the Client.**
- **interval, which is the polling time in seconds at which the client should poll the server for an access token.**

3. The Client keeps attempting to acquire an access token every few seconds (at a rate specified by interval) by POSTing to the access token endpoint on the Authorization Server:

- **grant_type, with the value urn:ietf:params:oauth:grant-type:device_code.**
- **client_id, which is the code the user enters once they've authenticated with the Authorization Server.**
- **code, which is the value of the device_code from the JSON response in the previous request.**

4. The Client should continue to request an access token until the end user grants or denies the request or the verification code expires.

5. The Authorization Server will respond with a JSON object containing these properties:

- **token_type, with the value Bearer.**
- **expires_in, which is an integer representing the time to live of the access token.**
- **access_token, which is the access token itself.**
- **refresh_token, which is the token to get a new access token once the previous one expires.**

Device Code Grant Example

Because this has so many moving pieces, a code example is useful. The following is an example Device Code grant the service would receive:

```
POST /device HTTP/1.1
Host: authorization-server.com

client_id={CLIENT_ID}
```

The server responds with the following JSON response:

```
{
  "device_code": "NGU4QWFiNjQ5YmQwNG3YTdmZMEyNzQ3YzQ1YSA",
  "verification_uri": "https://example.com/device",
  "user_code": "BDSD-HQMK",
  "expires_in": 1800,
  "interval": 5
}
```

Now, the device needs to display the URL and user code to the user somehow. While the device waits for the user to enter the code and log in, it will make a POST request every five seconds as specified by the interval returned. This POST request will be made to the token endpoint using a grant type of device_code:

```
POST /token HTTP/1.1
Host: authorization-server.com

grant_type=urn:ietf:params:oauth:grant-type:device_code
&client_id={CLIENT_ID}
&device_code=xxxxxx
```

Once the login is finished, the device makes the POST request to the token endpoint. It gets back the following JSON response:

```
{
  "access_token": "xxxxxxxxxxxx",
  "expires_in": 3600,
  "refresh_token": "xyxyxyxyxyxyxyxyx"
}
```

UMA Grant

The User-Managed Access (UMA) grant type is an authorization standard protocol built as an extension of OAuth 2.0. This grants an access token to the requesting party (a requesting party token, or RPT) to allow access to a resource.

The UMA grant type gives the resource owner control over who can access their protected resources from a centralized Authorization Server without taking into consideration where the resources live. For example, a student can share their private data like health issues and exam scores with their school, teachers, and parents.

Flow for UMA Grant

1. The Resource Server puts resources and their available scopes under Authorization Server protection. Once the resource is registered, the Resource Owner can set policy conditions at the Authorization Server.

2. The Client acting on behalf of the Requesting Party makes an access request to the protected resource (with invalid/no RPT access token).

3. The Resource Server requests permissions bound to that resource from the Authorization Server. The standard doesn't recommend any specific way this request needs to be performed. One example can be found here

4. The Authorization Server returns a permission ticket to the Resource Server.

5. The Resource Server returns Authorization Server URI and permission ticket. The response must contain a WWW-Authenticate header with the value UMA and the following parameters:

- **as_uri: The issuer URI from the Authorization Server's discovery document**
- **ticket: The permission ticket**

6. The Client requests an access token from the Authorization Server's token endpoint by making a POST request with the following parameters (the Client can also redirect the requesting party to the Authorization Server as explained here):

- **grant_type: Must be set to ` urn:ietf:params:oauth:grant-type:uma-ticket`**
- **ticket: The permission ticket received in the previous step**
- **claim_token (optional): A string containing the claim information the format specified by the claim_token_format parameter.**
- **claim_token_format (optional): Specifies the format of the claim_token.**
- **pct (optional): If the Authorization Server issued a PCT (persistent claims token) along with the RPT, this parameter may be included to optimize the process of requesting a new RPT**
- **rpt (optional): An existing RPT, if one is available. This gives the Authorization Server the option to upgrade the RPT instead of assigning a new one.**
- **scope (optional): A space-separated list of scopes.**

7. The Authorization Server issues an RPT after verifying the claims. The response contains the following parameters (scope and refresh_token may be returned as well):

- **token_type, with the value Bearer.**
- **expires_in, with an integer representing the TTL of the access token.**
- **access_token, the access token itself.**
- **pct (optional), a persistent token claim**
- **upgraded (optional), a boolean value to indicate whether an existing RPT was upgraded.**

8. The Client requests for the protected resource with RPT as a bearer token.

9. The Resource Server introspects RPT at the Authorization Server and the Authorization Server returns token introspection status. There isn't a standard recommended way of performing the introspection.

10. The Resource Server gives access to the protected resource.

SAML 2.0 Bearer Grant

The SAML 2.0 Bearer Grant uses a SAML 2.0 assertion for authorization grant as well as client credentials, as described in (Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants)[https://www.rfc-editor.org/rfc/rfc7521]. The following is the flow of using SAML 2.0 assertion for authorization grant.

Flow for SAML 2.0 Bearer Grant

1. The Client sends a POST request to the token endpoint of the Authorization Server with the following parameters:

- **grant_type: This must be set to urn:ietf:params:oauth:grant-type:saml2-bearer**
- **assertion: This parameter must contain the SAML 2.0 assertion, encoded with base64url. The assertion must follow the requirements outlined in RFC 7522**
- **scope (optional): A list of space-delimited scopes**

2. After validating the assertion, the Authorization Server returns the access token. The response format is identical to the one described in the authorization code grant.

OAuth 2.0 Token

The OAuth 2.0 Token Exchange Grant is useful for scenarios where one token needs to be exchanged for another, usually when a user or a service needs to act on behalf of another. For example, a Resource Server might need to make a request to a backend service on behalf of a user. Depending on the way the authentication needs to happen, the flow can be categorized into two types: delegation and impersonation.

- **Delegation: In this case, the Resource Server uses its own credentials to make the request with some annotation that denotes that the request is made on behalf of another user. Here, the Resource Server is in possession of a token on behalf of the user (the subject token), and another for itself (the actor token) and both the tokens are exchanged to get an access token that can be used to call the backend service.**
- **Impersonation: In this case, the Resource Server uses the requesting user's credentials to make the request to the service. The user's token is exchanged for an access token that lets the Resource Server to call the backend service.**

Flow for
OAuth 2.0 Token Exchange Grant

1. The Client makes a POST request to the token endpoint of the Authorization Server. Additionally, client authentication is performed if it's needed, according to the section 2.3 of RFC 6749. The POST request has the following parameters:

- **grant_type: This must be set to urn:ietf:params:oauth:grant-type:token-exchange**
- **subject_token: This is the security token that represents the identity of the user on behalf of whom the request is being made.**
- **subject_token_type: This denotes the type of the subject token. For example, urn:ietf:params:oauth:token-type:access_token to indicate that it's an OAuth 2.0 access token, or urn:ietf:params:oauth:token-type:saml2 for a base64url-encoded SAML 2.0 assertion. The full list can be found here**
- **actor_token and actor_token_type (optional): Similar to the previous two parameters, but these are optional and only required for a delegation flow. The actor_token represents the identity of the acting party - the party that uses the security token on behalf of the subject. The actor_token_type is required only when actor_token is present.**
- **requested_token_type (optional): This denotes the type of the token that is requested. If unspecified, the Authorization Server tries to infer it from the resource and audience parameters.**
- **resource (optional): This should be a URI of the target service where the Client wants to use the requested token. This parameter helps the Authorization Server figure out what policies to apply, the type and content of the token, as well as the encryption, if it's needed.**

- **audience (optional): The logical name of the target service. This is similar to the resource parameter, except this is a logical name that both the Client and the Authorization Server must understand.**
- **scope (optional) : A space-delimited list of scopes.**

2. The server validates the request and in case of successful validation, responds with the following values:

- **access_token: This is the security token issued by the Authorization Server. Although the name of the parameter is access_token, it might not actually be an OAuth 2.0 access token. The type of this token is dictated by the requested_token_type parameter in the request.**
- **issued_token_type: This represents the type of the issued token.**
- **expires_in (optional): This is the time, in seconds, to denote how long the issued token stays valid.**
- **scope (optional): If the scope of the issued token is identical to the scope requested by the Client, this parameter is optional. Otherwise, this is a required parameter.**
- **refresh_token (optional): A refresh token can be issued in the cases where the Client needs to access the resource even when the original issued token expires. For example, if the Client needs long-term access to a service even when the user is offline or logged-out.**

Conclusion

You should now have a good sense of the different OAuth 2.0 grant types, from their basic definition to their usage, as well as the flow used for obtaining an access token from each one.

FusionAuth

Auth. Built for Devs, by Devs.

FusionAuth is the authentication and authorization platform built for developers, by developers. For technical leaders creating products for external users, it solves the problem of building essential user security without distracting from the primary application.

Learn more at **FusionAuth.io**