International Journal of Leading Research Publication (IJLRP)

E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

OAuth 2.0 Evolution, Grant Types and Best Practices for Secure Authorization in Web, Mobile, and API Platforms

Arun Neelan

Independent Researcher
PA, USA
arunneelan@yahoo.co.in

Abstract

OAuth 2.0, as defined in [1], is a widely adopted authorization protocol across various platforms, including web, mobile, and API-driven applications. It is built around access tokens, which define the scope, lifetime, and other critical access attributes, offering a secure and flexible way of managing resource access. This paper reviews the evolution, core components, and different token types as outlined in [1], focusing on their role in secure access management. Additionally, the paper provides a detailed analysis of the four primary grant types—Authorization Code, Implicit, Resource Owner Password Credentials, and Client Credentials—highlighting their functionality, advantages, and potential limitations. By examining these elements, this paper aims to guide all relevant stakeholders –including technologists, product team, security professionals, compliance team, end users and other partners -- in selecting the most suitable grant type for specific OAuth 2.0 implementations.

Keywords: OAuth2.0, Authorization Protocols, Secure Authorization, API Security, Grant Types, OAuth Flows, Authorization Code Flow, Implicit Flow, Resource Owner Password Credentials Flow, Client Credentials Flow

I. INTRODUCTION

OAuth 2.0, as specified in [1], is an authorization protocol that allows applications to access user data across various platforms without exposing sensitive user credentials. Traditional models require applications to manage user credentials (such as usernames and passwords), which raises significant security concerns. OAuth 2.0 addresses this by delegating the authorization process to a trusted authorization server, which issues access tokens that grant limited, scoped access to protected resources. This approach enhances security by ensuring that the client application never needs to manage or store user credentials directly.

The necessity for OAuth 2.0 arose from the limitations of earlier authorization systems, particularly OAuth 1.0, which was complex due to its reliance on cryptographic signing for token verification, making it challenging to securely manage tokens and secrets. OAuth 2.0, as outlined in [1], simplifies this process by providing a more flexible and easier-to-implement authorization framework. This simplicity has contributed to its widespread adoption across web, mobile, and API-driven applications. However, the flexibility of OAuth 2.0 introduces new challenges, particularly in the secure management of access tokens and in selecting the most appropriate grant type for different use cases.

As outlined in [1] OAuth 2.0 relies on four primary grant types: Authorization Code, Implicit, Resource Owner Password Credentials, and Client Credentials. Each grant type has its unique characteristics, strengths, weaknesses, and ideal use cases. This paper examines each of these grant types as defined in the RFC, offering an in-depth explanation of how they function, their associated security implications, and guidance on when to use them to ensure secure and effective OAuth 2.0 implementation.

II. TRADITIONAL CLIENT-SERVER AUTHENTICATION MODEL

In this model, as mentioned in [1], the resource owner shares its credentials with the client to gain access to restricted resources on the server.

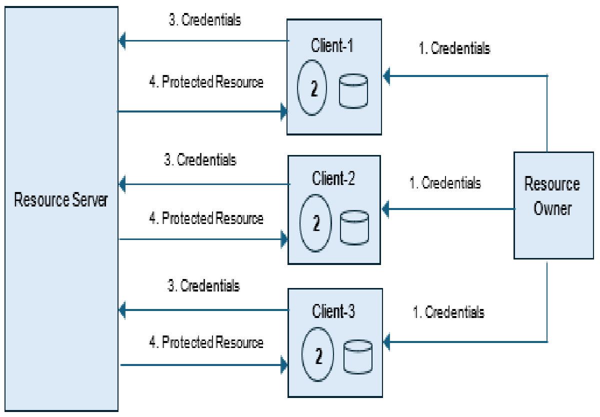

Fig. 1 Traditional Client-Server Authentication Model

Flow Overview:

1. The Resource Owner shares their credentials directly with clients.
2. Clients store the credentials and use them to access the restricted resource, acting as the Resource Owner.
3. The client submits the Resource Owner’s credentials to access the resource.
4. The Resource Server validates the credentials and grants access if valid.

There are specific issues with this model.

1. All clients need to store the credentials at their end. Any compromise to it at any client would require the resource owner to redo the process again.
2. The Resource Owner will be unable to restrict access to a limited subset of resources, and therefore, clients will have broader access.
3. Resource owners can't revoke access for a single client without affecting all clients. To do so, they must change and update the password for all valid clients.

III. OAUTH OVERVIEW AND THE EVOLUTION TO OAUTH 2.0

OAuth addresses these issues by introducing an authorization layer that separates the roles of the client and the resource owner. In OAuth (both OAuth 1.0 and OAuth 2.0), the client requests access to resources controlled by the resource owner and hosted on the resource server. While OAuth 1.0 relies on cryptographic signatures and tokens for secure communication, OAuth 2.0 simplifies the process with access tokens, each issued with a different set of credentials from those of the resource owner.

IV. OAUTH 2.0 ROLES

As outlined in [1] OAuth 2.0 defines four key roles:

- *Resource Owner:* An entity capable of granting access to a protected resource. When the resource owner is an individual, it is referred to as the end-user.
- *Resource Server:* The server hosting protected resources, responsible for accepting and responding to requests for these resources using access tokens.
- *Client:* The third-party application requesting access to protected resources.
- *Authorization Server:* The server responsible for issuing access tokens to the client after authenticating the resource owner and obtaining their consent. The authorization server and resource server may either be the same entity or separate systems. A single authorization server can issue access tokens that are accepted by multiple resource servers.

V. OAUTH 2.0 ABSTRACT FLOW OVERVIEW

The flow illustrated in the diagram below describes the interaction between four roles.

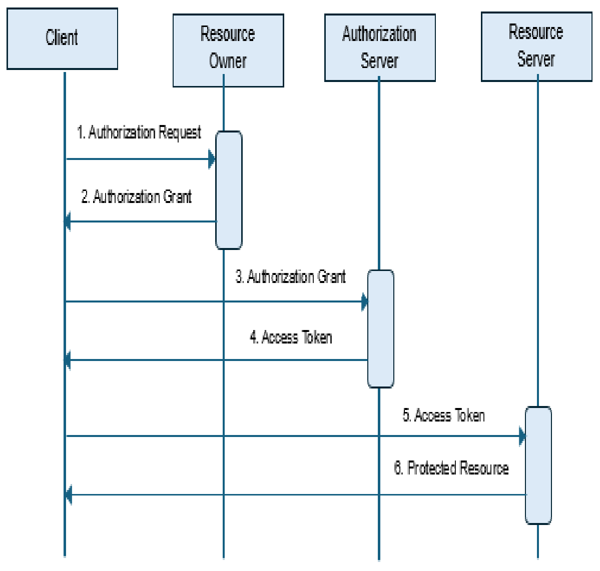

Fig. 2 OAuth Abstract Flow

1. The client requests authorization from the resource owner to access protected resources. This request can be made directly to the resource owner (e.g., asking for credentials), or more securely, through the authorization server, which acts as an intermediary to handle authentication and consent.
2. The client receives an authorization grant, a credential that represents the resource owner's consent. This grant is issued using one of the four standard grant types (authorization code, implicit, resource owner password credentials, client credentials), or an extension grant type. The

specific grant type used depends on the method the client employs to request authorization and the types supported by the authorization server.

3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
4. The authorization server authenticates the client, validates the authorization grant, and, if valid, issues an access token.
5. The client requests the protected resource from the resource server and authenticates by presenting the access token.
6. The resource server validates the access token and, if valid, serves the request.

VI. OAUTH TOKENS

The two types of OAuth tokens, access tokens and refresh tokens, are critical components in the OAuth 2.0, which enables secure and delegated access to protected resources.

Access Tokens are credentials issued to clients by the authorization server, with the resource owner's approval. The client uses the access token to request access to protected resources hosted on the resource server. An access token is a string that represents the authorization granted to the client, defining the specific scope and duration of access. [1] Since any party in possession of an access token can access the associated resources, these tokens must be protected in storage and in transport. [4], [5]

Refresh Tokens are credentials used to obtain access tokens. Refresh tokens are issued to the clients by the authorization server and are used to obtain a new access token when the current one becomes invalid or expires. Issuing a refresh token is optional at the discretion of the authorization server. If the authorization server issues a refresh token, it is included when an access token is issued. A refresh token is a string representing the authorization granted to the client by the resource owner. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers. [4]

Flow Overview:

1. The client requests authorization from the resource owner to access protected resources. This request can be made directly to the resource owner (e.g., asking for credentials), or more securely, through the authorization server, which acts as an intermediary to handle authentication and consent.
2. The client receives an authorization grant, a credential that represents the resource owner's consent. This grant is issued using one of the four standard grant types (authorization code, implicit, resource owner password credentials, client credentials), or an extension grant type. The specific grant type used depends on the method the client employs to request authorization and the types supported by the authorization server.
3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
4. The authorization server authenticates the client, validates the authorization grant, and, if valid, issues an access token.
5. The client requests the protected resource from the resource server and authenticates by presenting the access token.
    a) The resource server returns an error if the token is invalid or expired.

b) The client requests a new access token by authenticating with the authorization server and presenting the refresh token. The authentication requirements for the client depend on both the client type and the policies set by the authorization server.
c) The authorization server authenticates the client, validates the refresh token, and, if the token is valid, issues a new access token and optionally a new refresh token.
d) The client authenticates with the latest obtained access token.

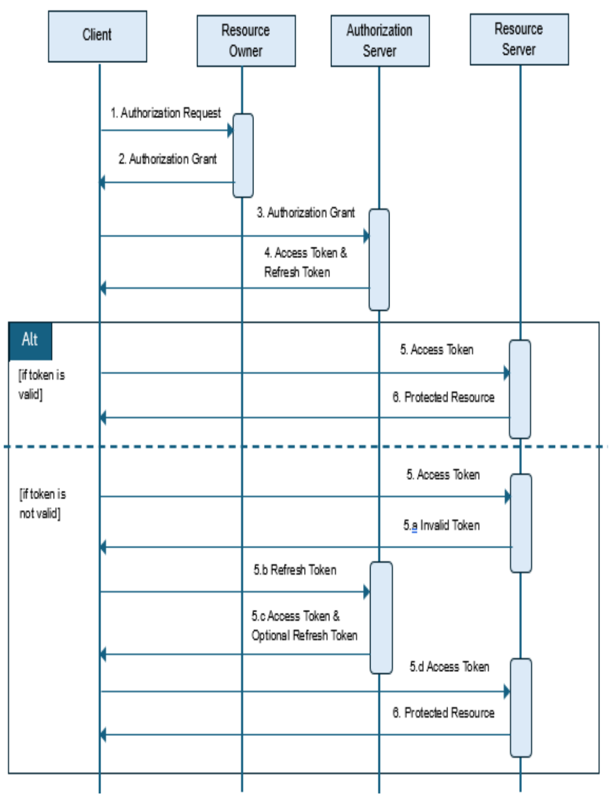

Fig. 3 Refresh Token Usage Flow

6. The resource server validates the access token and, if valid, serves the request.

Security Considerations and Best Practices

- *Always Use TLS (https):* Clients must always use TLS (https) or equivalent transport security when making requests with tokens. Failing to do so exposes the token to numerous attacks that could give attackers unintended access. [3], [4]
- *Token Security:* Token servers should issue short-lived (one hour or less) access tokens, particularly when issuing tokens to clients that run within a web browser or other environments where information leakage may occur. Using short-lived access tokens can reduce the impact of them being leaked. Client implementations must ensure that access tokens are not leaked to unintended parties, as they will be able to use them to gain access to protected resources. [1], [4]
- *Token Revocation:* Implement token revocation mechanisms, enabling users or administrators to invalidate access tokens and authorization codes if there is any suspicion of compromise. [4]

- *Scope Limitation:* Token servers should issue access tokens that contain an audience restriction, scoping their use to the intended relying party or set of relying parties. [1]
- *Token Integrity:* The authorization server must ensure that access and refresh tokens cannot be generated, modified, or guessed to produce valid tokens by unauthorized parties. [1], [4]

VII. OAuth 2.0 Client Types

OAuth 2.0 defines different client types based on their ability to securely store credentials and their interaction with the authorization and resource servers.

- *Confidential Clients:* Clients that can securely store credentials (e.g., client secrets) because they run in a trusted environment, such as a web server or backend server.
- *Public Clients:* Clients that cannot securely store credentials due to their environment, such as a mobile application, single-page web application (SPA), or a client-side application.
- *Third-Party Clients:* External applications or services requesting access to user data from another service, typically requiring user authorization.

VIII. Authorization Grants in OAuth 2.0

In OAuth 2.0, outlined in [1], the term authorization grant refers to a method for obtaining an access token. The grant is typically issued by an authorization server after the resource owner (user) has authorized the client (third-party application) to access specific resources on their behalf. OAuth defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials. While all flows ultimately result in the issuance of an access token, the paths to token issuance differ in terms of user involvement, security considerations, and the specific use cases they serve.

The Authorization Code Grant: Full User Consent and Secure Flows

The authorization code grant type is the most secure and widely adopted flow in OAuth 2.0. It is primarily used to obtain both access tokens and refresh tokens, making it ideal for web applications and scenarios involving confidential clients those capable of securely storing a client secret. This flow is redirection-based, requiring the client to interact with the resource owner's user-agent (typically a web browser) and handle incoming requests from the authorization server via redirection.

This Authorization Code Flow can also be used by public clients (such as mobile apps or single-page apps) that are unable to securely store client secrets. However, in these cases, it is recommended to follow best practices—such as implementing PKCE—to enhance security and mitigate risks like authorization code interception. [2] These best practices will be discussed in the following section.

1) Flow Overview

- *Registration: The c*lient registers with the authorization server and obtains client-id and client-secret.

1. The client directs the resource owner (user) to the authorization server's authorization endpoint.
2. The user authenticates (e.g., via username and password).
3. The authorization server validates the credentials and requests consent to access specific resources.
4. The user grants permission to the authorization server to let the client access specific resources.

5. The authorization server redirects the user back to the client, passing an authorization code as part of the URL. This code represents the user's consent.
6. The client then exchanges the authorization code for an access token by making a request to the authorization server's token endpoint, including its client credentials for authentication.
7. The authorization server provides the client the access token and refresh token to access the resources.

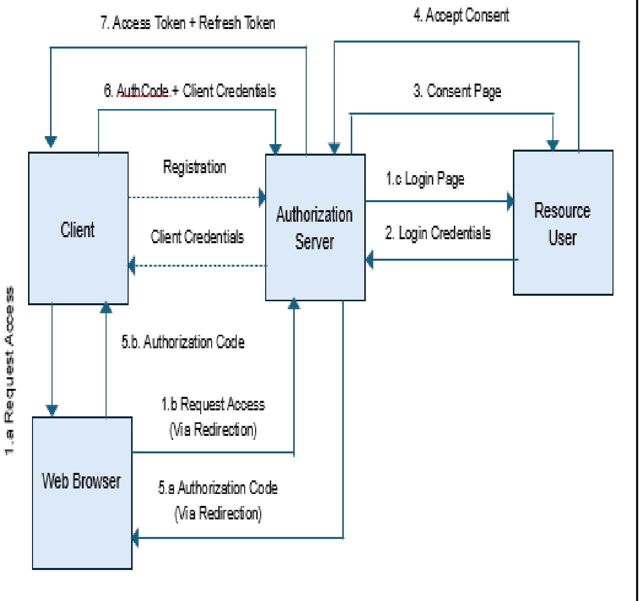

Fig. 4 Authorization Code Grant Flow

2) Use Case

The Authorization Code Grant is ideal for a secure, user-consented access to third-party services, especially in web, mobile, and enterprise applications, and it's commonly used in SSO systems, cloud service integrations, and other high-security scenarios.

3) Security Considerations and Best Practices

- *Client Secret Protection:* The client secret, which authenticates the client to the authorization server during the token exchange, must be stored securely and kept confidential to prevent unauthorized clients from acquiring tokens. [1], [4]
- *Authorization Code Validity:* Authorization codes must be short lived and single use. If the authorization server observes multiple attempts to exchange an authorization code for an access token, the authorization server should attempt to revoke all access tokens already granted based on the compromised authorization code. [1], [4]
- *Use the state parameter:* Always include the state parameter to mitigate CSRF (Cross-Site Request Forgery) attacks. The state value should be unpredictable, securely tied to the user's session, and verified upon the redirect to ensure the integrity of the request. [4]
- *Redirection URI Validation:* Ensure strict validation of the redirect URI during authorization code issuance. The URI must exactly match the one registered with the authorization server to prevent attackers from redirecting tokens to unauthorized destinations. [1], [4]

- *Use PKCE (Proof Key for Code Exchange):* Implement PKCE for public clients (e.g., mobile apps, SPAs) to mitigate authorization code interception risks. PKCE, as specified in RFC 7636, ensures that even if an attacker intercepts the authorization code, they cannot exchange it for an access token without the correct code verifier.[2]

The Implicit Grant: Quick Access but with Security Trade-offs

The Implicit Grant is used to obtain access tokens without refresh tokens and is designed for public clients, such as JavaScript-based applications running in a browser. This flow is redirection-based, where the client interacts with the user's browser and receives the access token directly in the URL after authorization. Unlike the Authorization Code Grant, the Implicit Grant combines authorization and token issuance into a single step. It does not require client authentication and relies on the registered redirect URI. However, since the access token is included in the URL, it may be exposed to the resource owner and other apps on the same device.

4) Flow Overview

- *Registration:* The client registers with the authorization server and obtains client-id.

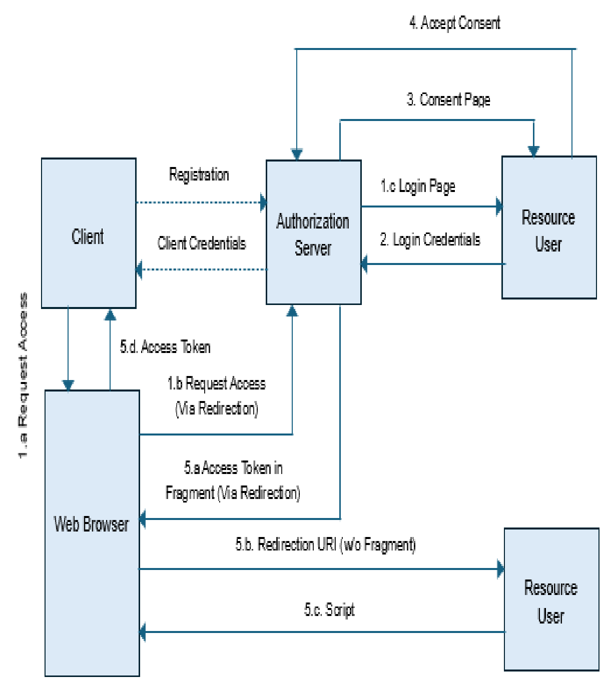

Fig. 5 Implicit Grant Flow

1. The client directs the user to the authorization server’s authorization endpoint, specifying that it requires an access token.
2. The user authenticates (e.g., via username and password).
3. The authorization server validates the credentials and requests consent to access specific resources.
4. The user grants permission to the authorization server to let the client access specific resources.
5. The authorization server redirects the user back to the client, passing an authorization code as part of the URL. This code represents the user's consent.

6. The client then exchanges the authorization code for an access token by making a request to the authorization server's token endpoint, including its client credentials for authentication.
7. The authorization server provides the client the access token and refresh token to access the resources.

5) Use Case

The Implicit Grant is most appropriate for client-side applications, particularly SPAs, where the client cannot safely store a client secret. However, due to security trade-offs, it is generally not recommended for applications handling sensitive or critical data.

6) Security Considerations and Best Practices

- *Use the state parameter:* Always include the state parameter to mitigate CSRF (Cross-Site Request Forgery) attacks. The state value should be unpredictable, securely tied to the user's session, and verified upon the redirect to ensure the integrity of the request. [1], [4]
- *Redirection URI Validation:* Ensure strict validation of the redirect URI during access token issuance. The URI must exactly match the one registered with the authorization server to prevent attackers from redirecting tokens to unauthorized destinations. [1], [4]
- *Protect Against XSS:* Implement strong Cross-Site Scripting (XSS) protections and Content Security Policy (CSP) headers to prevent attackers from injecting malicious scripts that could steal access tokens from the URL fragment or browser storage. [4]
- *Secure Token Storage:* Store tokens in sessionStorage instead of localStorage to limit their lifespan and reduce exposure in case of XSS attacks. Avoid storing tokens in any persistent client-side storage. [1], [4]
- *Avoid Token Exposure in URLs:* Since access tokens are passed in the URL fragment, ensure that sensitive information is not logged or cached in browser history. Control redirects and caching behavior to limit the risk of token leakage through browser history, referrers, or logs. [1], [4]
- *Limit CORS Access to Trusted Origins:* Configure CORS (Cross-Origin Resource Sharing) headers to only allow token requests from trusted origins. This prevents unauthorized applications from accessing the authorization server. [1], [4]

By adhering to these best practices, the risks associated with the Implicit Grant type can be minimized in OAuth 2.0 implementation. However, for stronger security, it's recommended to use the Authorization Code Flow with PKCE, which offers a more robust and secure alternative.

The Resource Owner Password Credentials Grant: Simplified for Trusted Clients

The Resource Owner Password Credentials Grant, often referred to as the password grant, simplifies the authorization flow by allowing the client to directly request an access token by submitting the user's username and password to the authorization server. This grant type is only recommended for trusted clients, where the user has a high level of trust in the application (e.g., trusted apps like mobile apps or desktop clients).

1) Flow Overview

- *Registration:* The client registers with the authorization server and obtains client-id and client-secret.

1. The resource owner provides the client with its username and password.
2. The client sends a request to the authorization server's token endpoint, with the user's credentials and its own client credentials (client id and secret).
3. The authorization server validates the credentials and issues an access token and optionally a refresh token.

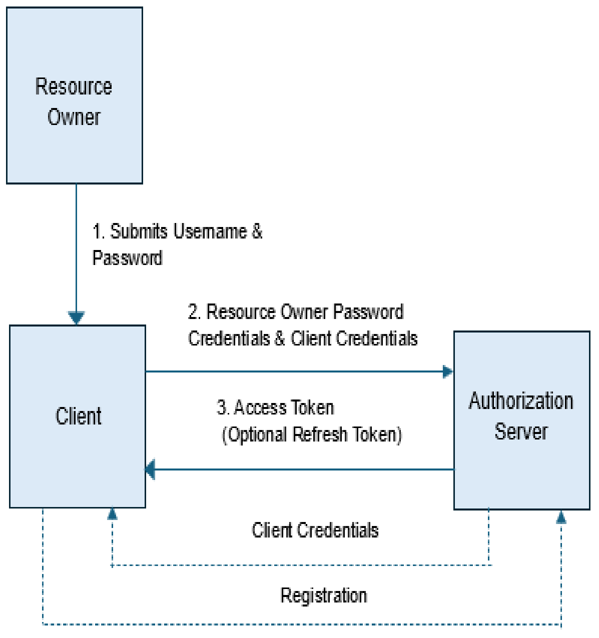
Resource Owner
1. Submits Username & Password
2. Resource Owner Password Credentials & Client Credentials
Client
3. Access Token (Optional Refresh Token)
Authorization Server
Client Credentials
Registration

Fig. 6 Resource Owner Password Credentials Grant Flow

2) Use Case

The Resource Owner Password Credentials Grant is typically used in scenarios where the client is fully trusted by the user, such as mobile apps or applications that the user has explicitly downloaded and installed. This should never be used with third-party or public applications – choose Authorization Code Grant. However, its use should be minimized due to the risks involved in exposing user credentials.

3) Security Considerations and Best Practices

- *Limit Usage to Trusted Clients:* Use the ROPC Grant only with trusted, first-party applications where users have a high level of trust to minimize the risk of exposing user credentials to untrusted parties. [1], [4]
- *Store Credentials Securely:* Ensure user credentials are stored securely on the client side, using encrypted storage solutions to protect them from unauthorized access in case of a breach. [4]
- *Implement Strong Password Policies:* Enforce strong password requirements, such as minimum length and complexity, to reduce the likelihood of unauthorized access through weak passwords.

The Client Credentials Grant: For Machine-to-Machine Authentication

The Client Credentials Grant is used for machine-to-machine (M2M) authentication, where the client is a service that needs to access resources without user intervention. This flow is typically used for server-to-server communication, such as accessing an API on behalf of an application rather than an individual user.

1) Flow Overview

- *Registration:* The client registers with the authorization server and obtains client-id and client-secret.

1. The client authenticates with the authorization server by sending its client id and secret.
2. The authorization server validates the client's credentials and issues an access token.

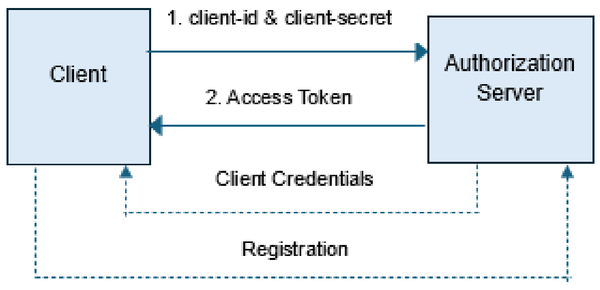

Fig. 7 Client Credentials Grant Flow

2) Use Case

The Client Credentials Grant is ideal for backend services, APIs, or microservices that need access to their own resources or a user's resources without requiring user involvement. For example, a payment service accessing its data or a communication service sending messages on behalf of an organization.

3) Security Considerations and Best Practices

- *Client Secret Protection:* The client secret, which authenticates the client to the authorization server during the token exchange, must be stored securely and kept confidential to prevent unauthorized clients from acquiring tokens. [1], [4]

IX. Conclusion

This review has comprehensively analyzed the various authorization grant types in the OAuth 2.0 framework, highlighting the security considerations and best practices for each flow. Key takeaways include:

- The Authorization Code Grant is the most secure and widely adopted flow. It offers features like client authentication and refresh tokens to enhance security. It is especially suitable for web applications with high-security requirements.
- The Implicit Grant, designed for public clients, requires careful management of security trade-offs, such as the risk of access token exposure in the URL.
- The Resource Owner Password Credentials Grant should be limited to highly trusted clients as it directly exposes user credentials, increasing the risk of compromise.

- The Client Credentials Grant is ideal for machine-to-machine authentication scenarios where a service needs to access resources without any user involvement.

This review provides guidance on selecting the most suitable OAuth 2.0 grant types for various use cases, assisting stakeholders in adopting appropriate implementations across modern web, mobile, and API platforms.

REFERENCES

[1] D. Hardt, "The OAUTH 2.0 Authorization Framework," Oct. 2012. doi: 10.17487/rfc6749. Available: https://doi.org/10.17487/rfc6749

[2] J. Bradley and N. Agarwal, "Proof key for code exchange by OAuth public clients," Sep. 2015. doi: 10.17487/rfc7636. Available: https://doi.org/10.17487/rfc7636

[3] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," Aug. 2008. doi: 10.17487/rfc5246. Available: https://doi.org/10.17487/rfc5246

[4] M. McGloin and P. Hunt, "OAUTH 2.0 threat model and security Considerations," Jan. 2013. doi: 10.17487/rfc6819. Available: https://doi.org/10.17487/rfc6819

[5] M. Jones and D. Hardt, "The OAUTH 2.0 Authorization Framework: Bearer Token usage," Oct. 2012. doi: 10.17487/rfc6750. Available: https://doi.org/10.17487/rfc6750

[6] S. Dronia and M. Scurtescu, "OAUTh 2.0 token revocation," Aug. 2013. doi: 10.17487/rfc7009. Available: https://doi.org/10.17487/rfc7009

[7] "OAUth 2.0 token introspection," Oct. 2015. doi: 10.17487/rfc7662. Available: https://doi.org/10.17487/rfc7662