Network Working Group	E. Hammer-Lahav, Ed.
Internet-Draft	Yahoo!
Obsoletes: 5849 (if approved)	D. Recordon
Intended status: Standards Track	Facebook
Expires: November 19, 2011	D. Hardt
	Microsoft
	May 18, 2011

TOC

The OAuth 2.0 Authorization Protocol draft-ietf-oauth-v2-16

Abstract

The OAuth 2.0 authorization protocol enables a third-party application to obtain limited access to an HTTP service, either on behalf of an end-user by orchestrating an approval interaction between the end-user and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as “work in progress.”

This Internet-Draft will expire on November 19, 2011.

Table of Contents

---

1. Introduction

TOC

In the traditional client-server authentication model, the client accesses a protected resource on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to protected resources, the resource owner shares its credentials with the third-party. This creates several problems and limitations:

- Third-party applications are required to store the resource-owner's credentials for future use, typically a password in clear-text.
- Servers are required to support password authentication, despite the security weaknesses created by passwords.
- Third-party applications gain overly broad access to the resource-owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- Resource owners cannot revoke access to an individual third-party without revoking access to all third-parties, and must do so by changing their password.

OAuth addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued

a different set of credentials than those of the resource owner.

Instead of using the resource owner's credentials to access protected resources, the client obtains an access token - a string denoting a specific scope, duration, and other access attributes. Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

For example, a web end-user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with a server trusted by the photo sharing service (authorization server) which issues the printing service delegation-specific credentials (access token).

This specification is designed for use with HTTP **[RFC2616]**. The use of OAuth with any transport protocol other than HTTP is undefined.

TOC

1.1. Roles

OAuth includes four roles working together to grant and provide access to protected resources - access restricted resources which require authentication to access:

resource owner
: An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

resource server
: The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

client
: An application making protected resource requests on behalf of the resource owner and with its authorization.

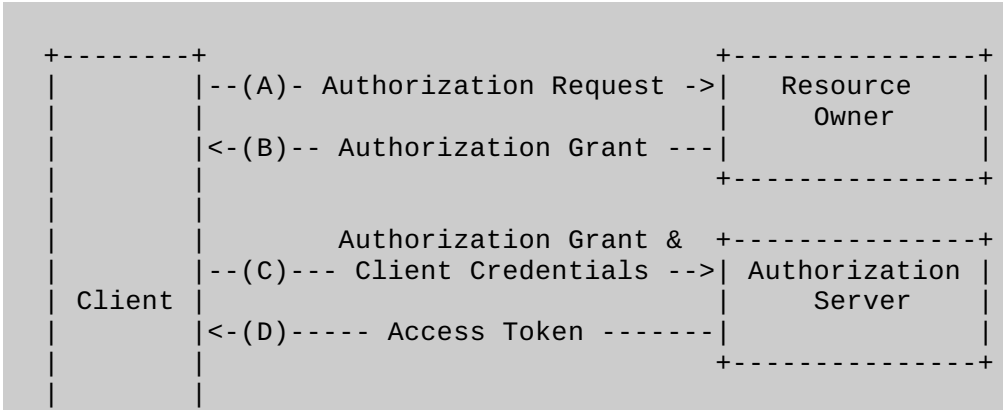
authorization server
: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The interaction between the authorization server and resource server is beyond the scope of this specification. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

TOC

1.2. Protocol Flow

When interacting with the authorization server, the client identifies itself using a set of client credentials which include a client identifier and other authentication attributes. The means through which the client obtains its credentials are beyond the scope of this specification, but typically involve registration with the authorization server.



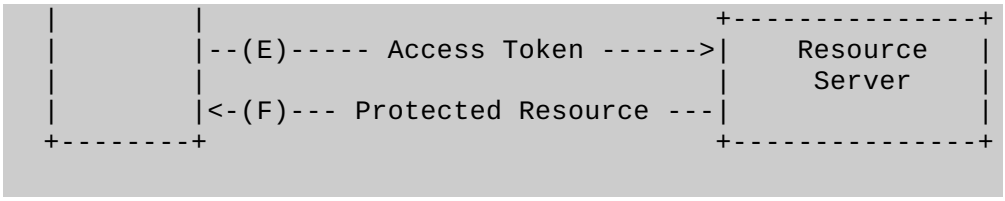


Figure 1: Abstract Protocol Flow

The abstract flow illustrated in **Figure 1** describes the interaction between the four roles and includes the following steps:

(A)
: The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via an intermediary such as an authorization server.

(B)
: The client receives an authorization grant which represents the authorization provided by the resource owner. The authorization grant type depends on the method used by the client and supported by the authorization server to obtain it.

(C)
: The client requests an access token by authenticating with the authorization server using its client credentials (prearranged between the client and authorization server) and presenting the authorization grant.

(D)
: The authorization server validates the client credentials and the authorization grant, and if valid issues an access token.

(E)
: The client requests the protected resource from the resource server and authenticates by presenting the access token.

(F)
: The resource server validates the access token, and if valid, serves the request.

TOC

1.3. Access Token

An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information, or self-contain the authorization information in a verifiable manner (i.e. a token string consisting of some data and a signature). Additional authentication credentials may be required in order for the client to use a token.

The access token provides an abstraction layer, replacing different authorization constructs (e.g. username and password) with a single token understood by the resource server. This abstraction enables issuing access tokens more restrictive than the authorization grant used to obtain them, as well as removing the resource server's need to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of utilization (e.g. cryptographic properties) based on the resource server security requirements. Access token attributes and the methods used to access protected resources are beyond the scope of this specification and are defined by companion specifications.

TOC

1.4. Authorization Grant

An authorization grant is a general term used to describe the intermediate credentials representing the resource owner authorization (to access its protected resources), and serves as an abstraction layer. An authorization grant is used by the client to obtain an access token.

This specification defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials, as well as an extensibility mechanism for defining additional types.

TOC

1.4.1. Authorization Code

The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server (via its user-agent as defined in **[RFC2616]**), which in turn directs the resource owner back to the client with the authorization code.

Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner and obtains authorization. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client.

The authorization code provides a few important security benefits such as the ability to authenticate the client and issuing the access token directly to the client without potentially exposing it to others, including the resource owner.

TOC

1.4.2. Implicit

When an access token is issued to the client directly as the result of the resource owner authorization, without an intermediary authorization grant (such as an authorization code), the grant is considered implicit.

When issuing an implicit grant, the authorization server cannot verify the identity of the client, and the access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application) since it reduces the number of round trips required to obtain an access token.

TOC

1.4.3. Resource Owner Password Credentials

The resource owner password credentials (e.g. a username and password) can be used directly as an authorization grant to obtain an access token. The credentials should only be used when there is a high degree of trust between the resource owner and the client (e.g. its computer operating system or a highly privileged application), and when other authorization grant types are not available (such as an authorization code).

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. Unlike the HTTP Basic authentication scheme defined in **[RFC2617]**, this grant type (when combined with a refresh token) eliminates the need for the client to store the resource-owner credentials for future use.

TOC

1.4.4. Client Credentials

The client credentials can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used as an authorization grant typically when the client is acting on its own behalf (the client is also the resource owner).

1.4.5. Extensions

TOC

Additional grant types may be defined to provide a bridge between OAuth and other protocols. For example, **[I-D.ietf-oauth-saml2-bearer]** defines a **SAML 2.0** [OASIS.saml-core-2.0-os] bearer assertion grant type, which can be used to obtain an access token.

1.5. Refresh Token

TOC

A refresh token is optionally issued by the authorization server to the client together with an access token. The client can use the refresh token to request another access token based on the same authorization, without having to involve the resource owner again, or having to retain the original authorization grant used to obtain the initial access token.

A refresh token is a string representing the authorization granted to the client by the resource owner. The string is usually opaque to the client. The token may denote an identifier used to retrieve the authorization information, or self-contain the authorization information in a verifiable manner. The refresh token is bound to the client it was issued to, and its usage requires client authentication.

The refresh token can be used to obtain a new access token when the current access token expires (access tokens may have a shorter lifetime than authorized by the resource owner), no longer valid, or to obtain additional access tokens with identical or narrower scope.

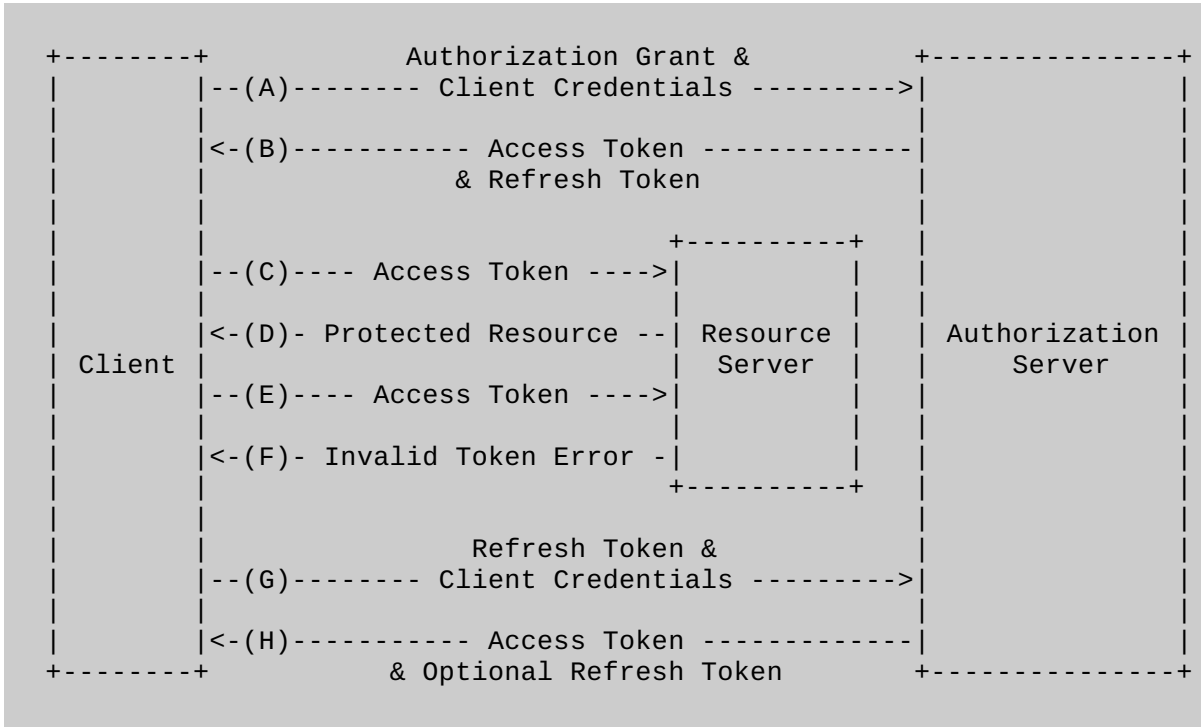


Figure 2: Refreshing an Expired Access Token

The flow illustrated in **Figure 2** includes the following steps:

(A)
: The client requests an access token by authenticating with the authorization server using its client credentials, and presenting an authorization grant.

(B)
: The authorization server validates the client credentials and the authorization grant, and if valid issues an access token and a refresh token.

(C)

The client makes a protected resource requests to the resource server by presenting the access token.

(D)
The resource server validates the access token, and if valid, serves the request.

(E)
Steps (C) and (D) repeat until the access token expires. If the client knows the access token expired, it skips to step (G), otherwise it makes another protected resource request.

(F)
Since the access token is invalid, the resource server returns an invalid token error.

(G)
The client requests a new access token by authenticating with the authorization server using its client credentials, and presenting the refresh token.

(H)
The authorization server validates the client credentials and the refresh token, and if valid issues a new access token (and optionally, a new refresh token).

TOC

1.6. Document Structure

This specification is organized into the following sections:

- Section 2 - describes the two endpoints used to obtain and utilize the various authorization grant types.
- Section 3 - describes client identification and authentication in general, and provides one such method for client authentication using password credentials.
- Section 4 - describes the complete flow for each authorization grant type, including requesting authorization, authorization response, and requesting an access token.
- Section 5 - describes the common access token response used for all non-implicit authorization grant types.
- Section 6 - describes the use of a refresh token to obtain additional access tokens using the same resource owner authorization.
- Section 7 - describes how access tokens are used to access protected resources.
- Section 8 - describes how to extend certain elements of the protocol.
- Section 9 - provides a security analysis of the protocol.

TOC

1.7. Notational Conventions

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this specification are to be interpreted as described in **[RFC2119]**.

This specification uses the Augmented Backus-Naur Form (ABNF) notation of **[RFC5234]**.

Unless otherwise noted, all the protocol parameter names and values are case sensitive.

TOC

2. Protocol Endpoints

The authorization process utilizes two endpoints (HTTP resources):

- Authorization endpoint - used to obtain authorization from the resource owner via user-agent redirection.
- Token endpoint - used to exchange an authorization grant for an access token, typically with client authentication.

Not every authorization grant type utilizes both endpoints. Extension grant types MAY define additional endpoints as needed.

2.1. Authorization Endpoint

TOC

The authorization endpoint is used to interact with the resource owner and obtain authorization which is expressed explicitly as an authorization code (exchanged for an access token), or implicitly by direct issuance of an access token.

The authorization server MUST first verify the identity of the resource owner. The way in which the authorization server authenticates the resource owner (e.g. username and password login, session cookies) is beyond the scope of this specification.

The means through which the client obtains the location of the authorization endpoint are beyond the scope of this specification but is typically provided in the service documentation. The endpoint URI MAY include a query component as defined by **[RFC3986]** section 3, which MUST be retained when adding additional query parameters.

Since requests to the authorization endpoint result in user authentication and the transmission of clear-text credentials (in the HTTP response), the authorization server MUST require the use of a transport-layer security mechanism when sending requests to the authorization endpoint. The authorization server MUST support TLS 1.2 as defined in **[RFC5246]**, and MAY support additional transport-layer mechanisms meeting its security requirements.

The authorization server MUST support the use of the HTTP `GET` method **[RFC2616]** for the authorization endpoint, and MAY support the use of the `POST` method as well.

The REQUIRED `response_type` request parameter is used to identify which grant type the client is requesting: authorization code or implicit, described in **Section 4.1.1** and **Section 4.2.1** respectively. If the request is missing the `response_type` parameter, the authorization server SHOULD return an error response as described in **Section 4.1.2.1**.

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server SHOULD ignore unrecognized request parameters.

Request and response parameters MUST NOT repeat more than once, unless noted otherwise.

2.1.1. Redirection URI

TOC

The client directs the resource owner's user-agent to the authorization endpoint and includes a redirection URI to which the authorization server will redirect the user-agent back once authorization has been obtained (or denied). The client MAY omit the redirection URI if one has been established between the client and authorization server via other means, such as during the client registration process.

The redirection URI MUST be an absolute URI and MAY include a query component, which MUST be retained by the authorization server when adding additional query parameters.

The authorization server SHOULD require the client to pre-register their redirection URI or at least certain components such as the scheme, host, port and path. If a redirection URI was registered, the authorization server MUST compare any redirection URI received at the authorization endpoint with the registered URI.

The authorization server SHOULD NOT redirect the user-agent to unregistered or untrusted URIs to prevent the endpoint from being used as an open redirector. If no valid redirection URI is available, the authorization server SHOULD inform the resource owner directly of the error.

2.2. Token Endpoint

TOC

The token endpoint is used by the client to obtain an access token by authenticating with the

authorization server and presenting its authorization grant or refresh token. The token endpoint is used with every authorization grant except for the implicit grant type (since an access token is issued directly).

The means through which the client obtains the location of the token endpoint are beyond the scope of this specification but is typically provided in the service documentation. The endpoint URI MAY include a query component, which MUST be retained when adding additional query parameters.

Since requests to the token endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server MUST require the use of a transport-layer security mechanism when sending requests to the token endpoint. The authorization server MUST support TLS 1.2 as defined in **[RFC5246]**, and MAY support additional transport-layer mechanisms meeting its security requirements.

The token endpoint requires client authentication as described in **Section 3**. The authorization server MAY accept any form of client authentication meeting its security requirements. The client MUST NOT use more than one authentication method in each request.

The client MUST use the HTTP `POST` method when making access token requests.

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server SHOULD ignore unrecognized request parameters.

Request and response parameters MUST NOT repeat more than once, unless noted otherwise.

---

3. Client Authentication

TOC

Client credentials are used to identify and authenticate the client. The client credentials include a client identifier - a unique string issued to the client to identify itself to the authorization server. The client identifier is not a secret, it is exposed to the resource owner, and MUST NOT be used alone for client authentication. Client authentication is accomplished via additional means such as a matching client password.

The methods through which the client obtains its client credentials are beyond the scope of this specification. However, the client registration process typically includes gathering relevant information which is used to educate the resource owner about the client when requesting authorization.

Due to the nature of some clients, the authorization server should not make assumptions about the confidentiality of client credentials without establishing trust with the client. The authorization server SHOULD NOT issue client credentials to clients incapable of keeping their credentials confidential (typically determined during the client registration process).

In addition, the authorization server MAY allow unauthenticated access token requests when the client identity does not matter (e.g. anonymous client) or when the client identity is established via other means. For readability purposes only, this specification is written under the assumption that the authorization server requires some form of client authentication. However, such language does not affect the authorization server's discretion in allowing unauthenticated client requests.

3.1. Client Password Authentication

TOC

[[Pending Consensus]]

The HTTP Basic authentication scheme as defined in **[RFC2617]** MAY be used to authenticate clients issued with a shared symmetric secret together with the client identifier. The client identifier is used as the username, and the secret is used as the password.

When using the HTTP Basic authentication scheme, the client identifier is included twice in the request (in the `Authorization` header and in the `client_id` parameter). The authorization

server MUST ensure the two identifiers belong to the same client.

For example (extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&client_id=s6BhdRkqt3&
code=i1WsRn1uB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

Alternatively, the authorization server MAY allow including the client secret in the request body using the following parameter:

client_secret
: REQUIRED. The client secret.

The use of the `client_secret` parameter is NOT RECOMMENDED, and should be limited to clients unable to directly utilize the HTTP Basic authentication scheme.

For example (extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&client_id=s6BhdRkqt3&
client_secret=gX1fBat3bV&code=i1WsRn1uB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

Since requests using this authentication method result in the transmission of clear-text credentials, the authorization server MUST require the use of a transport-layer security mechanism when sending requests to the token endpoint.

TOC

3.2. Other Client Authentication Methods

The authorization server MAY support any suitable HTTP authentication scheme matching its security requirements. When using other authentication methods, the authorization server MUST define a mapping between the client identifier and the credentials used to authenticate.

TOC

4. Obtaining Authorization

To request an access token, the client obtains authorization from the resource owner. The authorization is expressed in the form of an authorization grant which the client uses to request the access token. OAuth defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials. It also provides an extension mechanism for defining additional grant types.

TOC

4.1. Authorization Code

The authorization code grant type is suitable for clients capable of maintaining their client credentials confidential (for authenticating with the authorization server) such as a client implemented on a secure server. As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

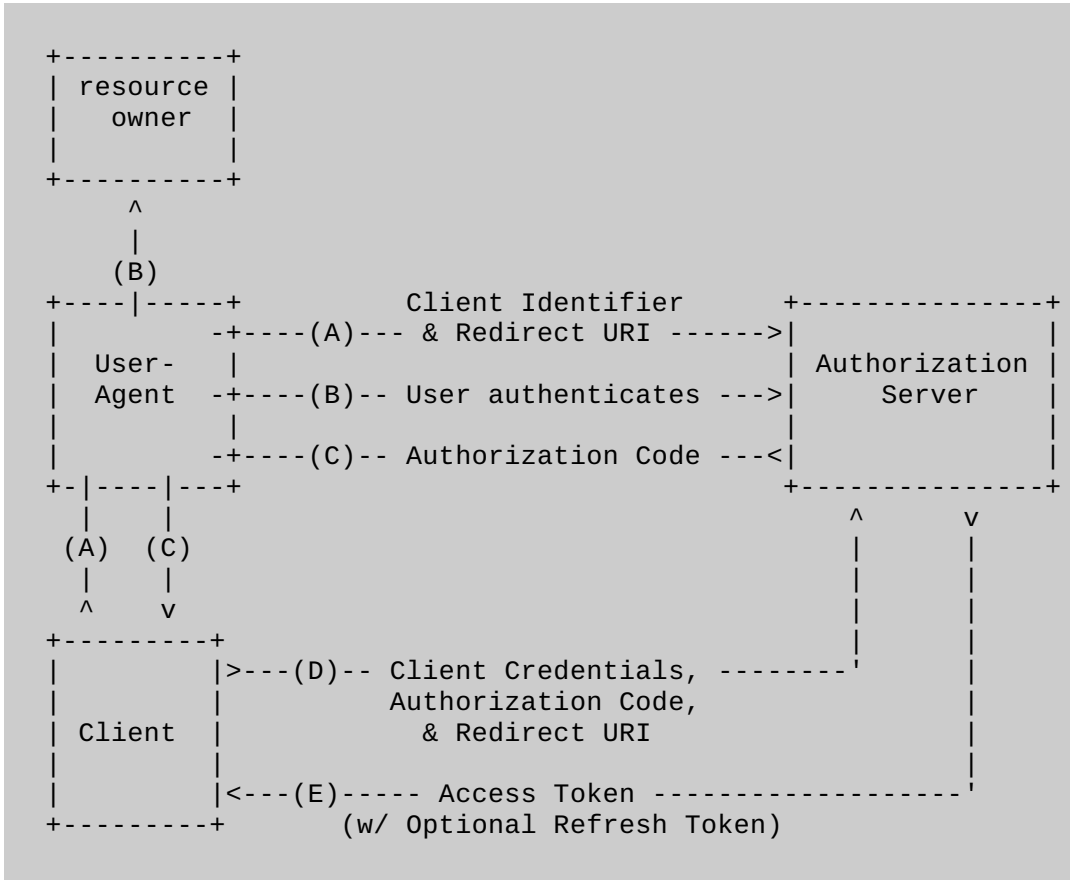


Figure 3: Authorization Code Flow

The flow illustrated in **Figure 3** includes the following steps:

(A)
: The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B)
: The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C)
: Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes an authorization code and any local state provided by the client earlier.

(D)
: The client requests an access token from the authorization server's token endpoint by authenticating using its client credentials, and includes the authorization code received in the previous step. The client includes the redirection URI used to obtain the authorization code for verification.

(E)
: The authorization server validates the client credentials, the authorization code, and ensures the redirection URI received matches the URI used to redirect the client in step (C). If valid, responds back with an access token.

4.1.1. Authorization Request

TOC

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the `application/x-www-form-urlencoded` format as defined by **[W3C.REC-html401-19991224]**:

response_type
: REQUIRED. Value MUST be set to `code`.

client_id
: REQUIRED. The client identifier as described in **Section 3**.

redirect_uri
: REQUIRED, unless a redirection URI has been established between the client and authorization server via other means. Described in **Section 2.1.1**.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited, case sensitive strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

state
: OPTIONAL. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client.

The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent.

For example, the client directs the user-agent to make the following HTTP request using transport-layer security (extra line breaks are for display purposes only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&
    redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

The authorization server validates the request to ensure all required parameters are present and valid. If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

4.1.2. Authorization Response

TOC

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirection URI using the `application/x-www-form-urlencoded` format:

code
: REQUIRED. The authorization code generated by the authorization server. The authorization code SHOULD expire shortly after it is issued to minimize the risk of leaks. The client MUST NOT reuse the authorization code. If an authorization code is used more than once, the authorization server MAY revoke all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

state
: REQUIRED if the `state` parameter was present in the client authorization request. Set to the exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=i1WsRn1uB1
```

The client SHOULD ignore unrecognized response parameters. The authorization code string size is left undefined by this specification. The client should avoid making assumptions about code value sizes. The authorization server should document the size of any value it issues.

TOC

4.1.2.1. Error Response

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier provided is invalid, the authorization server SHOULD inform the resource owner of the error, and MUST NOT redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the query component of the redirection URI using the `application/x-www-form-urlencoded` format:

error
: REQUIRED. A single error code from the following:

  invalid_request
  : The request is missing a required parameter, includes an unsupported parameter or parameter value, or is otherwise malformed.

  unauthorized_client
  : The client is not authorized to request an authorization code using this method.

  access_denied
  : The resource owner or authorization server denied the request.

  unsupported_response_type
  : The authorization server does not support obtaining an authorization code using this method.

  invalid_scope
  : The requested scope is invalid, unknown, or malformed.

  a 4xx or 5xx HTTP status code (except for 400 and 401)
  : The authorization server MAY set the `error` parameter value to a numerical HTTP status code from the 4xx or 5xx range, with the exception of the 400 (Bad Request) and 401 (Unauthorized) status codes. For example, if the service is temporarily unavailable, the authorization server MAY return an error response with `error` set to `503`.

error_description
: OPTIONAL. A human-readable text providing additional information, used to assist in the understanding and resolution of the error occurred. [[add language and encoding information]]

error_uri
: OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the resource owner with additional information about the error.

state
: REQUIRED if a valid `state` parameter was present in the client authorization request. Set to the exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?error=access_denied
```

TOC

4.1.3. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `authorization_code`.

client_id
: REQUIRED. The client identifier as described in **Section 3**.

code
: REQUIRED. The authorization code received from the authorization server.

redirect_uri
: REQUIRED. The redirection URI used by the authorization server to return the authorization response in the previous step.

The client includes its authentication credentials as described in **Section 3**

For example, the client makes the following HTTP using transport-layer security (extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&client_id=s6BhdRkqt3&
code=i1WsRn1uB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST:

- Validate the client credentials and ensure that the authorization code was issued to that client.
- Verify that the authorization code is valid, and that the redirection URI matches the redirection URI used by the authorization server to deliver the authorization code.

TOC

4.1.4. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request client authentication failed or is invalid, the authorization server returns an error response as described in **Section 5.2**.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
```

```
  "access_token":"SlAV32hkKG",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"8xLOxBtZp8",
  "example_parameter":"example_value"
}
```

TOC

4.2. Implicit Grant

The implicit grant type is suitable for clients incapable of maintaining their client credentials confidential (for authenticating with the authorization server) such as client applications residing in a user-agent, typically implemented in a browser using a scripting language such as JavaScript.

As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

Unlike the authorization code grant type in which the client makes separate requests for authorization and access token, the client receives the access token as the result of the authorization request.

Using the implicit grant type does not include client authentication since the client is unable to maintain their credential confidentiality (the client resides on the resource owner's computer or device which makes the client credentials accessible and exploitable). Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on its computer or device.

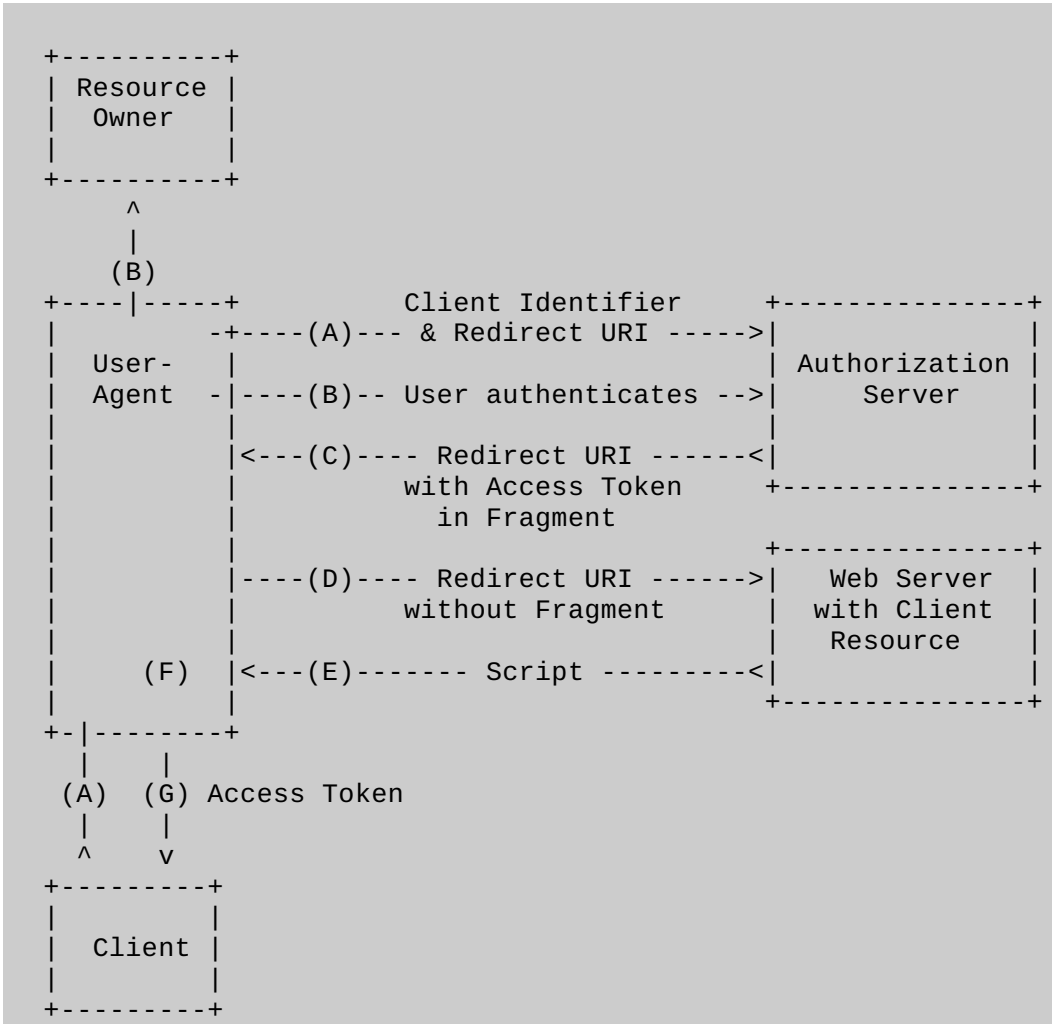


Figure 4: Implicit Grant Flow

The flow illustrated in **Figure 4** includes the following steps:

(A)
: The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B)
: The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C)
: Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes the access token in the URI fragment.

(D)
: The user-agent follows the redirection instructions by making a request to the web server (does not include the fragment). The user-agent retains the fragment information locally.

(E)
: The web server returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI including the fragment retained by the user-agent, and extracting the access token (and other parameters) contained in the fragment.

(F)
: The user-agent executes the script provided by the web server locally, which extracts the access token and passes it to the client.

TOC

4.2.1. Authorization Request

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the `application/x-www-form-urlencoded` format:

response_type
: REQUIRED. Value MUST be set to `token`.

client_id
: REQUIRED. The client identifier as described in **Section 3**.

redirect_uri
: REQUIRED, unless a redirection URI has been established between the client and authorization server via other means. Described in **Section 2.1.1**.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited, case sensitive strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

state
: OPTIONAL. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client.

The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent.

For example, the client directs the user-agent to make the following HTTP request using transport-layer security (extra line breaks are for display purposes only):

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3&
    redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
```

```
Host: server.example.com
```

The authorization server validates the request to ensure all required parameters are present and valid. If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

TOC

4.2.2. Access Token Response

If the resource owner grants the access request, the authorization server issues an access token and delivers it to the client by adding the following parameters to the fragment component of the redirection URI using the `application/x-www-form-urlencoded` format:

access_token
: REQUIRED. The access token issued by the authorization server.

token_type
: REQUIRED. The type of the token issued as described in **Section 7.1**. Value is case insensitive.

expires_in
: OPTIONAL. The duration in seconds of the access token lifetime. For example, the value `3600` denotes that the access token will expire in one hour from the time the response was generated.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited, case sensitive strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope. The authorization server SHOULD include the parameter if the requested scope is different from the one requested by the client.

state
: REQUIRED if the `state` parameter was present in the client authorization request. Set to the exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response (URI extra line breaks are for display purposes only):

```
HTTP/1.1 302 Found
Location: http://example.com/rd#access_token=FJQbwq9&
          token_type=example&expires_in=3600
```

The client SHOULD ignore unrecognized response parameters. The access token string size is left undefined by this specification. The client should avoid making assumptions about value sizes. The authorization server should document the size of any value it issues.

TOC

4.2.2.1. Error Response

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier provided is invalid, the authorization server SHOULD inform the resource owner of the error, and MUST NOT redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the

following parameters to the fragment component of the redirection URI using the `application/x-www-form-urlencoded` format:

error
: REQUIRED. A single error code from the following:

  invalid_request
  : The request is missing a required parameter, includes an unsupported parameter or parameter value, or is otherwise malformed.

  unauthorized_client
  : The client is not authorized to request an access token using this method.

  access_denied
  : The resource owner or authorization server denied the request.

  unsupported_response_type
  : The authorization server does not support obtaining an access token using this method.

  invalid_scope
  : The requested scope is invalid, unknown, or malformed.

  a 4xx or 5xx HTTP status code (except for 400 and 401)
  : The authorization server MAY set the `error` parameter value to a numerical HTTP status code from the 4xx or 5xx range, with the exception of the 400 (Bad Request) and 401 (Unauthorized) status codes. For example, if the service is temporarily unavailable, the authorization server MAY return an error response with `error` set to `503`.

error_description
: OPTIONAL. A human-readable text providing additional information, used to assist in the understanding and resolution of the error occurred. [[add language and encoding information]]

error_uri
: OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the resource owner with additional information about the error.

state
: REQUIRED if a valid `state` parameter was present in the client authorization request. Set to the exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb#error=access_denied
```

TOC

4.3. Resource Owner Password Credentials

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as its computer operating system or a highly privileged application. The authorization server should take special care when enabling the grant type, and only when other flows are not viable.

The grant type is suitable for clients capable of obtaining the resource owner credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials with an access token.

```
```

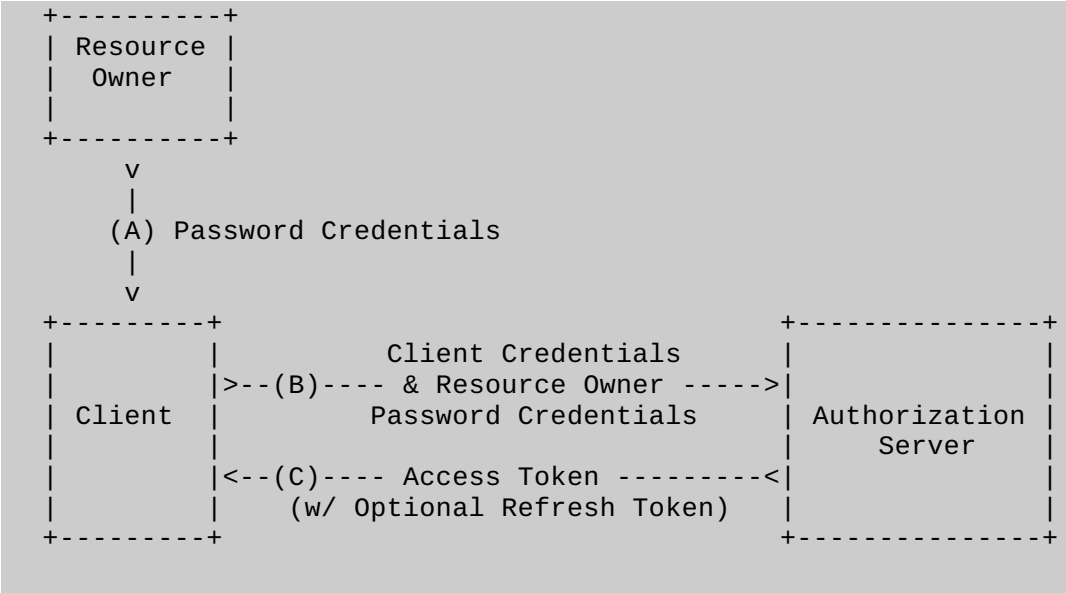


Figure 5: Resource Owner Password Credentials Flow

The flow illustrated in **Figure 5** includes the following steps:

(A)
: The resource owner provides the client with its username and password.

(B)
: The client requests an access token from the authorization server's token endpoint by authenticating using its client credentials, and includes the credentials received from the resource owner.

(C)
: The authorization server validates the resource owner credentials and the client credentials and issues an access token.

TOC

4.3.1. Authorization Request and Response

The method through which the client obtains the resource owner credentials is beyond the scope of this specification. The client MUST discard the credentials once an access token has been obtained.

TOC

4.3.2. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `password`.

client_id
: REQUIRED. The client identifier as described in **Section 3**.

username
: REQUIRED. The resource owner username, encoded as UTF-8.

password
: REQUIRED. The resource owner password, encoded as UTF-8.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited, case sensitive strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

The client includes its authentication credentials as described in **Section 3**

For example, the client makes the following HTTP request using transport-layer security

(extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=password&client_id=s6BhdRkqt3&
username=johndoe&password=A3ddj3w
```

The authorization server MUST:

- Validate the client credentials.
- Validate the resource owner password credentials.

TOC

4.3.3. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token":"SlAV32hkKG",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"8xLOxBtZp8",
  "example_parameter":"example_value"
}
```

TOC

4.4. Client Credentials

The client can request an access token using only its client credentials when the client is requesting access to the protected resources under its control, or those of another resource owner which has been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

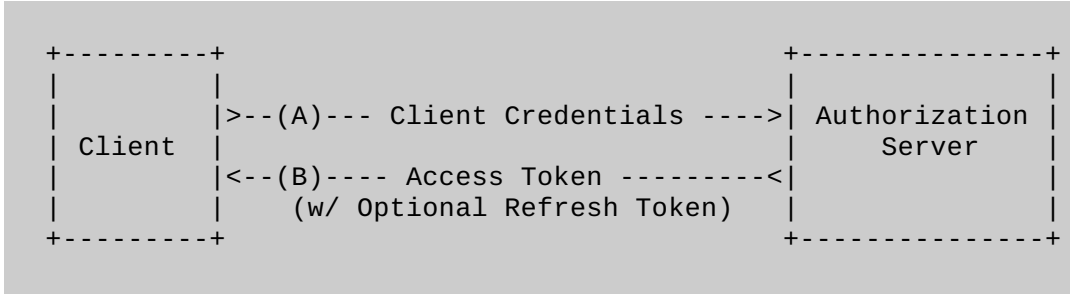


Figure 6: Client Credentials Flow

The flow illustrated in **Figure 6** includes the following steps:

(A)
: The client requests an access token from the token endpoint by authenticating using its client credentials.

(B)
: The authorization server validates the client credentials and issues an access token.

TOC

4.4.1. Authorization Request and Response

Since the client credentials are used as the authorization grant, no additional authorization request is needed as the client is already in the possession of its client credentials.

TOC

4.4.2. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `client_credentials`.

client_id
: REQUIRED. The client identifier as described in **Section 3**.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited, case sensitive strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

The client includes its authentication credentials as described in **Section 3**

For example, the client makes the following HTTP request using transport-layer security (extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=s6BhdRkqt3
```

The authorization server MUST validate the client credentials.

TOC

4.4.3. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token":"SlAV32hkKG",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"8xLOxBtZp8",
  "example_parameter":"example_value"
}
```

TOC

4.5. Extensions

The client uses an extension grant type by specifying the grant type using an absolute URI (defined by the authorization server) as the value of the `grant_type` parameter of the token endpoint, and by adding any additional parameters necessary.

For example, to request an access token using a SAML 2.0 assertion grant type as defined by **[I-D.ietf-oauth-saml2-bearer]**, the client makes the following HTTP request using transport-layer security (line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=http%3A%2F%2Foauth.net%2Fgrant_type%2Fassertion%2F
saml%2F2.0%2Fbearer&assertion=PEFzc2VydGlvbiBJc3N1ZUluc3RhbnQ
[...omitted for brevity...]V0aG5TdGF0ZW1lbnQ-PC9Bc3NlcnRpb24-
```

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

TOC

5. Issuing an Access Token

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

TOC

5.1. Successful Response

The authorization server issues an access token and optional refresh token, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 (OK) status code:

access_token
: REQUIRED. The access token issued by the authorization server.

token_type
: REQUIRED. The type of the token issued as described in **Section 7.1**. Value is case insensitive.

expires_in
: OPTIONAL. The duration in seconds of the access token lifetime. For example, the value `3600` denotes that the access token will expire in one hour from the time the response was generated.

refresh_token
: OPTIONAL. The refresh token which can be used to obtain new access tokens using the same authorization grant as described in **Section 6**.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited, case sensitive strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope. The authorization server SHOULD include the parameter if the requested scope is different from the one requested by the client.

The parameters are included in the entity body of the HTTP response using the `application/json` media type as defined by **[RFC4627]**. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers.

The authorization server MUST include the HTTP `Cache-Control` response header field **[RFC2616]** with a value of `no-store` in any response containing tokens, secrets, or other sensitive information.

For example:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token":"SlAV32hkKG",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"8xLOxBtZp8",
  "example_parameter":"example_value"
}
```

The client SHOULD ignore unrecognized response parameters. The sizes of tokens and other values received from the authorization server are left undefined. The client should avoid making assumptions about value sizes. The authorization server should document the size of any value it issues.

TOC

5.2. Error Response

The authorization server responds with an HTTP 400 (Bad Request) status code and includes the following parameters with the response:

error
: REQUIRED. A single error code from the following:

    invalid_request
    : The request is missing a required parameter, includes an unsupported parameter or parameter value, repeats a parameter, includes multiple credentials, utilizes more than one mechanism for authenticating the client, or is otherwise malformed.

    invalid_client
    : Client authentication failed (e.g. unknown client, no client credentials included, multiple client credentials included, or

unsupported credentials type). The authorization server MAY return an HTTP 401 (Unauthorized) status code to indicate which HTTP authentication schemes are supported. If the client attempted to authenticate via the `Authorization` request header field, the authorization server MUST respond with an HTTP 401 (Unauthorized) status code, and include the `WWW-Authenticate` response header field matching the authentication scheme used by the client.

invalid_grant
: The provided authorization grant is invalid, expired, revoked, does not match the redirection URI used in the authorization request, or was issued to another client.

unauthorized_client
: The authenticated client is not authorized to use this authorization grant type.

unsupported_grant_type
: The authorization grant type is not supported by the authorization server.

invalid_scope
: The requested scope is invalid, unknown, malformed, or exceeds the scope granted by the resource owner.

error_description
: OPTIONAL. A human-readable text providing additional information, used to assist in the understanding and resolution of the error occurred. [[add language and encoding information]]

error_uri
: OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the resource owner with additional information about the error.

The parameters are included in the entity body of the HTTP response using the `application/json` media type as defined by **[RFC4627]**. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{
  "error":"invalid_request"
}
```

If the authorization server encounters an error condition other than the 400 (Bad Request) and 401 (Unauthorized) responses described above (e.g. the service is temporarily unavailable), the authorization server SHOULD include an error response in the entity body, and set the `error` parameter value to the numerical HTTP status code returned.

For example:

```
HTTP/1.1 503 Service Unavailable
Content-Type: application/json

{
  "error":"503"
}
```

6. Refreshing an Access Token

TOC

If the authorization server issued a refresh token to the client, the client makes a refresh request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `refresh_token`.

client_id
: REQUIRED. The client identifier as described in **Section 3**.

refresh_token
: REQUIRED. The refresh token issued to the client.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited, case sensitive strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope. The requested scope MUST be equal or lesser than the scope originally granted by the resource owner, and if omitted is treated as equal to the scope originally granted by the resource owner.

The client includes its authentication credentials as described in **Section 3**.

For example, the client makes the following HTTP request using transport-layer security (extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&client_id=s6BhdRkqt3&
refresh_token=n4E9O119d
```

The authorization server MUST validate the client credentials, ensure that the refresh token was issued to the authenticated client, validate the refresh token, and verify that the resource owner's authorization is still valid. If valid and authorized, the authorization server issues an access token as described in **Section 5.1**. If the request failed verification or is invalid, the authorization server returns an error response as described in **Section 5.2**.

The authorization server MAY issue a new refresh token, in which case, the client MUST discard the old refresh token and replace it with the new refresh token.

7. Accessing Protected Resources

TOC

The client accesses protected resources by presenting the access token to the resource server. The resource server MUST validate the access token and ensure it has not expired and that its scope covers the requested resource. The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification, but generally involve an interaction or coordination between the resource server and the authorization server.

The method in which the client utilized the access token to authenticate with the resource server depends on the type of access token issued by the authorization server. Typically, it involves using the HTTP `Authorization` request header field **[RFC2617]** with an authentication scheme defined by the access token type specification.

7.1. Access Token Types

TOC

The access token type provides the client with the information required to successfully utilize the access token to make a protected resource request (along with type-specific attributes). The client MUST NOT use an access token if it does not understand the token type.

For example, the `bearer` token type defined in **[I-D.ietf-oauth-v2-bearer]** is utilized by simply including the access token string in the request:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer 7Fjfp0ZBr1KtDRbnfVdmIw
```

while the `mac` token type defined in **[I-D.ietf-oauth-v2-http-mac]** is utilized by issuing a MAC key together with the access token which is used to sign certain components of the HTTP requests:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: MAC id="h480djs93hd8",
                   nonce="274312:dj83hs9s",
                   mac="kDZvddkndxvhGRXZhvuDjEWhGeE="
```

The above examples are provided for illustration purposes only. Developers are advised to consult the **[I-D.ietf-oauth-v2-bearer]** and **[I-D.ietf-oauth-v2-http-mac]** specifications before use.

Each access token type definition specifies the additional attributes (if any) sent to the client together with the `access_token` response parameter. It also defines the HTTP authentication method used to include the access token when making a protected resource request.

---

8. Extensibility

TOC

8.1. Defining Access Token Types

TOC

Access token types can be defined in one of two ways: registered in the access token type registry (following the procedures in **Section 10.1**), or use a unique absolute URI as its name.

Types utilizing a URI name SHOULD be limited to vendor-specific implementations that are not commonly applicable, and are specific to the implementation details of the resource server where they are used.

All other types MUST be registered. Type names MUST conform to the type-name ABNF. If the type definition includes a new HTTP authentication scheme, the type name SHOULD be identical to the HTTP authentication scheme name (as defined by **[RFC2617]**).

```
type-name  = 1*name-char
name-char   = "-" / "." / "_" / DIGIT / ALPHA
```

8.2. Defining New Endpoint Parameters

TOC

New request or response parameters for use with the authorization endpoint or the token endpoint are defined and registered in the parameters registry following the procedure in **Section 10.2**.

Parameter names MUST conform to the param-name ABNF and parameter values syntax MUST be well-defined (e.g., using ABNF, or a reference to the syntax of an existing parameter).

```
param-name  = 1*name-char
name-char   = "-" / "." / "_" / DIGIT / ALPHA
```

Unregistered vendor-specific parameter extensions that are not commonly applicable, and are specific to the implementation details of the authorization server where they are used SHOULD utilize a vendor-specific prefix that is not likely to conflict with other registered values (e.g. begin with 'companyname_').

8.3. Defining New Authorization Grant Types

TOC

New authorization grant types can be defined by assigning them a unique absolute URI for use with the `grant_type` parameter. If the extension grant type requires additional token endpoint parameters, they MUST be registered in the OAuth parameters registry as described by **Section 10.2**.

8.4. Defining Additional Error Codes

TOC

In cases where protocol extensions (i.e. access token types, extension parameters, or extension grant types) require additional error codes to be used with the authorization code grant error response (**Section 4.1.2.1**), the implicit grant error response (**Section 4.2.2.1**), or the token error response (**Section 5.2**), such error codes MAY be defined.

Extension error codes MUST be registered (following the procedures in **Section 10.3**) if the extension they are used in conjunction with is a registered access token type, a registered endpoint parameter, or an extension grant type. Error codes used with unregistered extensions MAY be registered.

Error codes MUST conform to the error-code ABNF, and SHOULD be prefixed by an identifying name when possible. For example, an error identifying an invalid value set to the extension parameter `example` should be named `example_invalid`.

```
error-code  = ALPHA *error-char
error-char  = "-" / "." / "_" / DIGIT / ALPHA
```

9. Security Considerations

TOC

As a flexible and extensible framework, OAuth's security considerations depend on many factors. The following sections provide implementers with security guidelines focused on

three common client types:

Web Application
: A web application is a client running on a web server. End-users access the client via an HTML user interface rendered in a user-agent on the end-user's device. The client credentials as well as any access token issued to the client are stored on the web server and are not exposed to or accessible by the end-user.

User-Agent-based Application
: A user-agent-based application is a client in which the client code is downloaded from a web server and executes within a user-agent on the end-user's device. The OAuth protocol data and credentials are accessible to the end-user. Since such applications directly reside within the user-agent, they can make seamless use of the user-agent capabilities in the end-user authorization process.

Native Application
: A native application is a client which is installed and executes on the end-user's device. The OAuth protocol data and credentials are accessible to the end-user. It is assumed that such an application can protect dynamically issued credentials, such as refresh tokens, from eavesdropping by other applications residing on the same device.

A comprehensive OAuth security model and analysis, as well as background for the protocol design is provided in **[I-D.lodderstedt-oauth-security]**.

TOC

9.1. Client Authentication

The authorization server issues client credentials to web applications for the purpose of authenticating them. The authorization server is encouraged to consider using stronger client authentication means than a client password. Application developers MUST ensure confidentiality of client passwords and other credentials.

The authorization server MUST NOT issue client passwords or other credentials to native or user-agent-based applications for the purpose of client authentication. The authorization server MAY issue a client password or other credentials for a specific installation of a native application on a specific device.

TOC

9.2. Client Impersonation

Given the inability of some clients to keep their client credentials confidential, a malicious client can impersonate another client and obtain access to protected resources. The authorization server MUST authenticate the client whenever possible. If the authorization server cannot authenticate the a client due to the client's limitations, the authorization server should utilize other means to protect resource owners from such malicious clients, including but not limited to engaging the end-user to assist in identifying the client and its source.

The authorization server SHOULD enforce explicit end-user authentication, or prompt the end-user to authorize access again, providing the end-user with information about the client, scope, and duration of the authorization. It is up to the end-user to review the information in the context of the current client, and authorize the request.

The authorization server SHOULD NOT automatically, without active end-user interaction, process repeated authorization requests without authenticating the client or relying on other measures to ensure the repeated request comes from a valid client and not an impersonator.

The authorization server SHOULD require the client to pre-register its redirection URI and validate the value of the `redirect_uri` against the pre-registered value. The client MUST NOT serve an open redirector resource which can be used by an attacker to construct an URI that will pass the authorization server's redirection URI matching rules, and will redirect the end-user's user-agent to the attacker's server.

The authorization server SHOULD issue access tokens with limited scope and duration to clients incapable of authenticating.

9.3. Access Token Credentials

TOC

Access token credentials MUST be kept confidential in transit and storage, and shared only among the authorization server, the resource servers the credentials are valid for, and the client to whom the credentials were issued. Application developers MUST NOT store access token credentials in non-transient memory.

When using the implicit grant type, the access token credentials are transmitted in the URI fragment, which can expose the credentials to unauthorized parties.

The authorization server MUST ensure that access token credentials cannot be generated, modified, or guessed to produce valid access token credentials.

The client SHOULD request access token credentials with the minimal scope and duration necessary. The authorization server SHOULD take the client identity into account when choosing to honor the requested scope, and MAY issue credentials with a lesser scope than requested.

9.4. Refresh Tokens

TOC

Authorization servers MAY issue refresh tokens to web and native applications.

Refresh tokens MUST be kept confidential in transit and storage, and shared only among the authorization server and the client to whom the refresh tokens were issued. The authorization server MUST maintain the link between a refresh token and the client to whom it was issued.

The authorization server MUST verify the link between the refresh token and client identity whenever the client's identity can be authenticated. When client authentication is not possible, the authorization server SHOULD deploy other means to detect refresh token abuse.

The authorization server MUST ensure that refresh tokens cannot be generated, modified, or guessed to produce valid refresh tokens.

9.5. Request Confidentiality

TOC

Access token credentials, refresh tokens, resource owner passwords, and client secrets MUST NOT be transmitted in the clear. Authorization codes SHOULD NOT be transmitted in the clear.

9.6. Endpoints Authenticity

TOC

In order to prevent man-in-the-middle and phishing attacks, the authorization server MUST implement and require TLS with server-side authentication in all exchanges. The client MUST verify the authorization server TLS certificate, as well as the certificate chain.

9.7. Credentials Guessing Attacks

TOC

The authorization server MUST prevent attackers from guessing access tokens, authorization codes, refresh tokens, resource owner passwords, and client secrets.

When generating tokens and other secrets not intended for direct human utilization, the authorization server MUST use a reasonable level of entropy in order to mitigate the risk of

guessing attacks. When creating secrets intended for human usage, the authorization server MUST utilize other means to protect those secrets.

9.8. Phishing Attacks

TOC

Native applications SHOULD use external browsers instead of embedding browsers within the application when requesting end-user authorization. External browsers offer a familiar user experience and a trusted environment in which end-users can confirm the authenticity of the authorization server.

To reduce the risk of phishing attacks, the authorization servers MUST utilize TLS to allow user-agents to validate the authorization server's identity. Service providers should educate their end-users about the risks of phishing attacks and how they can verify the authorization server's identity.

9.9. Authorization Codes

TOC

The transmission of authorization codes SHOULD be made over a secure channel, and the client SHOULD implement TLS for use with its redirection URI. Authorization codes MUST be kept confidential. Since authorization codes are transmitted via user-agent redirections, they could potentially be disclosed through user-agent history and HTTP referrer headers.

Authorization codes SHOULD be short lived and MUST be single use. If the authorization server observes multiple attempts to exchange an authorization code for an access token, the authorization server SHOULD revoke all access tokens already granted based on the compromised authorization code.

If the client can be authenticated, the authorization servers MUST authenticate the client and ensure that the authorization code was issued to the same client.

Authorization codes operate as plaintext bearer credentials, used to verify that the end-user who granted authorization at the authorization server, is the same end-user returning to the client to complete the process. Therefore, if the client relies on the authorization code for its own end-user authentication, the client redirection endpoint MUST require TLS.

9.10. Session Fixation

TOC

Session fixation attacks leverage the authorization code grant type, by tricking an end-user to authorize access to a legitimate client, but to a client account under the control of the attacker. The only difference between a valid flow and the attack flow is in how the victim reached the authorization server to grant access. Once at the authorization server, the victim is prompted with a normal, valid request on behalf of a legitimate and familiar client. The attacker then uses the victim's authorization to gain access to the information authorized by the victim.

In order to prevent such an attack, authorization servers MUST ensure that the redirection URI used to obtain the authorization code, is the same as the redirection URI provided when exchanging the authorization code for an access token. The authorization server SHOULD require the client to pre-register their redirection URI and if provided, MUST validate the redirection URI received in the authorization request against the pre-registered value.

9.11. Redirection URI Validation

TOC

[[Add specific recommendations about redirection validation and matching]]

TOC

9.12. Resource Owner Password Credentials

The resource owner password credentials grant type is often used for legacy or migration reasons. It reduces the overall risk of storing username and password in the client, but does not eliminate the need to expose highly privileged credentials to the client.

This grant type carries a higher risk than the other grant types because it maintains the password anti-pattern OAuth seeks to avoid. The client could abuse the password or the password could unintentionally be disclosed to an attacker (e.g. via log files or other records kept by the client).

Additionally, because the resource owner does not have control over the authorization process (the resource owner involvement ends when it hands over its credentials to the client), the client can obtain access tokens with a broader scope and longer duration than desired by the resource owner. The authorization server SHOULD restrict the scope and duration of access tokens issued via this grant type.

The authorization server and client SHOULD minimize use of this grant type and utilize other grant types whenever possible.

---

10. IANA Considerations

10.1. The OAuth Access Token Type Registry

This specification establishes the OAuth access token type registry.

Access token types are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from **[RFC5226]**). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access toke type: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: oauth-ext-review.]]

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert can be first appealed to Application Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

10.1.1. Registration Template

Type name:
: The name requested (e.g., "example").

Additional Token Endpoint Response Parameters:
: Additional response parameters returned together with the `access_token` parameter. New parameters MUST be separately registered in the OAuth parameters registry as described by **Section 10.2**.

HTTP Authentication Scheme(s):

The HTTP authentication scheme name(s), if any, used to authenticate protected resources requests using access token of this type.

Change controller:

For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

Specification document(s):

Reference to document that specifies the parameter, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

TOC

10.2. The OAuth Parameters Registry

This specification establishes the OAuth parameters registry.

Additional parameters for inclusion in the authorization endpoint request, the authorization endpoint response, the token endpoint request, or the token endpoint response, are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from **[RFC5226]**). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for parameter: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: oauth-ext-review.]]

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert can be first appealed to Application Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

TOC

10.2.1. Registration Template

Parameter name:

The name requested (e.g., "example").

Parameter usage location:

The location(s) where parameter can be used. The possible locations are: authorization request, authorization response, token request, or token response.

Change controller:

For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

Specification document(s):

Reference to document that specifies the parameter, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

TOC

10.2.2. Initial Registry Contents

The OAuth Parameters Registry's initial contents are:

- Parameter name: client_id
- Parameter usage location: authorization request, token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: client_secret
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: response_type
- Parameter usage location: authorization request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: redirect_uri
- Parameter usage location: authorization request, token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: scope
- Parameter usage location: authorization request, authorization response, token request, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: state
- Parameter usage location: authorization request, authorization response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: code
- Parameter usage location: authorization response, token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: error_description
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: error_uri
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: grant_type
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: access_token
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: token_type
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: expires_in
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: username
- Parameter usage location: token request
- Change controller: IETF

- Specification document(s): [[this document]]

- Parameter name: password
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: refresh_token
- Parameter usage location: token request, token response
- Change controller: IETF
- Specification document(s): [[this document]]

TOC

10.3. The OAuth Extensions Error Registry

This specification establishes the OAuth extensions error registry.

Additional error codes used together with other protocol extensions (i.e. extension grant types, access token types, or extension parameters) are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from **[RFC5226]**). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for error code: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: oauth-ext-review.]]

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert can be first appealed to Application Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

TOC

10.3.1. Registration Template

Error name:
: The name requested (e.g., "example").

Error usage location:
: [[Pending Consensus]] The location(s) where the error can be used. The possible locations are: authorization code grant error response (**Section 4.1.2.1**), implicit grant error response (**Section 4.2.2.1**), or token error response (**Section 5.2**).

Related protocol extension:
: The name of the extension grant type, access token type, or extension parameter, the error code is used in conjunction with.

Change controller:
: For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

Specification document(s):
: Reference to document that specifies the error code, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

---

11. Acknowledgements

TOC

The initial OAuth 2.0 protocol specification was edited by David Recordon, based on two previous publications: the OAuth 1.0 community specification **[RFC5849]**, and OAuth WRAP (OAuth Web Resource Authorization Profiles) **[I-D.draft-hardt-oauth-01]**. The Security Considerations section was drafted by Torsten Lodderstedt, Mark McGloin, Phil Hunt, and Anthony Nadalin.

The OAuth 1.0 community specification was edited by Eran Hammer-Lahav and authored by Mark Atwood, Dirk Balfanz, Darren Bounds, Richard M. Conlan, Blaine Cook, Leah Culver, Breno de Medeiros, Brian Eaton, Kellan Elliott-McCrea, Larry Halff, Eran Hammer-Lahav, Ben Laurie, Chris Messina, John Panzer, Sam Quigley, David Recordon, Eran Sandler, Jonathan Sergent, Todd Sieling, Brian Slesinsky, and Andy Smith.

The OAuth WRAP specification was edited by Dick Hardt and authored by Brian Eaton, Yaron Goland, Dick Hardt, and Allen Tom.

This specification is the work of the OAuth Working Group which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording which shaped and formed the final specification:

Michael Adams, Andrew Arnott, Dirk Balfanz, Scott Cantor, Blaine Cook, Brian Campbell, Leah Culver, Bill de hÓra, Brian Eaton, Brian Ellin, Igor Faynberg, George Fletcher, Tim Freeman, Evan Gilbert, Yaron Goland, Brent Goldman, Kristoffer Gronowski, Justin Hart, Craig Heath, Phil Hunt, Michael B. Jones, John Kemp, Mark Kent, Raffi Krikorian, Chasen Le Hara, Rasmus Lerdorf, Torsten Lodderstedt, Hui-Lan Lu, Paul Madsen, Alastair Mair, Eve Maler, James Manger, Mark McGloin, Laurence Miao, Chuck Mortimore, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Naitik Shah, Luke Shepard, Vlad Skvortsov, Justin Smith, Jeremy Suriel, Christian Stübner, Paul Tarjan, Allen Tom, Franklin Tse, Nick Walker, Skylar Woodward.

Appendix A. Editor's Notes

TOC

While many people contributed to this specification throughout its long journey, the editor would like to acknowledge and thank a few individuals for their outstanding and invaluable efforts leading up to the publication of this specification. It is these individuals without whom this work would not have existed, or reached its successful conclusion.

David Recordon for continuously being one of OAuth's most valuable assets, bringing pragmatism and urgency to the work, and helping shape it from its very beginning, as well as being one of the best collaborators I had the pleasure of working with.

Mark Nottingham for introducing OAuth to the IETF and setting the community on this course. Lisa Dusseault for her support and guidance as the Application area director. Blaine Cook, Peter Saint-Andre, and Hannes Tschofenig for their work as working group chairs.

James Manger for his creative ideas and always insightful feedback. Brian Campbell, Torsten Lodderstedt, Chuck Mortimore, Justin Richer, Marius Scurtescu, and Luke Shepard for their continued participation and valuable feedback.

Special thanks goes to Mike Curtis and Yahoo! for their unconditional support of this work for over three years.

---

12. References

TOC

12.1. Normative References

TOC

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels," BCP 14, RFC 2119, March 1997 (TXT, HTML, XML).

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," RFC 2616, June 1999 (TXT, PS, PDF, HTML, XML).

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," RFC 2617, June 1999 (TXT, HTML, XML).

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," STD 66, RFC 3986, January 2005 (TXT, HTML, XML).

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627, July 2006 (TXT).

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs," BCP 26, RFC 5226, May 2008 (TXT).

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," STD 68, RFC 5234, January 2008 (TXT).

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, August 2008 (TXT).

[W3C.REC-html401-19991224] Hors, A., Jacobs, I., and D. Raggett, "HTML 4.01 Specification," World Wide Web Consortium Recommendation REC-html401-19991224, December 1999 (HTML).

12.2. Informative References

TOC

[I-D.draft-hardt-oauth-01] Hardt, D., Ed., Tom, A., Eaton, B., and Y. Goland, "OAuth Web Resource Authorization Profiles," January 2010.

[I-D.ietf-oauth-saml2-bearer] Campbell, B. and C. Mortimore, "SAML 2.0 Bearer Assertion Grant Type Profile for OAuth 2.0," draft-ietf-oauth-saml2-bearer-03 (work in progress), February 2011 (TXT).

[I-D.ietf-oauth-v2-bearer] Jones, M., Hardt, D., and D. Recordon, "The OAuth 2.0 Protocol: Bearer Tokens," draft-ietf-oauth-v2-bearer-04 (work in progress), March 2011 (TXT).

[I-D.ietf-oauth-v2-http-mac] Hammer-Lahav, E., Barth, A., and B. Adida, "HTTP Authentication: MAC Access Authentication," draft-ietf-oauth-v2-http-mac-00 (work in progress), May 2011 (TXT, PDF).

[I-D.lodderstedt-oauth-security] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," draft-lodderstedt-oauth-security-01 (work in progress), March 2011 (TXT).

[OASIS.saml-core-2.0-os] Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0," OASIS Standard saml-core-2.0-os, March 2005.

[RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol," RFC 5849, April 2010 (TXT).

Authors' Addresses

TOC

Eran Hammer-Lahav (editor)
Yahoo!
Email: eran@hueniverse.com
URI: http://hueniverse.com

David Recordon
Facebook
Email: dr@fb.com
URI: http://www.davidrecordon.com/

Dick Hardt
Microsoft
Email: dick.hardt@gmail.com
URI: http://dickhardt.org/