The OAuth 2.0 Authorization Protocol
draft-ietf-oauth-v2-18

## Abstract

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

## Status of this Memo

## Copyright Notice

## Table of Contents

## [1.](#) Introduction

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.

   *Compromise of any third-party application results in compromise
    of the end-user's password and all of the data protected by that
    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, lifetime, 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, an 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.

## 1.1. Roles

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

**resource owner**
             An entity capable of granting access to a protected
   resource (e.g. 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.

## 1.2. Protocol Flow

```
  +--------+                               +---------------+
  |        |--(A)- Authorization Request ->|   Resource    |
  |        |                               |     Owner     |
  |        |<-(B)-- Authorization Grant ---|               |
  |        |                               +---------------+
  |        |
  |        |                               +---------------+
  |        |--(C)-- Authorization Grant -->| Authorization |
  | Client |                               |     Server    |
  |        |<-(D)----- Access Token -------|               |
  |        |                               +---------------+
  |        |
  |        |                               +---------------+
  |        |--(E)----- Access Token ------>|   Resource    |
  |        |                               |     Server    |
  |        |<-(F)--- Protected Resource ---|               |
  +--------+                               +---------------+
```
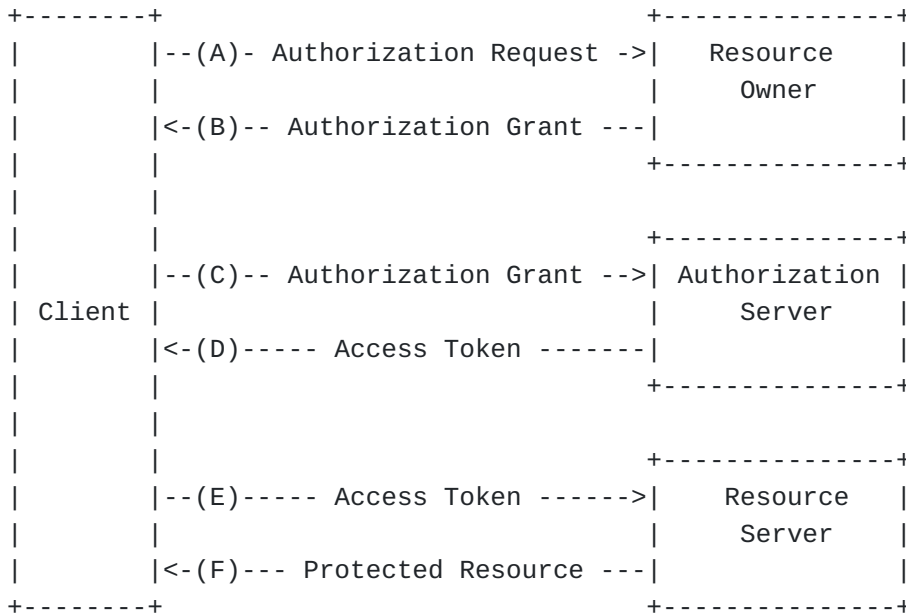
The abstract flow illustrated in Figure 1 describes the interaction
between the four roles and includes the following steps:

**(%C)**  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.

**(%C)**  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 and presenting the authorization grant.

**(%C)**  The authorization server authenticates the client and validates
   the authorization grant, and if valid issues an access token.

**(%C)**    The client requests the protected resource from the resource
    server and authenticates by presenting the access token.

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

### 1.3. Access Token

Access tokens are credentials used to access protected resources. 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, which are beyond the
scope of this specification, 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.

### 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.

### 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.

### 1.4.2. Implicit

The authorization grant is implicit when an access token is issued to the client directly as the result of the resource owner authorization, without using intermediate credentials (such as an authorization code). When issuing an implicit grant, the authorization server does not authenticate the client and the client identity is verified via the redirection URI used to deliver the access token to the client. 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. However, this convenience should be weighted against the security implications of using implicit grants, especially when the authorization code grant type is available.

### 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 device 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.

### 1.4.4. Client Credentials

The client credentials (or other forms of client authentication) 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

Additional grant types may be defined to provide a bridge between OAuth
and other protocols.

### 1.5. Refresh Token

Refresh tokens are credentials used to obtain access tokens. Refresh
tokens are issued to the client by the authorization server and are
used to obtain a new access token when the current access token becomes
invalid or expires, or to obtain additional access tokens with
identical or narrower scope (access tokens may have a shorter lifetime
and fewer permissions than authorized by the resource owner). Issuing a
refresh token is optional and is included when issuing an 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 denotes an identifier used to retrieve the
authorization information. Unlike access tokens, refresh tokens are
intended for use only with authorization servers and are never sent to
resource servers.

```
  +--------+                                          +---------------+
  |        |--(A)------- Authorization Grant --------->|               |
  |        |                                           |               |
  |        |<-(B)----------- Access Token -------------|               |
  |        |               & Refresh Token             |               |
  |        |                                           |               |
  |        |                            +----------+   |               |
  |        |--(C)---- Access Token ---->|          |   |               |
  |        |                            |          |   |               |
  |        |<-(D)- Protected Resource --| Resource |   | Authorization |
  | Client |                            |  Server  |   |     Server    |
  |        |--(E)---- Access Token ---->|          |   |               |
  |        |                            |          |   |               |
  |        |<-(F)- Invalid Token Error -|          |   |               |
  |        |                            +----------+   |               |
  |        |                                           |               |
  |        |--(G)----------- Refresh Token ----------->|               |
  |        |                                           |               |
  |        |<-(H)----------- Access Token -------------|               |
  +--------+           & Optional Refresh Token        +---------------+
```
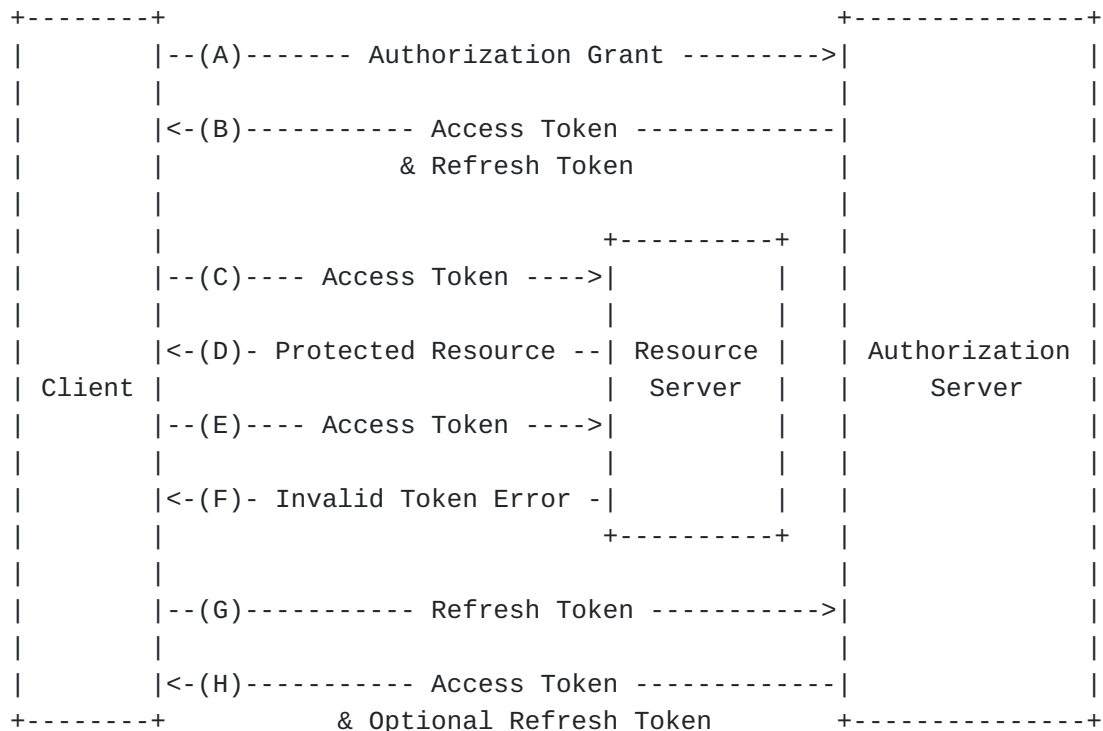
The flow illustrated in [Figure 2](#) includes the following steps:

(%C)  The client requests an access token by authenticating with the
      authorization server, and presenting an authorization grant.

(%C)  The authorization server authenticates the client and validates
      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.

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

(%C)  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.

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

(%C)  The client requests a new access token by authenticating with the
      authorization server and presenting the refresh token.

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

## 1.6. 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].
Certain security-related terms are to be understood in the sense
defined in [RFC4949]. These terms include, but are not limited to,
'attack', 'authentication', 'authorization', 'certificate',
'confidentiality', 'credential', 'encryption', 'identity', 'sign',
'signature', 'trust', 'validate', and 'verify'.
Unless otherwise noted, all the protocol parameter names and values are
case sensitive.

## 2. Client Registration

[[ Pending Consensus ]]
Before initiating the protocol, the client registers with the
authorization server. The means through which the client registers with
the authorization server are beyond the scope of this specification,

but typically involve end-user interaction with an HTML registration form.
Client registration does not require a direct interaction between the client and the authorization server. When supported by the authorization server, registration can rely on other means for establishing trust and obtaining the required client properties (e.g. redirection URI, client type). For example, registration can be accomplished using a self-issued or third-party-issued assertion, or by the authorization server performing client discovery using a trusted channel.

## 2.1. Client Types

OAuth defines two client types, based on their ability to authenticate securely with the authorization server (i.e. ability to maintain the confidentiality of their client credentials):

**private**

Clients capable of maintaining the confidentiality of their credentials (e.g. client implemented on a secure server with restricted access to the client credentials), or capable of secure client authentication using other means.

**public**

Clients incapable of maintaining the confidentiality of their credentials (e.g. clients executing on the resource owner's device such as an installed native application or a user-agent-based application), and incapable of secure client authentication via any other mean.

The client type designation is based on the authorization server's definition of secure authentication and its acceptable exposure levels of client credentials.

## 2.2. Registration Requirements

When registering a client, the client developer MUST specify:

*the client type as described in Section 2.1,

*the client redirection URIs as described in Section 3.1.2, and

*any other information required by the authorization server (e.g. application name, website, description, logo image, the acceptance of legal terms).

## 2.3. Client Identifier

The authorization server issues the registered client a client identifier - a unique string representing the registration information

provided by the client. The client identifier is not a secret, it is
exposed to the resource owner, and cannot not be used alone for client
authentication.

## 2.4. Client Authentication

In addition, the client and authorization server establish a client
authentication method suitable for the client type and security
requirements of the authorization server. The authorization server MAY
accept any form of client authentication meeting its security
requirements.
Private clients are typically issued (or establish) a set of client
credentials used for authenticating with the authorization server (e.g.
password, public/private key pair).
The authorization server SHOULD NOT make assumptions about the client
type or accept the type information provided without establishing trust
with the client or its developer. The authorization server MUST NOT
rely on client authentication performed by public clients.
The client MUST NOT use more than one authentication method in each
request.

### 2.4.1. Client Password

Clients in possession of a client password MAY use the HTTP Basic
authentication scheme as defined in [RFC2617] to authenticate with the
authorization server. The client identifier is used as the username,
and the client password is used as the password.
For example (extra line breaks are for display purposes only):


    Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW



Alternatively, the authorization server MAY allow including the client
credentials in the request body using the following parameters:

**client_id**
        REQUIRED. The client identifier issued to the client during
   the registration process described by Section 2.3.

**client_secret**
            REQUIRED. The client secret.

Including the client credentials in the request body using the two
parameters is NOT RECOMMENDED, and should be limited to clients unable
to directly utilize the HTTP Basic authentication scheme (or other
password-based HTTP authentication schemes).
For example, requesting to refresh an access token (Section 6) using
the body parameters (extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded;charset=UTF-8

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
&client_id=s6BhdRkqt3&client_secret=7Fjfp0ZBr1KtDRbnfVdmIw
```

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

### 2.4.2. Other 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 (registration record) and authentication
scheme.

### 2.5. Unregistered Clients

This specification does not exclude the use of unregistered clients.
However, the use with such clients is beyond the scope of this
specification, and requires additional security analysis and review of
its interoperability impact.

### 3. 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.

### 3.1. Authorization Endpoint

The authorization endpoint is used to interact with the resource owner
and obtain authorization which is expressed explicitly as an
authorization code (later 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 the location 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. The endpoint URI MUST NOT include a fragment component.

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.

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 be included more than once.

### 3.1.1. Response Type

The authorization endpoint is used by the authorization code grant type and implicit grant type flows. The client informs the authorization server of the desired grant type using the following parameter:

**response_type**

> REQUIRED. The value MUST be one of code for requesting an authorization code as described by Section 4.1.1, token for requesting an access token (implicit grant) as described by Section 4.2.1, or a registered extension value as described by Section 8.4.

If an authorization request is missing the response_type parameter, the authorization server SHOULD return an error response as described in Section 4.1.2.1.

### 3.1.2. Redirection URI

[[ Pending Consensus ]]
After completing its interaction with the resource owner, the authorization server directs the resource owner's user-agent back to the client. The authorization server redirects the user-agent to the client's redirection URI previously established with the authorization server during the client registration process.

The redirection URI MUST be an absolute URI as defined by [RFC3986] section 4.3, MAY include a query component which MUST be retained by the authorization server when adding additional query parameters, and MUST NOT include a fragment component.

### 3.1.2.1. Endpoint Confidentiality

If a redirection request will result in the transmission of an authorization code or access token over an open network (between the resource owner's user-agent and the client), the client SHOULD require the use of a transport-layer security mechanism.
Lack of transport-layer security can have a severe impact on the security of the client and the protected resources it is authorized to access. The use of transport-layer security is particularly critical when the authorization process is used as a form of delegated end-user authentication by the client (e.g. third-party sign-in service).

### 3.1.2.2. Registration Requirements

The authorization server MUST require public clients to register their redirection URI, MUST require all clients to register their redirection URI prior to utilizing the implicit grant type, and SHOULD require all clients to register their redirection URI prior to utilizing the authorization code grant type.
The authorization server SHOULD require the client to provide the complete redirection URI (the client MAY use the state request parameter to achieve per-request customization). The authorization server MAY allow the client to register multiple redirection URIs. If requiring the registration of the complete redirection URI is not possible, the authorization server SHOULD require the registration of the URI scheme, authority, and path.

### 3.1.2.3. Dynamic Configuration

If multiple redirection URIs have been registered, if only part of the redirection URI has been registered, or if no redirection URI has been registered, the client MUST include a redirection URI with the authorization request using the redirect_uri request parameter.
When a redirection URI is included in an authorization request, the authorization server MUST compare and match the value received against at least one of the registered redirection URIs (or URI components) as defined in [RFC3986] section 6, if any redirection URIs were registered.
If the authorization server allows the client to dynamically change the query component of the redirection URI, the client MUST ensure that manipulation of the query component by an attacker cannot lead to an abuse of the redirection endpoint as an open redirector.

### [3.1.2.4.](#) Invalid Endpoint

If an authorization request fails validation due to a missing, invalid, or mismatching redirection URI, the authorization server SHOULD inform the resource owner of the error, and MUST NOT automatically redirect the user-agent to the invalid redirection URI.
The authorization server SHOULD NOT redirect the user-agent to unregistered or untrusted URIs to prevent the authorization endpoint from being used as an open redirector.

### [3.1.2.5.](#) Endpoint Content

The redirection request to the client's endpoint typically results in an HTML document response, processed by the user-agent. If the HTML response is served directly as the result of the redirection request, any script included in the HTML document will execute with full access to the redirection URI and the credentials it contains.
The client SHOULD NOT include any third-party scripts in the redirection endpoint response. Instead, it should extract the credentials from the URI and redirect the user-agent again to another endpoint without the credentials in the URI.
The client MUST NOT include any untrusted third-party scripts in the redirection endpoint response (e.g. third-party analytics, social plug-ins, ad networks) without first ensuring that its own scripts used to extract and remove the credentials from the URI will execute first.

### [3.2.](#) Token Endpoint

The token endpoint is used by the client to obtain an access token by 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 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 be
included more than once.

### 3.2.1. Client Authentication

[[ Pending Consensus ]]
Private clients, clients issued client credentials, or clients assigned
other authentication requirements, MUST authenticate with the
authorization server as described in Section 2.4 when making requests
to the token endpoint. Client authentication is used for:

   *Enforcing the binding of refresh tokens and authorization codes
    to the client they are issued. Client authentication is critical
    when an authorization code is transmitted to the redirection URI
    endpoint over an insecure channel, or when the redirection URI
    has not been registered in full.

   *Recovery from a compromised client by disabling the client or
    changing its credentials, by preventing an attacker from abusing
    stolen refresh tokens. Changing a single set of client
    credentials is significantly faster than revoking an entire set
    of refresh tokens.

   *Implementing authentication management best practices which
    require periodic credentials rotation. Rotation of an entire set
    of refresh tokens can be challenging, while rotation of a single
    set of client credentials is significantly easier. In addition,
    this specification does not provide a mechanism for refresh token
    rotation.

The security ramifications of allowing unauthenticated access by public
clients to the token endpoint MUST be considered, as well as the
issuance of refresh tokens to public clients, their scope, and
lifetime.

### 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.

### 4.1. Authorization Code

The authorization code grant type is used to obtain both access tokens
and refresh tokens and is optimized for private clients. 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.

```
   +----------+
   | resource |
   |   owner  |
   |          |
   +----------+
        ^
        |
       (B)
   +----|-----+          Client Identifier      +---------------+
   |         -+----(A)-- & Redirection URI ---->|               |
   |  User-   |                                 | Authorization |
   |  Agent  -+----(B)-- User authenticates --->|     Server    |
   |          |                                 |               |
   |         -+----(C)-- Authorization Code ---<|               |
   +-|----|---+                                 +---------------+
     |    |                                       ^       v
    (A)  (C)                                      |       |
     |    |                                       |       |
     ^    v                                       |       |
   +---------+                                    |       |
   |         |>---(D)-- Authorization Code --------'       |
   |  Client |          & Redirection URI                 |
   |         |                                             |
   |         |<---(E)----- Access Token -------------------'
   +---------+       (w/ Optional Refresh Token)
```
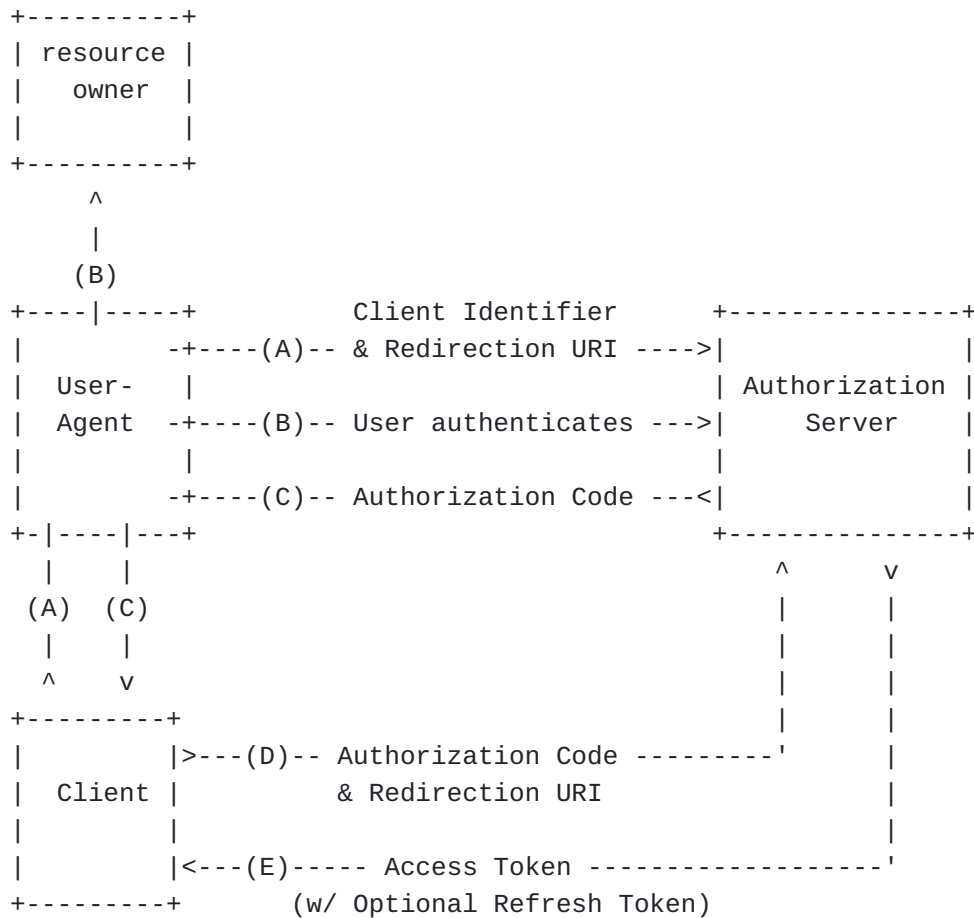
The flow illustrated in <u>Figure 5</u> includes the following steps:

**(%C)**  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).

**(%C)**  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.

**(%C)**
     The client requests an access token from the authorization
   server's token endpoint by including the authorization code received
   in the previous step. When making the request, the client
   authenticates with the authorization server. The client includes the
   redirection URI used to obtain the authorization code for
   verification.

**(%C)**  The authorization server authenticates the client, validates 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

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 2.3.

**redirect_uri**
               OPTIONAL, as described in Section 3.1.2.

**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&state=xyz
    &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

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 MUST expire shortly after it is
> issued to mitigate the risk of leaks. A maximum authorization code
> lifetime of 10 minutes is RECOMMENDED. The client MUST NOT reuse the
> authorization code. If an authorization code is used more than once,
> the authorization server SHOULD attempt to 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=SplxlOBeZQQYbYS6WxSbIA
          &state=xyz
```

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.

#### [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 automatically 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.

  **server_error**
           The authorization server encountered an unexpected
     condition which prevented it from fulfilling the request.

  **temporarily_unavailable**
                  The authorization server is currently
     unable to handle the request due to a temporary overloading or
     maintenance of the server.

**error_description**

   OPTIONAL. A human-readable UTF-8 encoded text providing additional
   information, used to assist the client developer in understanding
   the error that occurred.

**error_uri**

           OPTIONAL. A URI identifying a human-readable web page with
   information about the error, used to provide the client developer
   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&state=xyz
```

### [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.

**code**

     REQUIRED. The authorization code received from the authorization
   server.

**redirect_uri**

               REQUIRED, if the redirect_uri parameter was included in
   the authorization request described in [Section 4.1.1](#), and their
   values MUST be identical.

If the client type is private or was issued client credentials (or
assigned other authentication requirements), the client MUST
authenticate with the authorization server as described in [Section
3.2.1](#).
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;charset=UTF-8

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST:

   *require client authentication for private clients or for any
    client issued client credentials (or with other authentication
    requirements),

   *authenticate the client if client authentication is included and
    ensure the authorization code was issued to the authenticated
    client,

   *verify that the authorization code is valid, and

   *ensure that the redirect_uri parameter is present if the
    redirect_uri parameter was included in the initial authorization
    request described in Section 4.1.1, and that their values are
    identical.

## 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;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

## 4.2. Implicit Grant

The implicit grant type is used to obtain access tokens (it does not
support the issuance of refresh tokens) and is optimized for public
clients known to operate a particular redirection URI. These clients
are 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.
The implicit grant type does not include client authentication, and
relies on the presence of the resource owner and the registration of
the redirection URI. Because the access token is encoded into the
redirection URI, it may be exposed to the resource owner and other
applications residing on its device.

```
            +----------+
            | Resource |
            |  Owner   |
            |          |
            +----------+
                 ^
                 |
               (B)
        +----|-----+          Client Identifier     +---------------+
        |          -+----(A)-- & Redirection URI --->|               |
        |  User-   |                                 | Authorization |
        |  Agent   -|----(B)-- User authenticates -->|     Server    |
        |          |                                 |               |
        |          |<---(C)--- Redirection URI ----<|               |
        |          |             with Access Token    +---------------+
        |          |               in Fragment
        |          |                                  +---------------+
        |          |----(D)--- Redirection URI ---->|   Web-Hosted  |
        |          |             without Fragment    |     Client    |
        |          |                                 |    Resource   |
        |    (F)   |<---(E)------- Script ---------<|               |
        |          |                                 +---------------+
        +-|--------+
          |    |
         (A)  (G) Access Token
          |    |
          ^    v
        +---------+
        |         |
        | Client  |
        |         |
        +---------+
```
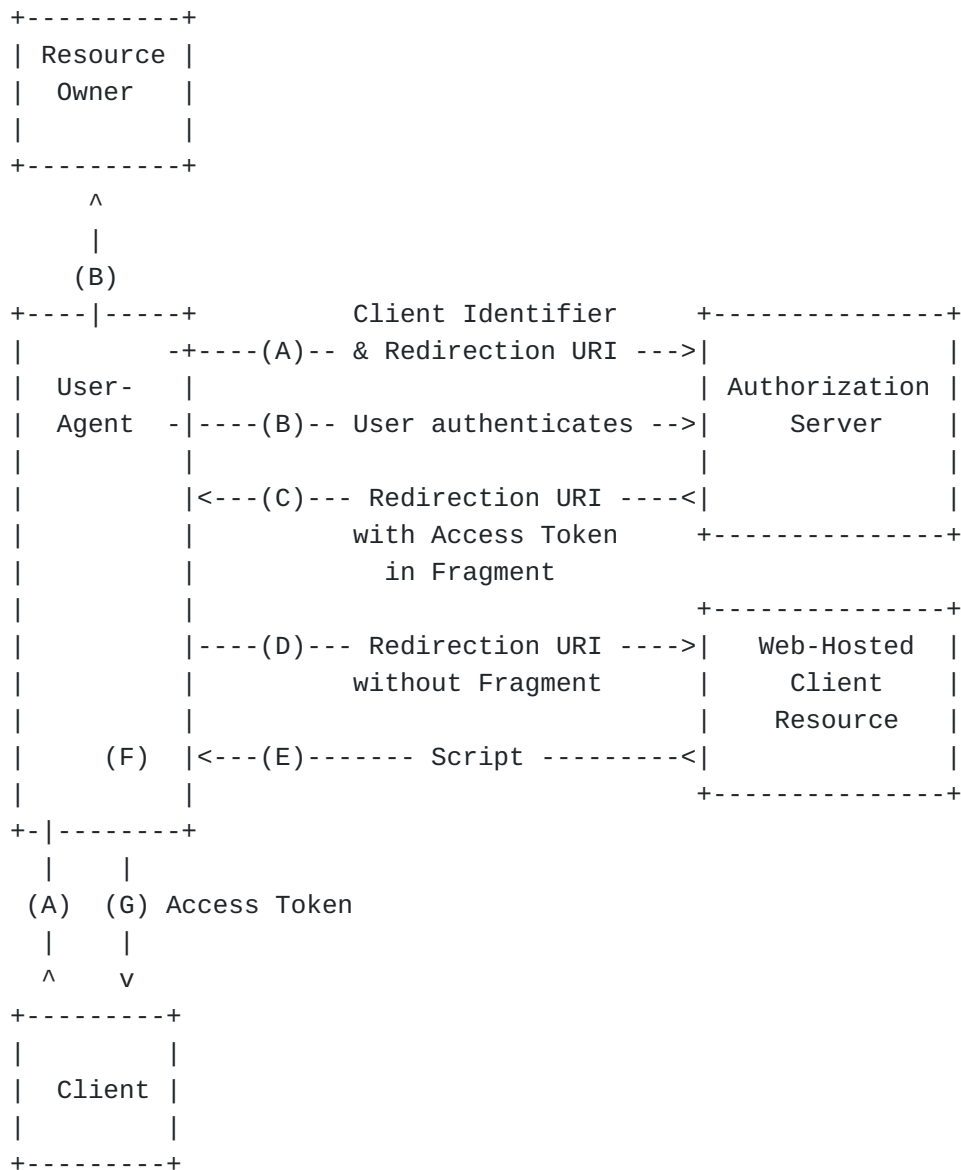
The flow illustrated in [Figure 11](#) includes the following steps:

**(%C)**  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).

**(%C)** 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.

**(%C)** The user-agent follows the redirection instructions by making a request to the web-hosted client resource (which does not include the fragment). The user-agent retains the fragment information locally.

**(%C)** The web-hosted client resource 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.

**(%C)** The user-agent executes the script provided by the web-hosted client resource locally, which extracts the access token and passes it to the client.

### 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 2.3.

**redirect_uri**
OPTIONAL, as described in Section 3.1.2.

**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&state=xyz
    &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. The authorization server MUST verify
that the redirection URI to which it will redirect the access token
matches a redirection URI registered by the client as described in
[Section 3.1.2](#).
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.

### [4.2.2.](#) 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 lifetime in seconds of the access token. 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 token 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 access token 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=2YotnFZFEjr1zCsicMWpAA
          &state=xyz&token_type=example&expires_in=3600
```

Developers should note that some HTTP client implementations do not
support the inclusion of a fragment component in the HTTP Location
response header field. Such client will require using other methods for
redirecting the client than a 3xx redirection response. For example,
returning an HTML page which includes a 'continue' button with an
action linked to the redirection URI.
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.

### 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 automatically 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.

   **server_error**
                The authorization server encountered an unexpected
      condition which prevented it from fulfilling the request.

   **temporarily_unavailable**
                          The authorization server is currently
      unable to handle the request due to a temporary overloading or
      maintenance of the server.

**error_description**
                  OPTIONAL. A human-readable UTF-8 encoded text
   providing additional information, used to assist the client
   developer in understanding the error that occurred.

**error_uri**
          OPTIONAL. A URI identifying a human-readable web page with
   information about the error, used to provide the client developer
   with additional information about the error.

**state**
      REQUIRED if a valid state parameter was present in the client

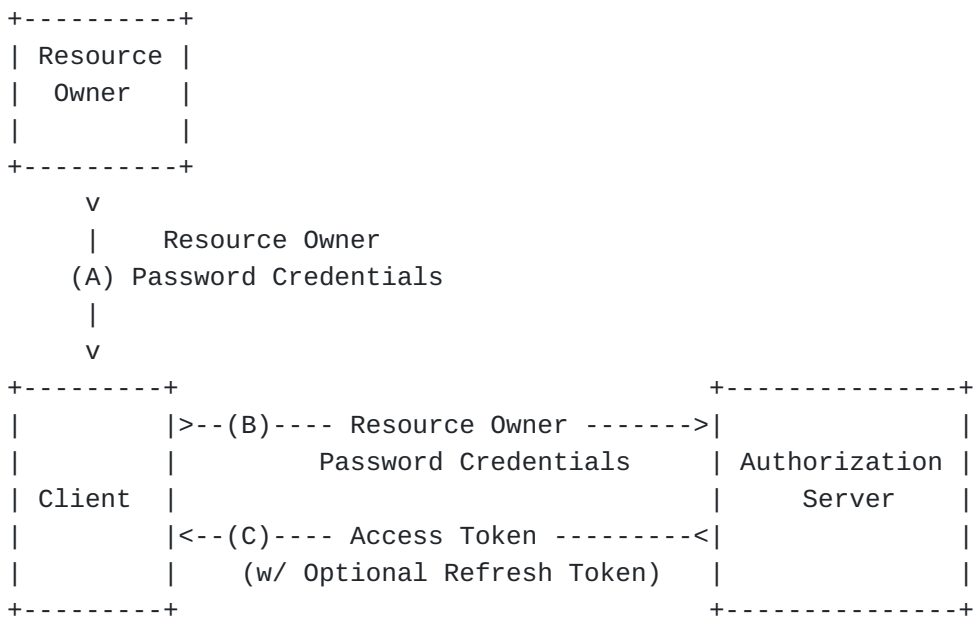authorization request. Set to the exact value received from the client.

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&state=xyz
```

## 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 device 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 to an access token.

```
+----------+
| Resource |
|  Owner   |
|          |
+----------+
     v
     |    Resource Owner
    (A) Password Credentials
     |
     v
+---------+                                +---------------+
|         |>--(B)---- Resource Owner ------->|               |
|         |          Password Credentials   | Authorization |
| Client  |                                 |    Server     |
|         |<--(C)---- Access Token ---------<|               |
|         |      (w/ Optional Refresh Token) |               |
+---------+                                +---------------+
```

The flow illustrated in Figure 15 includes the following steps:

**(%C)**

The resource owner provides the client with its username and password.

**(%C)**  The client requests an access token from the authorization server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.

**(%C)**  The authorization server authenticates the client and validates the resource owner credentials, and if valid issues an access token.

### [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.

### [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.

**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.

If the client type is private or was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in [Section 3.2.1](#).
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;charset=UTF-8

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

The authorization server MUST:

   *require client authentication for private clients or for any
    client issued client credentials (or with other authentication
    requirements),

   *authenticate the client if client authentication is included, and

   *validate the resource owner password credentials.

Since this access token request utilizes the resource owner's password,
the authorization server MUST protect the endpoint against brute force
attacks.

## 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
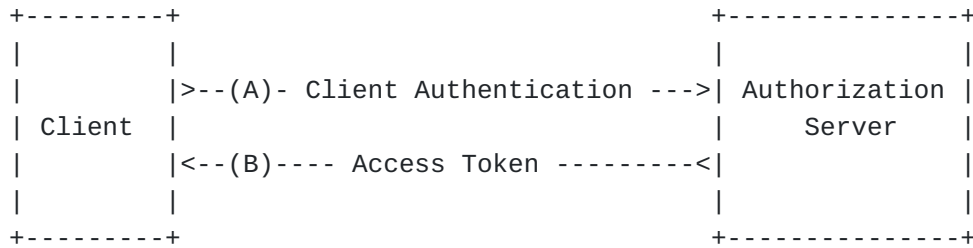invalid, the authorization server returns an error response as
described in Section 5.2.
An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

## 4.4. Client Credentials

The client can request an access token using only its client
credentials (or other supported means of authentication) 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).
The client credentials grant type MUST only be used by private clients.

```
  +---------+                               +---------------+
  |         |                               |               |
  |         |>--(A)- Client Authentication --->| Authorization |
  | Client  |                               |     Server    |
  |         |<--(B)---- Access Token ---------<|               |
  |         |                               |               |
  +---------+                               +---------------+
```

The flow illustrated in Figure 18 includes the following steps:

(%C)  The client authenticates with the authorization server and
   requests an access token from the token endpoint.

(%C)  The authorization server authenticates the client, and if valid
   issues an access token.

### 4.4.1. Authorization Request and Response

Since the client authentication is used as the authorization grant, no
additional authorization request is needed.

### 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.

**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 MUST authenticate with the authorization server as described in Section 3.2.1.

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;charset=UTF-8

grant_type=client_credentials
```

The authorization server MUST authenticate the client.

### 4.4.3. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token as described in Section 5.1. A refresh token SHOULD NOT be included. 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;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "example_parameter":"example_value"
}
```

### 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;charset=UTF-8

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.

## 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.

### 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 lifetime in seconds of the access token. 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 token 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 access token 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, credentials, or other sensitive information, as well as the Pragma response header field [RFC2616] with a value of no-cache. For example:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "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.

## 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 authentication included, multiple client
      authentications included, or unsupported authentication method).
      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 UTF-8 encoded text
   providing additional information, used to assist the client
   developer in understanding the error that occurred.

**error_uri**
             OPTIONAL. A URI identifying a human-readable web page with
   information about the error, used to provide the client developer
   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;charset=UTF-8
  Cache-Control: no-store
  Pragma: no-cache

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

## 6. Refreshing an Access Token

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.

**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.

Because refresh tokens are typically long-lasting credentials used to
request additional access tokens, the refresh token is bound to the
client it was issued. If the client type is private or was issued
client credentials (or assigned other authentication requirements), the
client MUST authenticate with the authorization server as described in
Section 3.2.1.

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;charset=UTF-8

  grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

The authorization server MUST:

   *require client authentication for private clients or for any
    client issued client credentials (or with other authentication
    requirements),

   *authenticate the client if client authentication is included and
    ensure 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. The authorization server MAY revoke the old refresh token after issuing a new refresh token to the client. If a new refresh token is issued, its scope MUST be identical to that of the refresh token included in the request.

## [7.](#) [Accessing Protected Resources](#)

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

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 or does not trust 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

## 8.1. Defining Access Token Types

Access token types can be defined in one of two ways: registered in the
access token type registry (following the procedures in Section 11.1),
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

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 11.2.
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

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
11.2.

## 8.4. Defining New Authorization Endpoint Response Types

[[ Pending consensus ]]
New response types for use with the authorization endpoint are defined
and registered in the authorization endpoint response type registry
following the procedure in Section 11.3. Response type names MUST
conform to the response-type ABNF.

```
response-type  = response-name *( "+" response-name )
response-name  = 1*response-char
response-char  = "_" / DIGIT / ALPHA
```

The "+" character is reserved for defining composite response types
made up of two or more existing registered response types. Only one
response type of each combination may be registered and used for making
requests. Composite response types are treated and compared in the same
as manner as non-composite response types. The "+" notation is meant
only to improve human readability and is not used for machine parsing.
For example, an extension can define and register the token+code
response type. However, once registered, the same combination cannot be
registered as code+token, or used to make an authorization request.

## 8.5. Defining Additional Error Codes

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 11.4) 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. Native Applications

Native applications are clients installed and executed on the resource
owner's device (i.e. desktop application, native mobile application).
Native applications may require special consideration related to
security, platform capabilities, and overall end-user experience.
The authorization endpoint requires interaction between the client and
the resource owner's user-agent. Native applications can invoke an
external user-agent or embed a user-agent within the application. For
example:

*External user-agent - the native application can capture the
     response from the authorization server using a redirection URI
     with an scheme registered with the operating system to invoke the
     client as the handler, manual copy-and-paste of the credentials,
     running a local web server, installing a user-agent extension, or
     by providing a redirection URI identifying a server-hosted
     resource under the client's control, which in turn makes the
     response available to the native application.

    *Embedded user-agent - the native application obtains the response
     by directly communicating with the embedded user-agent by
     monitoring state changes emitted during the resource load, or
     accessing the user-agent's cookies storage.

When choosing between an external or embedded user-agent, developers
should consider:

    *External user-agents may improve completion rate as the resource
     owner may already have an active session with the authorization
     server removing the need to re-authenticate. It provides a
     familiar end-user experience and functionality. The resource
     owner may also rely on user-agent features or extensions to
     assist with authentication (e.g. password manager, 2-factor
     device reader).

    *Embedded user-agents may offer an improved usability, as they
     remove the need to switch context and open new windows.

    *Embedded user-agents pose a security challenge because resource
     owners are authenticating in an unidentified window without
     access to the visual protections found in most external user-
     agents. Embedded user-agents educate end-user to trust
     unidentified requests for authentication (making phishing attacks
     easier to execute).

When choosing between the implicit grant type and the authorization
code grant type, the following should be considered:

    *Native applications that use the authorization code grant type
     SHOULD do so without using client credentials, due to the native
     application's inability to keep credentials confidential.

    *When using the implicit grant type flow a refresh token is not
     returned.

## 10. Security Considerations

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. Resource owners access the client via an HTML user interface rendered in a user-agent on the resource-owner'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 resource owner.

**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 resource owner's device. Protocol data and credentials are easily accessible (and often visible) to the resource owner. Since such applications reside within the user-agent, they can make seamless use of the user-agent capabilities when requesting authorization.

**Native Application**
A native application is a client installed and executed on the resource owner's device. Protocol data and credentials are accessible to the resource owner. It is assumed that any client authentication credentials included in the application can be extracted, and furthermore that rotation of the client authentication credentials is impractical. On the other hand, dynamically issued credentials such access tokens or refresh tokens, can receive an acceptable level of protection. At a minimum, these credentials are protected from hostile servers which the application may interact with. On some platform these credentials might be protected from other applications residing on the same device.

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

## 10.1. Client Authentication

The authorization server establishes client credentials with web application clients for the purpose of client authentication. The authorization server is encouraged to consider stronger client authentication means than a client password. Web application clients MUST ensure confidentiality of client passwords and other client credentials.
The authorization server MUST NOT issue client passwords or other client credentials to native application or user-agent-based

application clients 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 client on a specific device.

## 10.2. Client Impersonation

A malicious client can impersonate another client and obtain access to protected resources, if the impersonated client fails to, or is unable to, keep is client credentials confidential.
The authorization server MUST authenticate the client whenever possible. If the authorization server cannot authenticate the client due to the client nature, the authorization server MUST require the registration of any redirection URI used for receiving authorization, and SHOULD utilize other means to protect resource owners from such malicious clients. For example, engage the resource owner to assist in identifying the client and its origin.
The authorization server SHOULD enforce explicit resource owner authentication and provide the resource owner with information about the client and the requested authorization scope and lifetime. It is up to the resource owner to review the information in the context of the current client, and authorize the request.
The authorization server SHOULD NOT process repeated authorization requests automatically (without active resource owner interaction) without authenticating the client or relying on other measures to ensure the repeated request comes from an authentic client and not an impersonator.

## 10.3. Access Tokens

Access token (as well as any type-specific attributes) MUST be kept confidential in transit and storage, and only shared among the authorization server, the resource servers the access token is valid for, and the client to whom the access token is issued.
When using the implicit grant type, the access token is transmitted in the URI fragment, which can expose it to unauthorized parties.
The authorization server MUST ensure that access tokens cannot be generated, modified, or guessed to produce valid access tokens.
The client SHOULD request access tokens with the minimal scope and lifetime necessary. The authorization server SHOULD take the client identity into account when choosing how to honor the requested scope and lifetime, and MAY issue an access token with a less rights than requested.

## 10.4. Refresh Tokens

Authorization servers MAY issue refresh tokens to web application clients and native application clients.

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 binding between a refresh token and the client to whom it was issued. The authorization server MUST verify the binding between the refresh token and client identity whenever the client identity can be authenticated. When client authentication is not possible, the authorization server SHOULD deploy other means to detect refresh token abuse [[ add example ]].
The authorization server MUST ensure that refresh tokens cannot be generated, modified, or guessed to produce valid refresh tokens.

## [10.5.](#) Authorization Codes

The transmission of authorization codes SHOULD be made over a secure channel, and the client SHOULD implement TLS for use with its redirection URI if the URI identifies a network resource. Effort should be made to keep authorization codes 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 operate as plaintext bearer credentials, used to verify that the resource owner who granted authorization at the authorization server, is the same resource owner returning to the client to complete the process. Therefore, if the client relies on the authorization code for its own resource owner authentication, the client redirection endpoint MUST require TLS.
Authorization codes MUST be short lived and single use. If the authorization server observes multiple attempts to exchange an authorization code for an access token, the authorization server SHOULD attempt to revoke all access tokens already granted based on the compromised authorization code.
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.

## [10.6.](#) Authorization Code Leakage

Session fixation attacks leverage the authorization code grant type, by tricking a resource owner to authorize access to a legitimate client, but using a client account under the control of the attacker. The only difference between a valid request and the attack request 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 (via the client).
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 register their redirection URI and if provided, MUST validate the redirection URI received in the authorization request against the registered value.

### 10.7. 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 by the client, but does not eliminate the need to expose highly privileged credentials to the client.
This grant type carries a higher risk than other grant types because it maintains the password anti-pattern this protocol 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 lifetime than desired by the resource owner. The authorization server SHOULD restrict the scope and lifetime 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.

### 10.8. Request Confidentiality

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

### 10.9. Endpoints Authenticity

In order to prevent man-in-the-middle and phishing attacks, the authorization server MUST implement and require TLS with server authentication as defined by [RFC2818] for any request sent to the authorization and token endpoints. The client MUST validate the authorization server's TLS certificate in accordance with its requirements for server identity authentication.

### 10.10. Credentials Guessing Attacks

The authorization server MUST prevent attackers from guessing access tokens, authorization codes, refresh tokens, resource owner passwords, and client credentials.
When generating tokens and other credentials not intended for handling by end-users, the authorization server MUST use a reasonable level of entropy in order to mitigate the risk of guessing attacks. The

authorization server MUST utilize other means to protect credentials
intended for end-user usage.

### 10.11. Phishing Attacks

Wide deployment of this and similar protocols may cause end-users to
become inured to the practice of being redirected to websites where
they are asked to enter their passwords. If end-users are not careful
to verify the authenticity of these websites before entering their
credentials, it will be possible for attackers to exploit this practice
to steal resource owners' passwords.
Service providers should attempt to educate end-users about the risks
phishing attacks pose, and should provide mechanisms that make it easy
for end-users to confirm the authenticity of their sites. Client
developers should consider the security implications of how they
interact with the user-agent (e.g., external, embedded), and the
ability of the end-user to verify the authenticity of the authorization
server.
To reduce the risk of phishing attacks, the authorization servers MUST
utilize TLS on every endpoint used for end-user interaction.

### 10.12. Cross-Site Request Forgery

Cross-site request forgery (CSRF) is a web-based attack whereby HTTP
requests are transmitted from the user-agent of an end-user the server
trusts or has authenticated. CSRF attacks on the authorization endpoint
can allow an attacker to obtain authorization without the consent of
the resource owner.
The state request parameter SHOULD be used to mitigate against CSRF
attacks, particularly for login CSRF attacks. CSRF attacks against the
client's redirection URI allow an attacker to inject their own
authorization code or access token, which can result in the client
using an access token associated with the attacker's account rather
than the victim's. Depending on the nature of the client and the
protected resources, this can have undesirable and damaging effects.
It is strongly RECOMMENDED that the client includes the state request
parameter with authorization requests to the authorization server. The
state request parameter MUST contain a non-guessable value, and the
client MUST keep it in a location accessible only by the client or the
user-agent (i.e., protected by same-origin policy).
For example, using a DOM variable (protected by JavaScript or other
DOM-binding language's enforcement of SOP [[ add reference ]]), HTTP
cookie, or HTML5 client-side storage. The authorization server includes
the value of the state parameter when redirecting the user-agent back
to the client which MUST then ensure the received value matches the
stored value.

## 10.13. Clickjacking

[[ Rework to use specification terminology ]]
Clickjacking is the process of tricking end-users into revealing confidential information or taking control of their device while clicking on seemingly innocuous web pages. In more detail, a malicious site loads the target site in a transparent iframe overlaid on top of a set of dummy buttons which are carefully constructed to be placed directly under important buttons on the target site. When a user clicks a visible button, they are actually clicking a button (such as an "Authorize" button) on the hidden page.
To prevent clickjacking (and phishing attacks), native applications SHOULD use external browsers instead of embedding browsers in an iframe when requesting end-user authorization. For newer browsers, avoidance of iframes can be enforced by the authorization server using the x-frame-options header [[ Add reference ]]. This header can have two values, deny and sameorigin, which will block any framing or framing by sites with a different origin, respectively. For older browsers, javascript framebusting techniques can be used but may not be effective in all browsers.

## 11. IANA Considerations

## 11.1. 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.

### 11.1.1. 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 11.2.

**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.

### 11.2. 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.

### 11.2.1. 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.

### 11.2.2. 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 ]]

## 11.3.  The OAuth Authorization Endpoint Response Type Registry

This specification establishes the OAuth authorization endpoint
response type registry.
Additional response type for use with the authorization endpoint 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
response 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.

## 11.3.1.  Registration Template

**Response type name:**
                    The name requested (e.g., "example").

**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
   type, 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.

## 11.3.2. Initial Registry Contents

The OAuth Authorization Endpoint Response Type Registry's initial
contents are:

   *Response type name: code

   *Change controller: IETF

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

   *Response type name: token

   *Change controller: IETF

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

## 11.4. 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.

### 11.4.1. Registration Template

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

**Error usage location:**
> 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.

### 12. Acknowledgements

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, Brian Eaton, 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, Dick
Hardt, 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, Anthony Nadalin, 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,
Shane Weeden, and Skylar Woodward.

## 13. References

### 13.1. Normative References

| | |
|---|---|
| **[RFC2119]** | Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997. |
| **[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. |
| **[RFC2617]** | Franks, J., Hallam-Baker, P.M., Hostetler, J.L., Lawrence, S.D., Leach, P.J., Luotonen, A. and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999. |
| **[RFC2818]** | Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000. |
| **[RFC3986]** | Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005. |
| **[RFC4627]** | Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006. |
| **[RFC4949]** | Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007. |
| **[RFC5226]** | |

| | Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008. |
|---|---|
| **[RFC5234]** | Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008. |
| **[RFC5246]** | Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008. |
| **[W3C.REC-html401-19991224]** | Raggett, D., Hors, A. and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999. |

## 13.2. Informative References

| **[RFC5849]** | Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010. |
|---|---|
| **[I-D.ietf-oauth-v2-bearer]** | Jones, M, Hardt, D and D Recordon, "The OAuth 2.0 Protocol: Bearer Tokens", Internet-Draft draft-ietf-oauth-v2-bearer-04, March 2011. |
| **[I-D.ietf-oauth-saml2-bearer]** | Campbell, B and C Mortimore, "SAML 2.0 Bearer Assertion Grant Type Profile for OAuth 2.0", Internet-Draft draft-ietf-oauth-saml2-bearer-03, February 2011. |
| **[I-D.ietf-oauth-v2-http-mac]** | Hammer-Lahav, E, Barth, A and B Adida, "HTTP Authentication: MAC Access Authentication", Internet-Draft draft-ietf-oauth-v2-http-mac-00, May 2011. |
| **[I-D.lodderstedt-oauth-security]** | Lodderstedt, T, McGloin, M and P Hunt, "OAuth 2.0 Threat Model and Security Considerations", Internet-Draft draft-lodderstedt-oauth-security-01, March 2011. |
| **[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. |
| **[I-D.draft-hardt-oauth-01]** | Hardt, D, Tom, A, Eaton, B and Y Goland, "OAuth Web Resource Authorization Profiles", January 2010. |

## Authors' Addresses

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

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

Dick Hardt Hardt Microsoft EMail: [dick.hardt@gmail.com](mailto:dick.hardt@gmail.com) URI: [http://dickhardt.org/](http://dickhardt.org/)