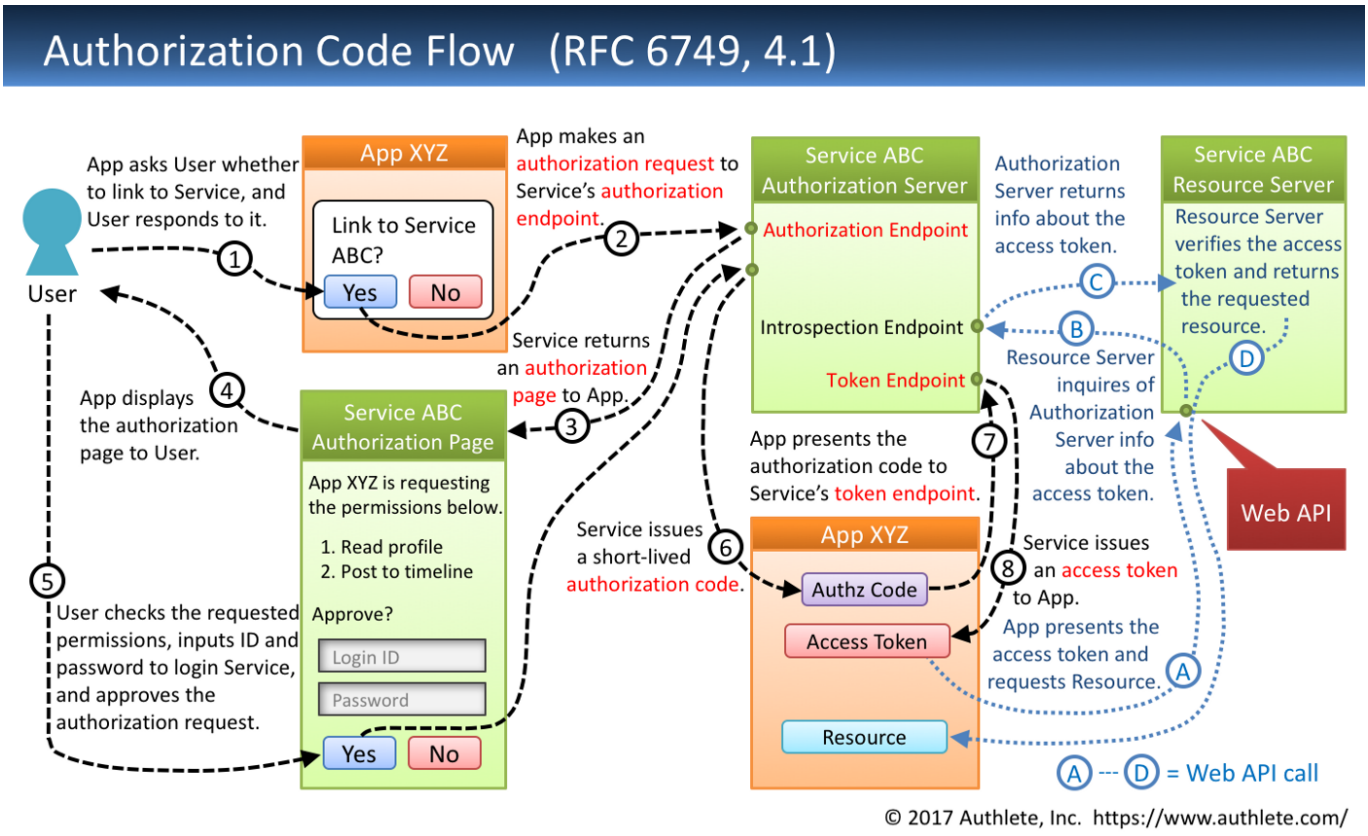
Diagrams And Movies Of All The OAuth 2.0 Flows

Takahiko Kawasaki Follow
May 27, 2017 · 6 min read

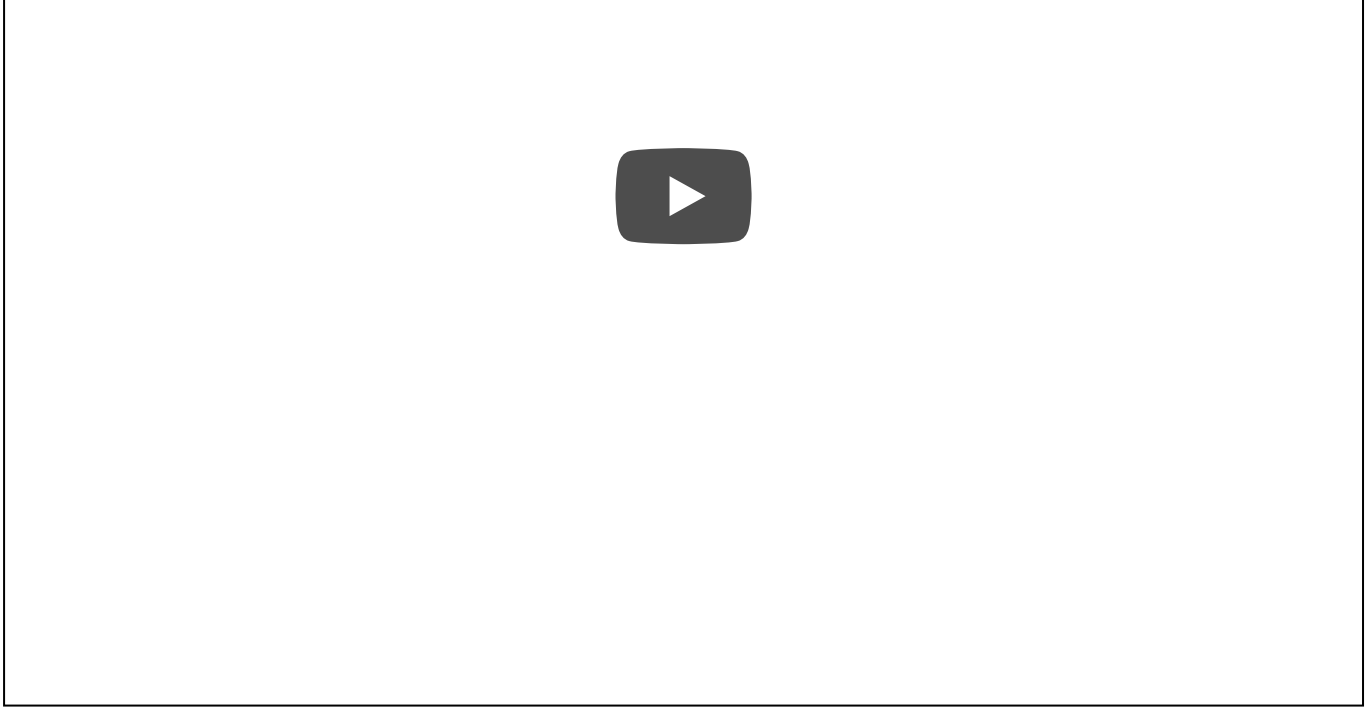
Diagrams and movies of all the 4 authorization flows defined in RFC 6749 (The OAuth 2.0 Authorization Framework) and one more flow to re-issue an access token using a refresh token.

1. Authorization Code Flow

This is the flow defined in RFC 6749, 4.1. Authorization Code Grant. A client application (a) makes an authorization request to an authorization endpoint, (b) receives a short-lived authorization code, (c) makes a token request to a token endpoint with the authorization code, and (d) gets an access token.



OAuth 2.0, Authorization code flow



1.1. Request To Authorization Endpoint

```
GET {Authorization Endpoint}
  ?response_type=code              // - Required
  &client_id={Client ID}           // - Required
  &redirect_uri={Redirect URI}     // - Conditionally required
  &scope={Scopes}                  // - Optional
  &state={Arbitrary String}        // - Recommended
  &code_challenge={Challenge}      // - Optional
  &code_challenge_method={Method} // - Optional
  HTTP/1.1
HOST: {Authorization Server}
```

Note: The snippet above contains request parameters from RFC 7636 in addition to ones from RFC 6749. See PKCE Authorization Request for details.

1.2. Response From Authorization Endpoint

```
HTTP/1.1 302 Found
Location: {Redirect URI}
  ?code={Authorization Code}  // - Always included
  &state={Arbitrary String}   // - Included if the authorization
                              //   request included 'state'.
```

1.3. Request To Token Endpoint

```
POST {Token Endpoint} HTTP/1.1
Host: {Authorization Server}
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code  // - Required
&code={Authorization Code}     // - Required
&redirect_uri={Redirect URI}   // - Required if the authorization
                               //   request included 'redirect_uri'.
&code_verifier={Verifier}      // - Required if the authorization
                               //   request included
                               //   'code_challenge'.
```

Note: The snippet above contains request parameters from RFC 7636 in addition to ones from RFC 6749. See PKCE Token Request for details.

If the client type of the client application is "public", the `client_id` request parameter is additionally required. On the other hand, if the client type is "confidential", depending on the client authentication method, an `Authorization` HTTP header, a pair of `client_id` & `client_secret` parameters, or some other input parameters are required. See *"OAuth 2.0 Client Authentication"* for details.

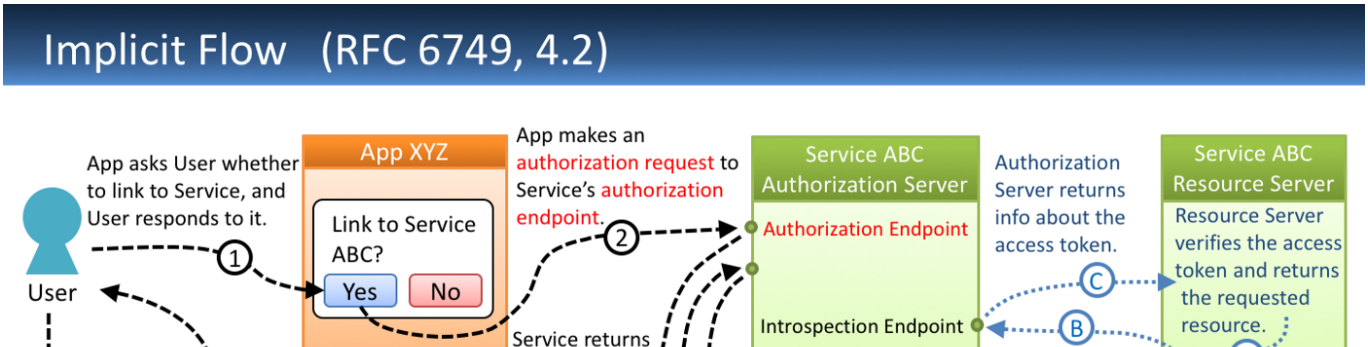
1.4. Response From Token Endpoint

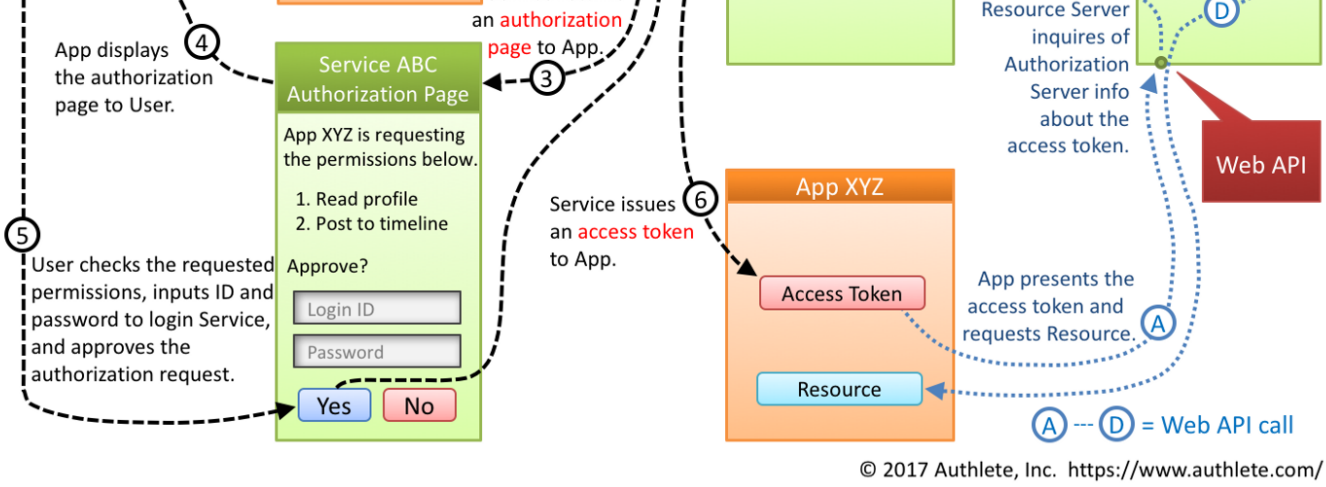
```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "{Access Token}",     // - Always included
  "token_type": "{Token Type}",         // - Always included
  "expires_in": {Lifetime In Seconds}, // - Optional
  "refresh_token": "{Refresh Token}",   // - Optional
  "scope": "{Scopes}"                   // - Mandatory if the granted
                                        //   scopes differ from the
                                        //   requested ones.
}
```

2. Implicit Flow

This is the flow defined in RFC 6749, 4.2. Implicit Grant. A client application (a) makes an authorization request to an authorization endpoint and (b) gets an access token directly from the authorization endpoint.





OAuth 2.0, Implicit flow

2.1. Request To Authorization Endpoint

```
GET {Authorization Endpoint}
  ?response_type=token           // - Required
  &client_id={Client ID}         // - Required
  &redirect_uri={Redirect URI}   // - Conditionally required
  &scope={Scopes}                // - Optional
  &state={Arbitrary String}      // - Recommended
  HTTP/1.1
HOST: {Authorization Server}
```

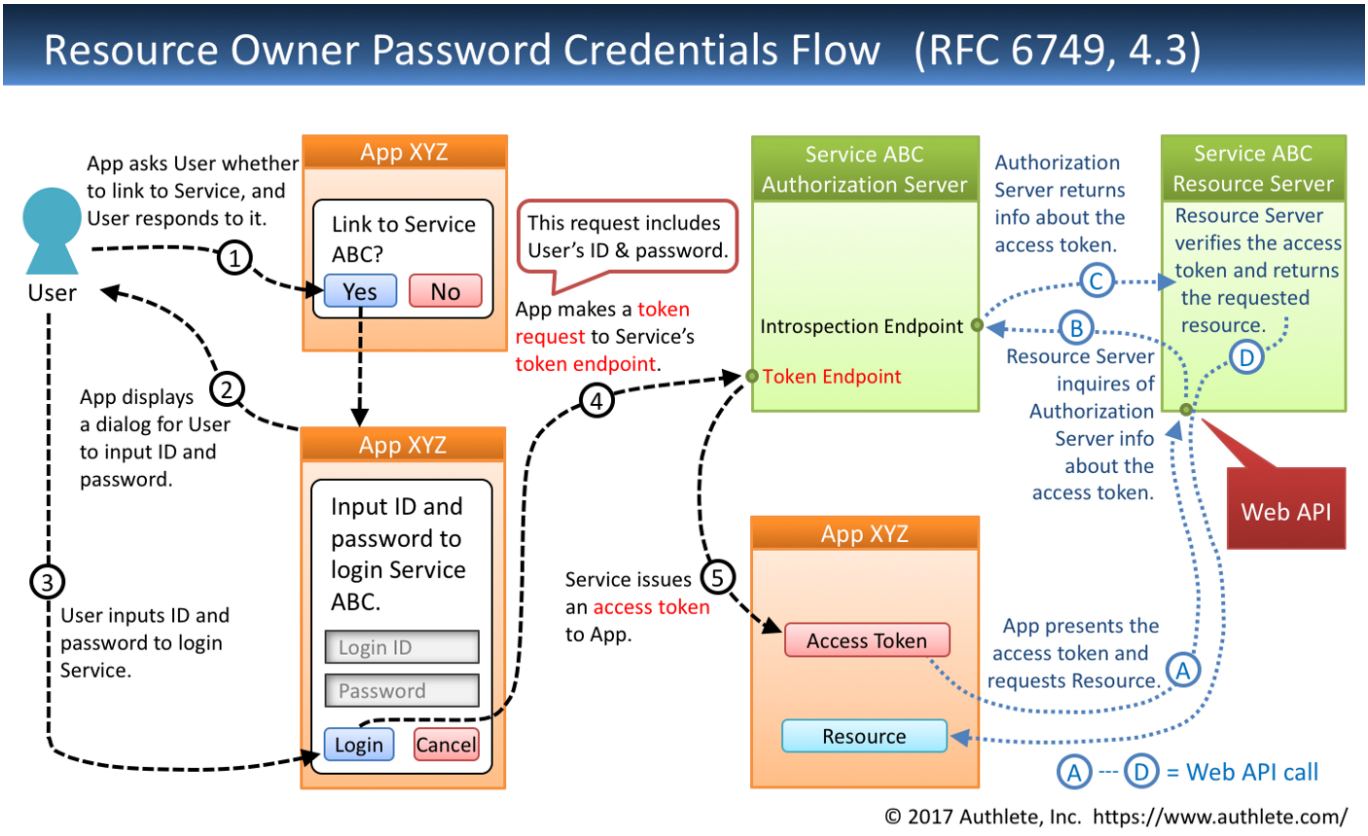
2.2. Response From Authorization Endpoint

```
HTTP/1.1 302 Found
Location: {Redirect URI}
  #access_token={Access Token}          // - Always included
  &token_type={Token Type}              // - Always included
  &expires_in={Lifetime In Seconds}     // - Optional
  &state={Arbitrary String}             // - Included if the request
                                        //   included 'state'.
  &scope={Scopes}                       // - Mandatory if the granted
                                        //   scopes differ from the
                                        //   requested ones.
```

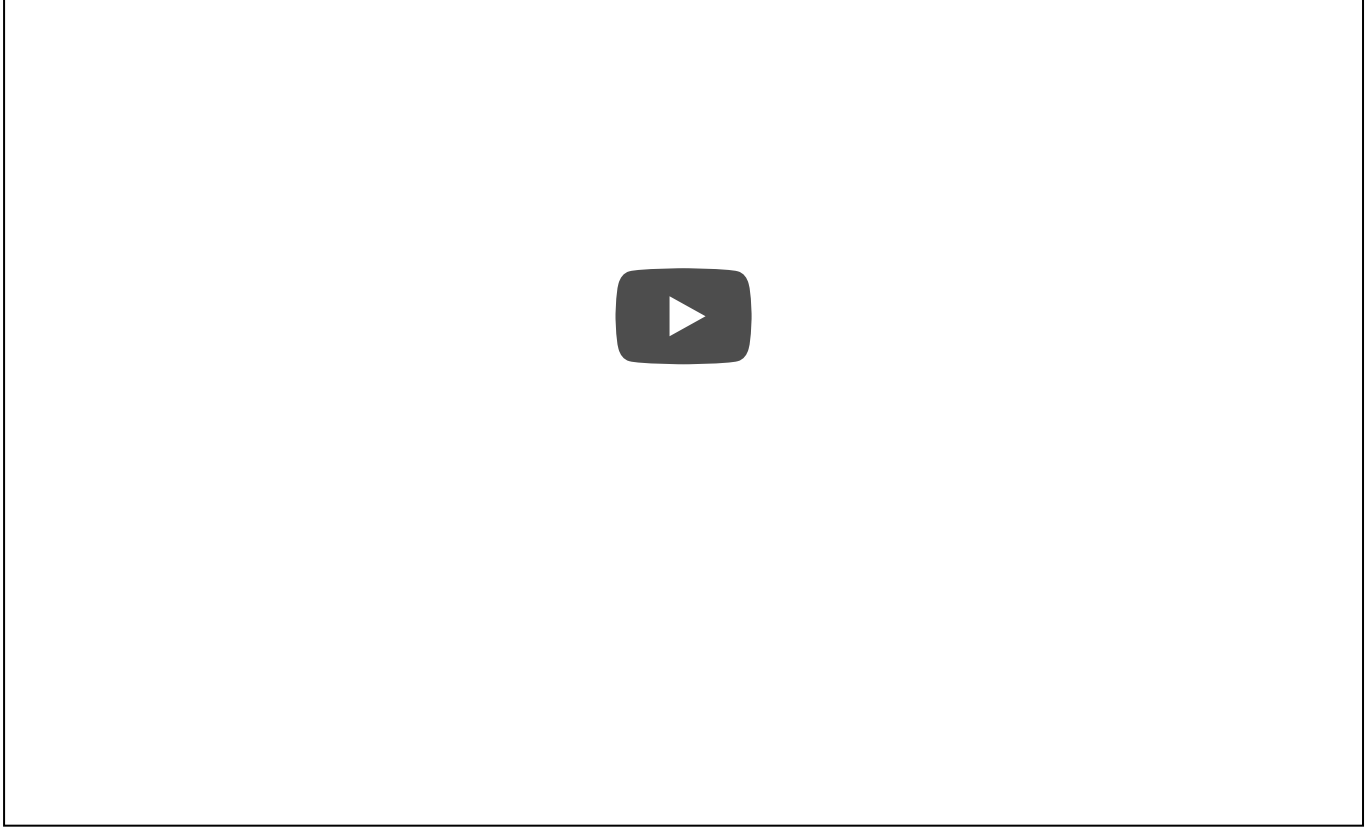
Implicit Flow does not issue refresh tokens.

3. Resource Owner Password Credentials Flow

This is the flow defined in RFC 6749, 4.3. Resource Owner Password Credentials Grant. A client application (a) makes a token request to a token endpoint and (b) gets an access token. In this flow, a client application accepts a user's ID and password although the primary purpose of OAuth 2.0 is to give limited permissions to a client application WITHOUT revealing the user's credentials to the client application.



OAuth 2.0, Resource owner password credentials flow



3.1. Request To Token Endpoint

```
POST {Token Endpoint} HTTP/1.1
Host: {Authorization Server}
Content-Type: application/x-www-form-urlecoded

grant_type=password    // - Required
&username={User ID}    // - Required
&password={Password}   // - Required
&scope={Scopes}        // - Optional
```

If the client type of the client application is "public", the `client_id` request parameter is additionally required. On the other hand, if the client type is "confidential", depending on the client authentication method, an `Authorization` HTTP header, a pair of `client_id` & `client_secret` parameters, or some other input parameters are required. See *"OAuth 2.0 Client Authentication"* for details.

3.2. Response From Token Endpoint

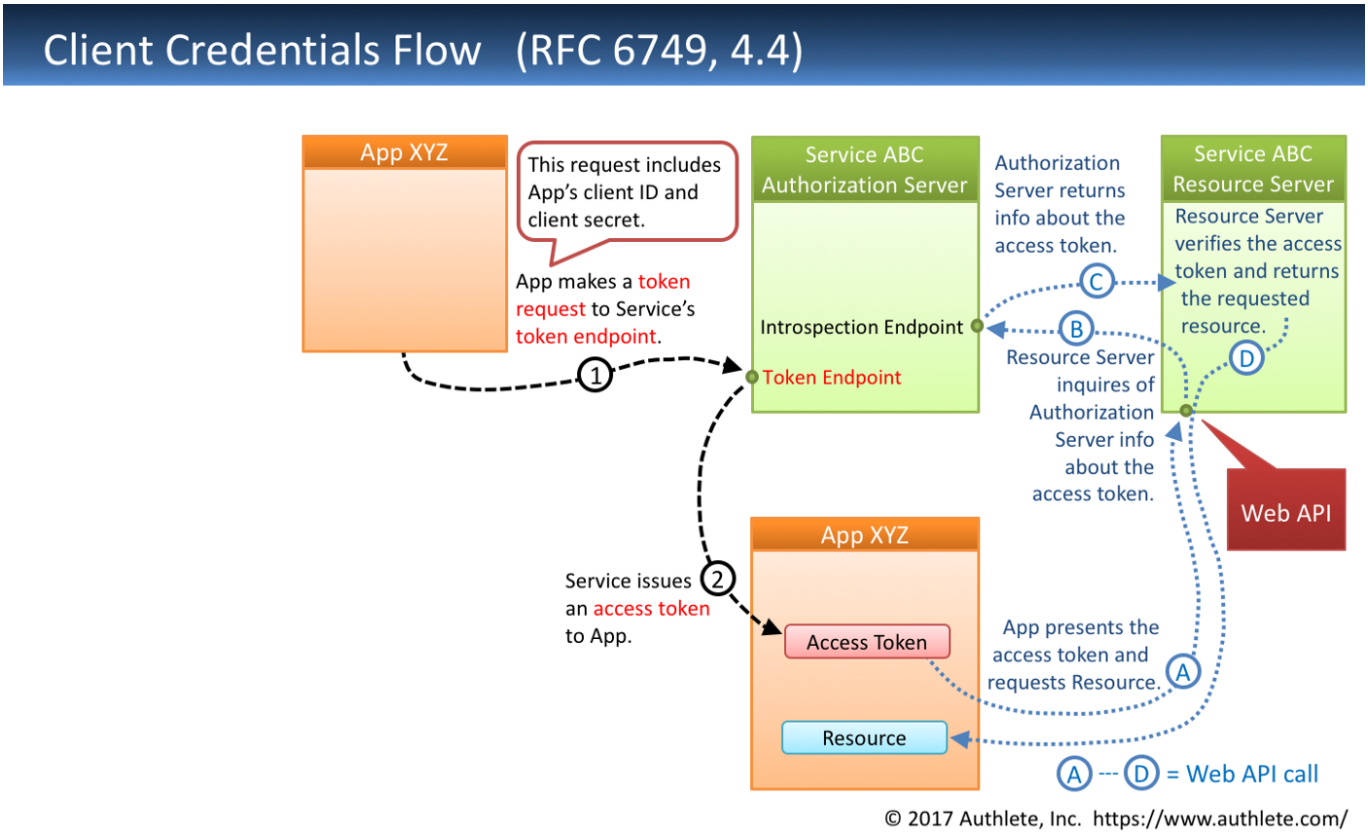
```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "{Access Token}",     // - Always included
  "token_type": "{Token Type}",         // - Always included
  "expires_in": {Lifetime In Seconds}, // - Optional
```

```
    "refresh_token": "{Refresh Token}",   // - Optional
    "scope": "{Scopes}"                   // - Mandatory if the granted
                                          //   scopes differ from the
                                          //   requested ones.
}
```

4. Client Credentials Flow

This is the flow defined in RFC 6749, 4.4. Client Credentials Grant. A client application (a) makes a token request to a token endpoint and (b) gets an access token. In this flow, user authentication is not performed and client application authentication only is performed.



OAuth 2.0, Client credentials flow

4.1. Request To Token Endpoint

```
POST {Token Endpoint} HTTP/1.1
Host: {Authorization Server}
Authorization: Basic {Client Credentials}
Content-Type: application/x-www-form-urlecoded

grant_type=client_credentials  // - Required
&scope={Scopes}                // - Optional
```

Client Credentials Flow is allowed only for *confidential clients* (cf. RFC 6749, 2.1. Client Types). As a result, `Authorization` header, a pair of `client_id` & `client_secret` parameters, or some other input parameters for client authentication are required. See *"OAuth 2.0 Client Authentication"* for details.

4.2. Response From Token Endpoint

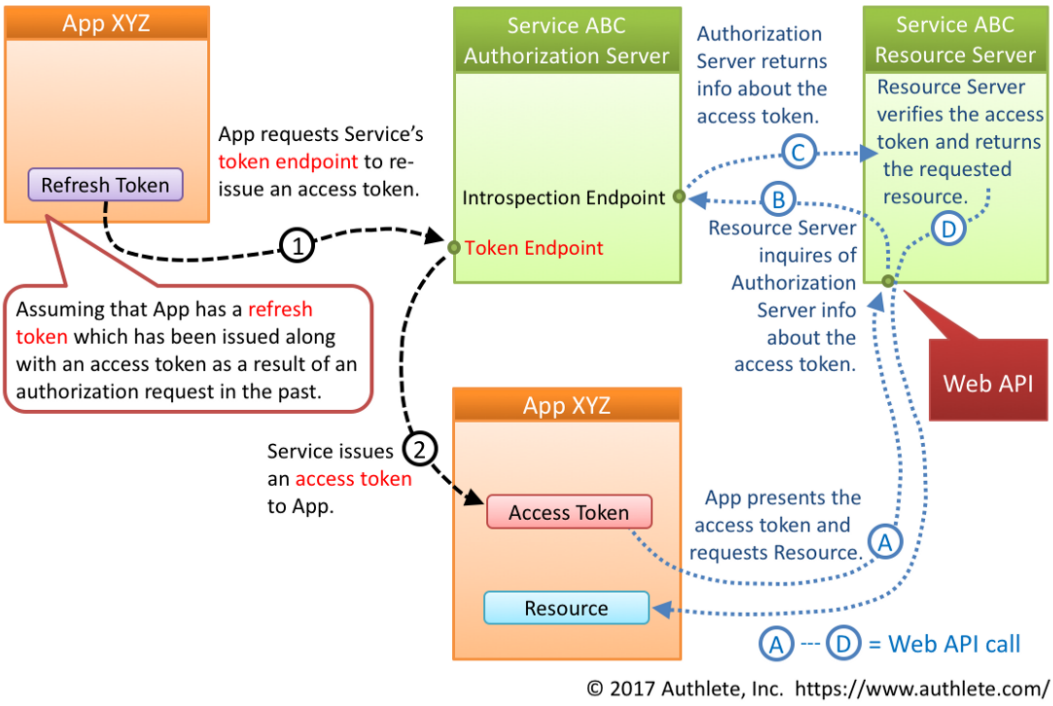
```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "{Access Token}",    // - Always included
  "token_type": "{Token Type}",        // - Always included
  "expires_in": {Lifetime In Seconds}, // - Optional
  "scope": "{Scopes}"                  // - Mandatory if the granted
                                       //   scopes differ from the
                                       //   requested ones.
}
```

The specification says Client Credentials Flow should not issue refresh tokens.

5. Refresh Token Flow

This is the flow defined in RFC 6749, 6. Refreshing an Access Token. A client application (a) presents a refresh token to a token endpoint and (b) gets a new access token.



OAuth 2.0, Refresh token flow

5.1. Request To Token Endpoint

```
POST {Token Endpoint} HTTP/1.1
Host: {Authorization Server}
```

```
Content-Type: application/x-www-form-urlecoded

grant_type=refresh_token          // - Required
&refresh_token={Refresh Token}    // - Required
&scope={Scopes}                   // - Optional
```

If the client type of the client application is "public", the `client_id` request parameter is additionally required. On the other hand, if the client type is "confidential", depending on the client authentication method, an `Authorization` HTTP header, a pair of `client_id` & `client_secret` parameters, or some other input parameters are required. See *"OAuth 2.0 Client Authentication"* for details.

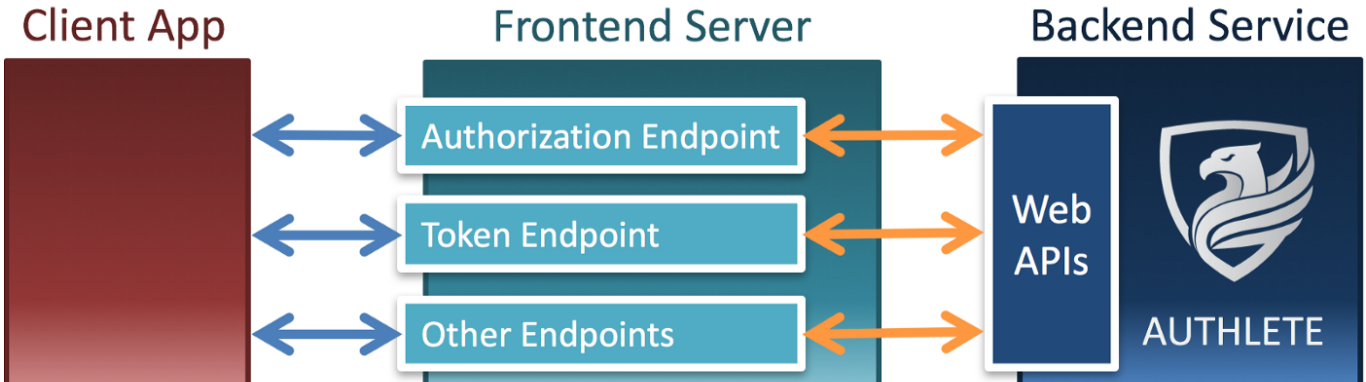
5.2. Response From Token Endpoint

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

{
  "access_token": "{Access Token}",     // - Always included
  "token_type": "{Token Type}",         // - Always included
  "expires_in": {Lifetime In Seconds}, // - Optional
  "refresh_token": "{Refresh Token}",   // - Optional
  "scope": "{Scopes}"                   // - Mandatory if the granted
                                        //   scopes differ from the
                                        //   original ones.
}
```

Appendix

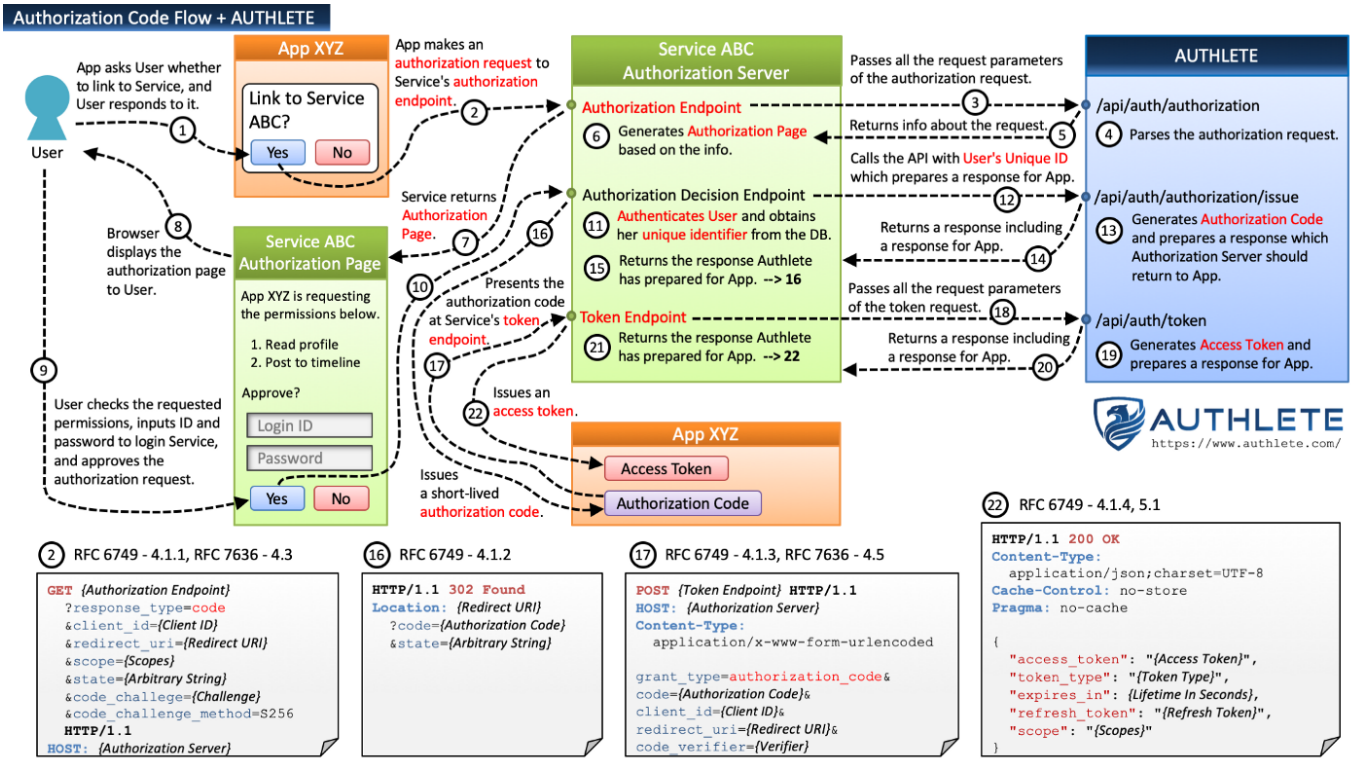
"**Semi-hosted service pattern**" is a new architecture of OAuth 2.0 and OpenID Connect implementation. In the pattern, a frontend server (an authorization server and an OpenID provider) utilizes a backend service which provides APIs to help the frontend server implement OAuth 2.0 and OpenID Connect. **Authlete** is a real-world example of such backend services. The figure below illustrates the relationship between a frontend server and a backend service (Authlete).



Semi-Hosted Service Pattern

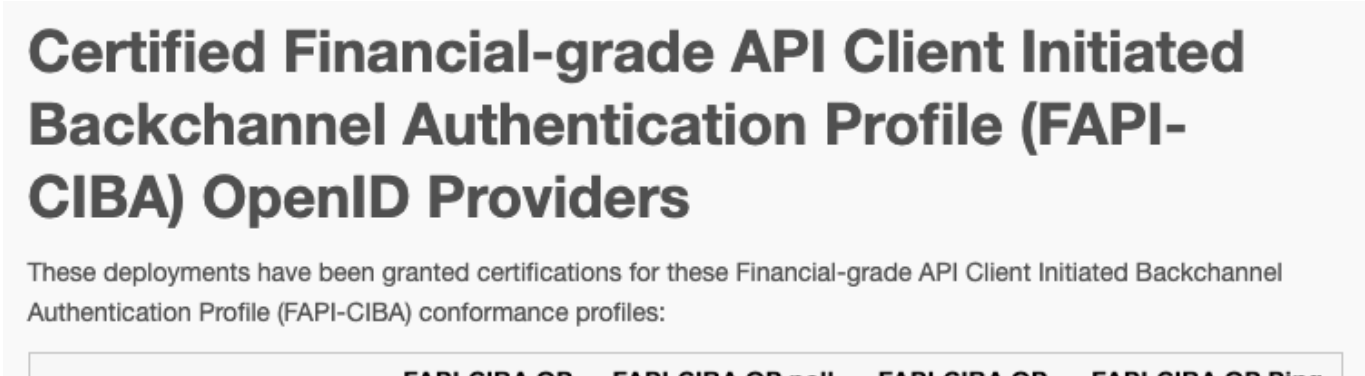
The primary advantage of this architecture is in that the core part of OAuth 2.0 and OpenID Connect implementation is clearly separated from other technical components such as identity management, user authentication, login session management, API management and fraud detection.

For example, the following diagram illustrates how user authentication is separated from OAuth 2.0 implementation. Please read *"**New Architecture of OAuth 2.0 and OpenID Connect Implementation**"* for details about the semi-hosted service pattern and its architectural advantages.



Authorization Code Flow + Authlete

Authlete is a certified implementation. Especially, it supports FAPI (Financial-grade API) and CIBA (Client Initiated Backchannel Authentication). As of October 2019, Authlete is the only certified implementation in the world that supports the FAPI-CIBA profile. Please check **Authlete**.



Organization	Implementation	FAPI-CIBA OP poll w/ MTLS	FAPI-CIBA OP poll w/ Private Key	FAPI-CIBA OP Ping w/ MTLS	FAPI-CIBA OP Ping w/ Private Key
Authlete	Authlete 2.1	6-Sep-2019 [view]	6-Sep-2019 [view]	6-Sep-2019 [view]	6-Sep-2019 [view]

Certified FAPI-CIBA OpenID Providers as of October 2019

DISCLAIMER: I'm a co-founder of Authlete, Inc.

Oauth