OAuth 2.0 Client Authentication

darutk.medium.com/oauth-2-0-client-authentication-4b5f929305d4

18 July 2019

This article explains "**OAuth 2.0 client authentication**".

In addition to the client authentication methods described in RFC 6749, this article explains methods that utilize a client assertion and a client certificate.

1. Client Authentication Methods

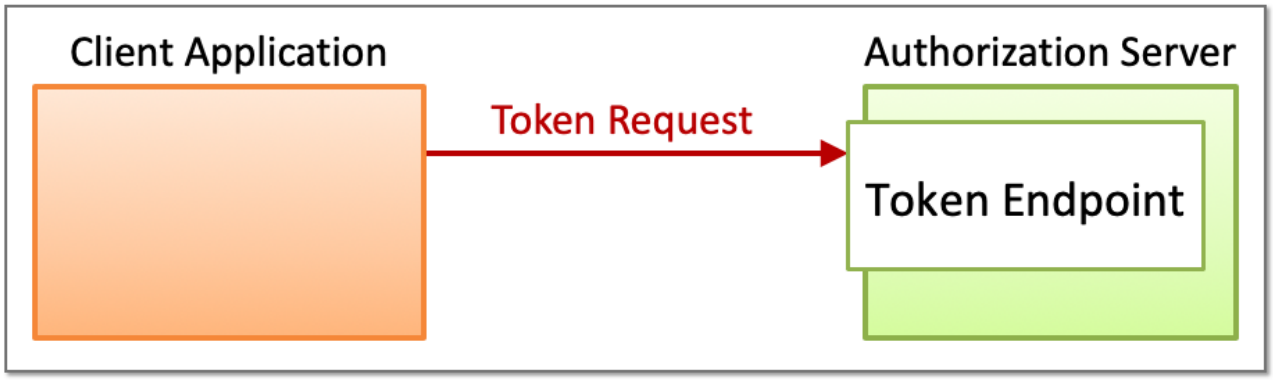
1.1. Token Endpoint

There is an **authorization server**.

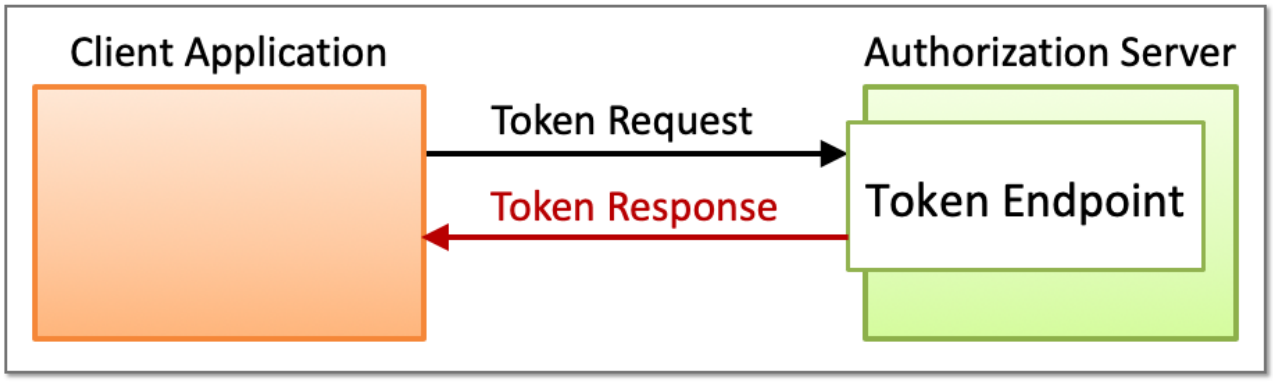
There is a **client application** that wants to get an **access token** from the authorization server.

In most cases, access tokens are issued from a **token endpoint**. Therefore, the authorization server prepares a token endpoint.

The client application sends a **token request** to the token endpoint to get an access token.

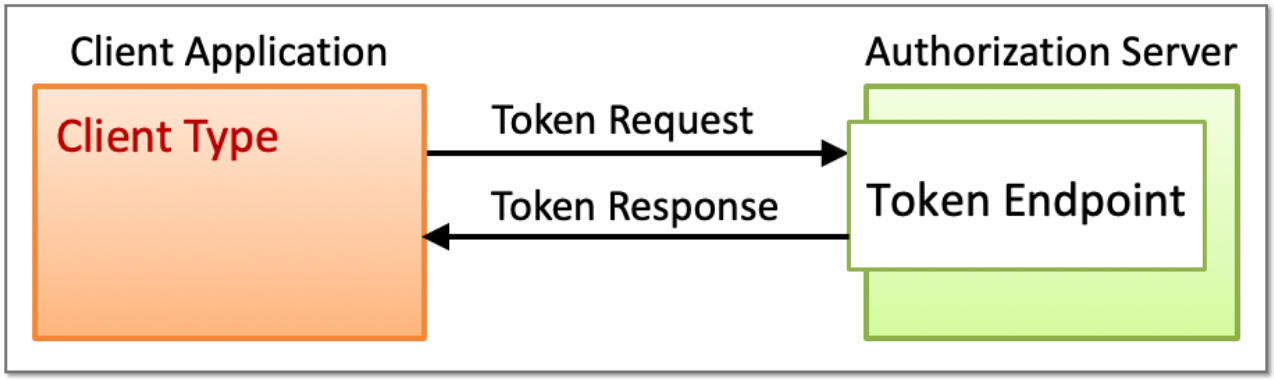


The authorization server returns a **token response**. The response contains an access token.

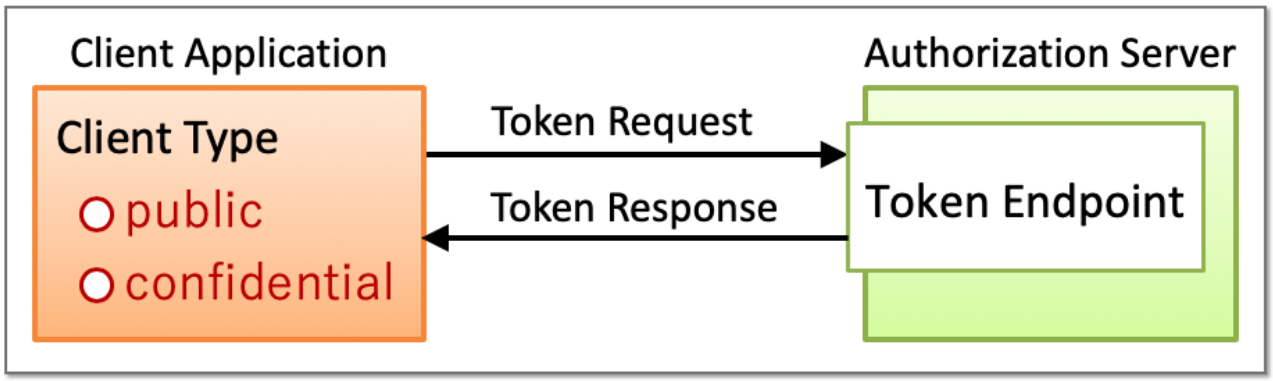


1.2. Client Type

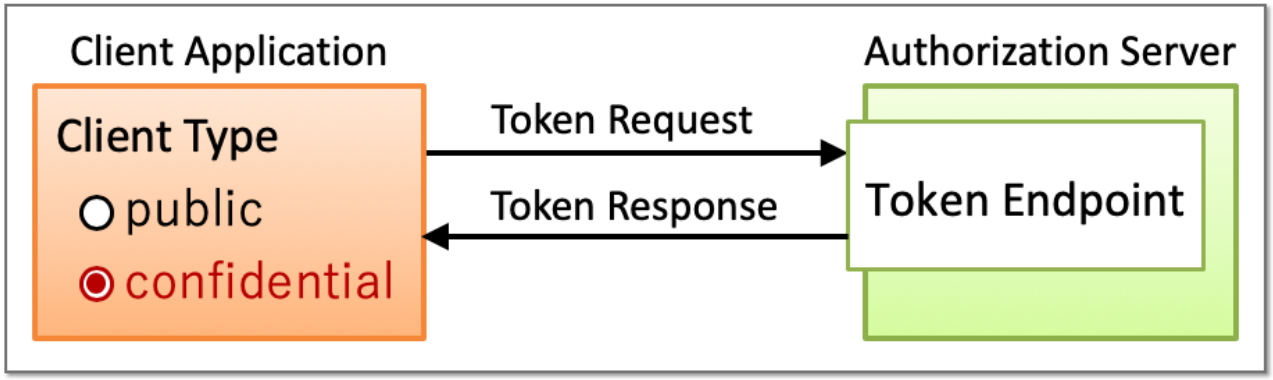
A client application has an attribute named **client type** (RFC 6749, 2.1. Client Types).



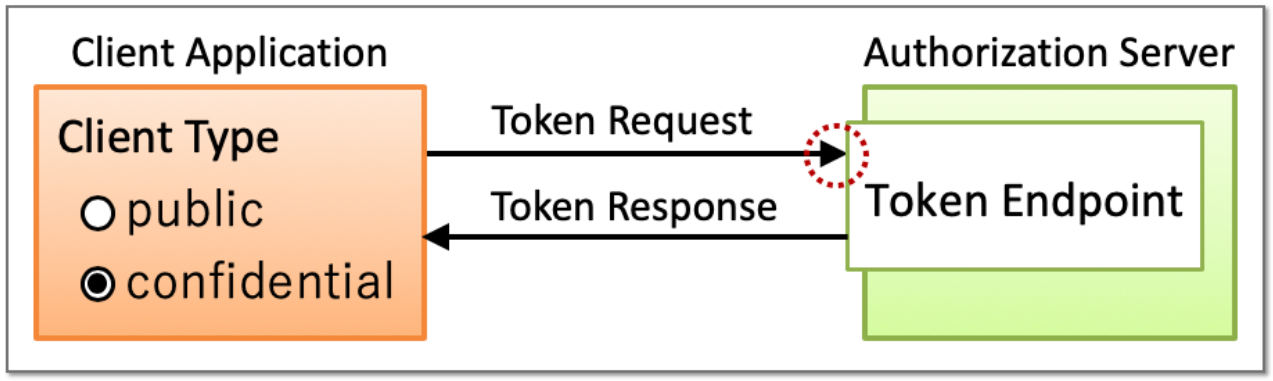
The value of the attribute is either **public** or **confidential**.



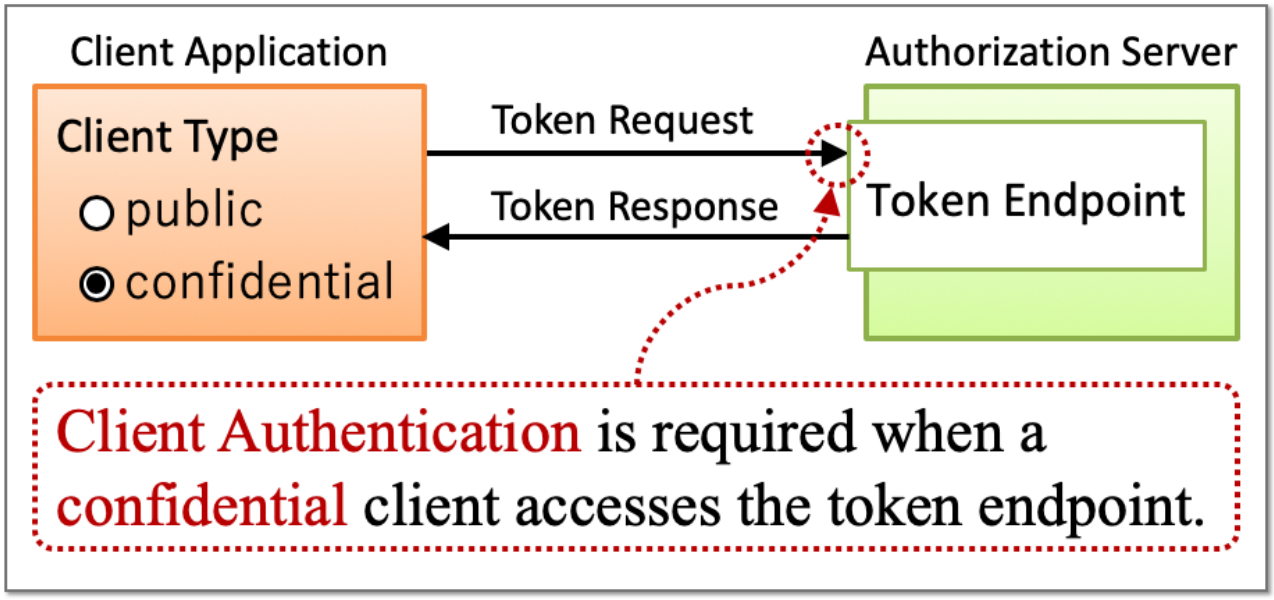
If the client type is confidential,



when the client application accesses the token endpoint,



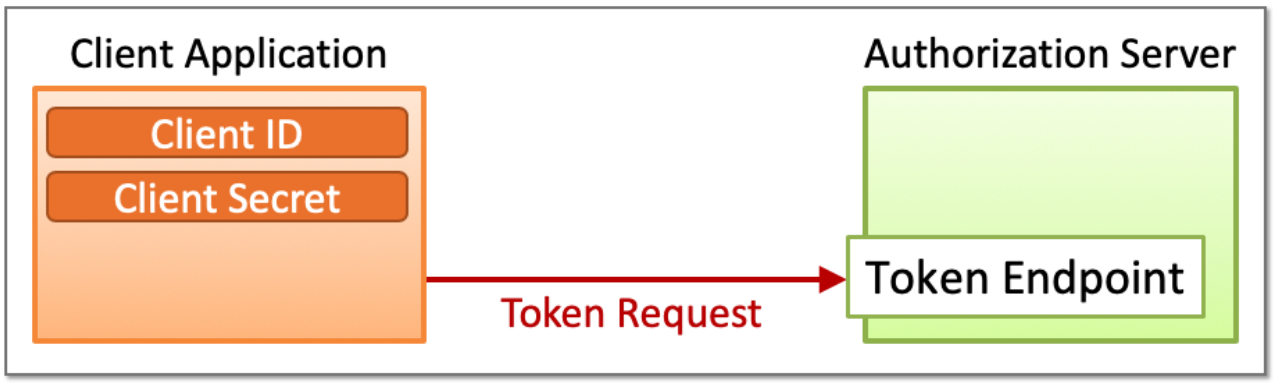
client authentication is required.



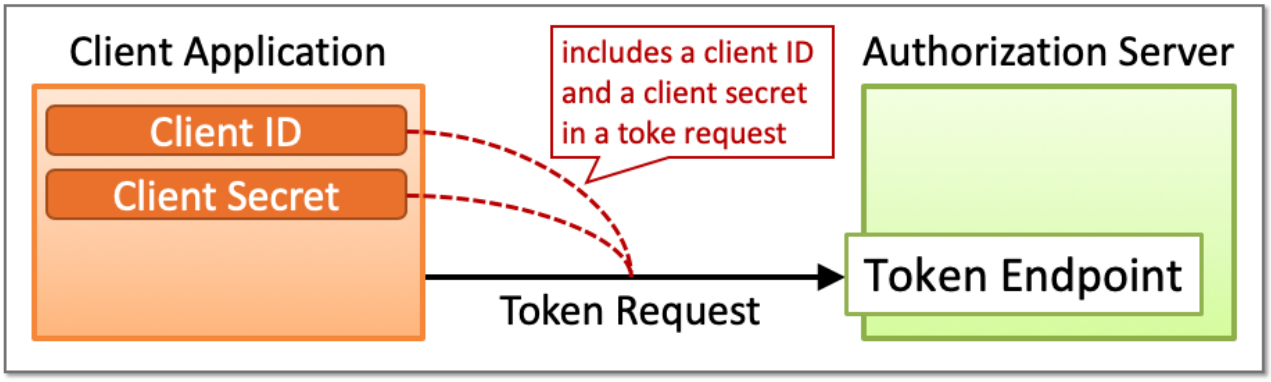
1.3. client_secret_post

In traditional ways, to begin with, an authorization server generates a pair of **client ID** and **client secret** and gives the pair to a client application in advance.

When the client application sends a token request,



it includes the client ID and the client secret in the request. The authorization server checks whether the pair is valid.



The simplest way to include a client ID and a client secret in a token request is to use the `client_id` and `client_secret` request parameters.

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

client_id={Client ID}&
client_secret={Client Secret}&
(abbrev)
```

This client authentication method has a name, `client_secret_post` (OIDC Core, 9. Client Authentication).

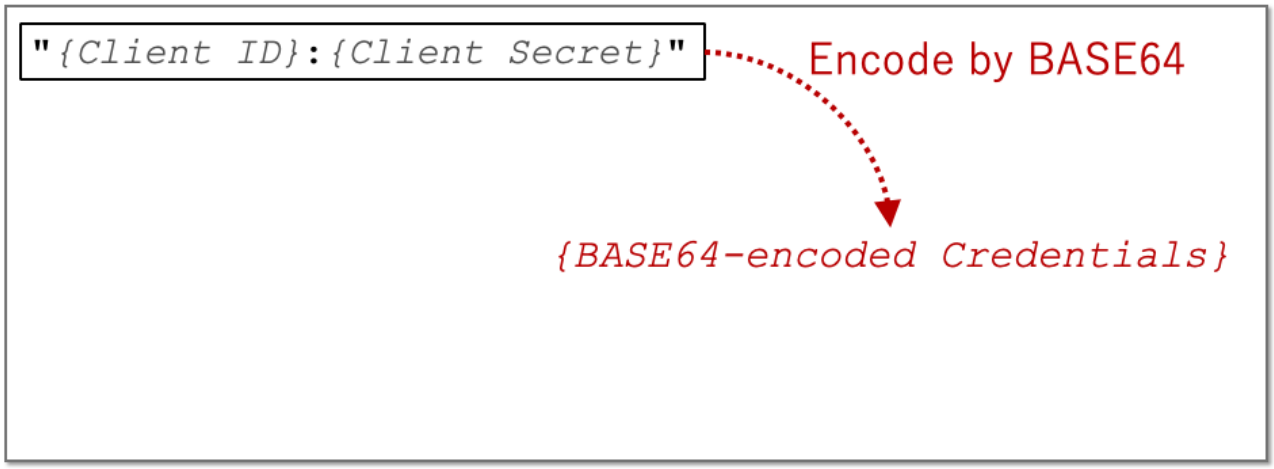
1.4. client_secret_basic

Another way to include a client ID and a client secret in a token request is to use Basic Authentication (RFC 7617).

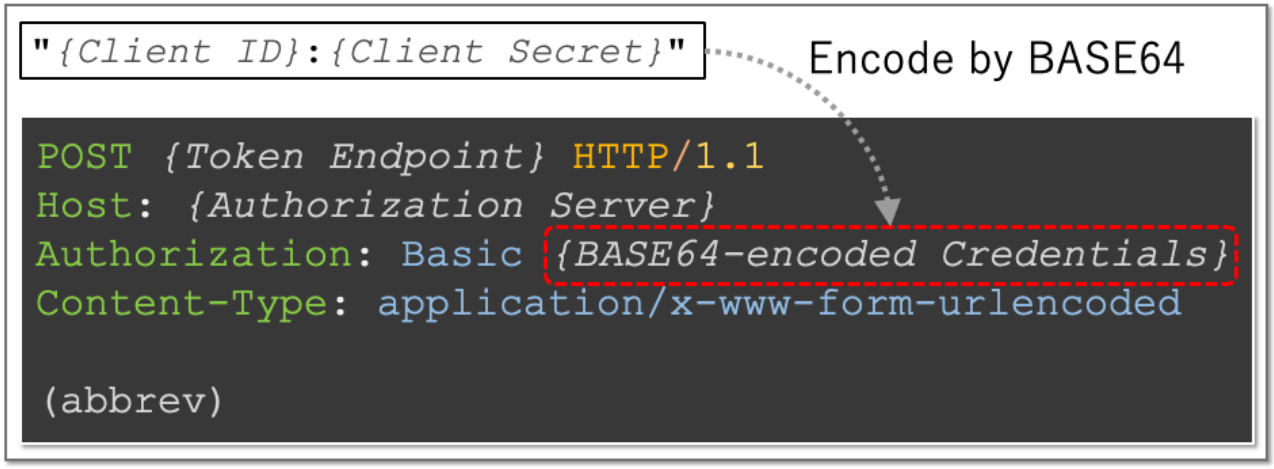
To use this method, first, build a string by concatenating a client ID, a colon and a client secret.

```
"{Client ID}:{Client Secret}"
```

Next, encode the string with BASE64 (RFC 4648).



Finally, embed the BASE64 string in the `Authorization` header in a token request.



This client authentication method has a name, `client_secret_basic` (OIDC Core, 9. Client Authentication).

`client_secret_post` and `client_secret_basic` are client authentication methods described in RFC 6749, 2.3.1. Client Password.

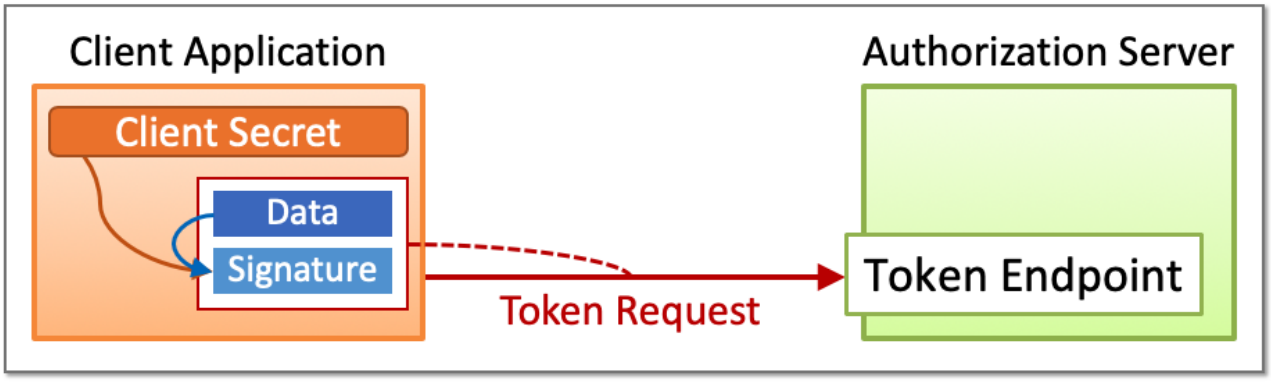
1.5. client_secret_jwt

There is an indirect way to prove that a client application has a client secret without including the client secret directly in a token request.

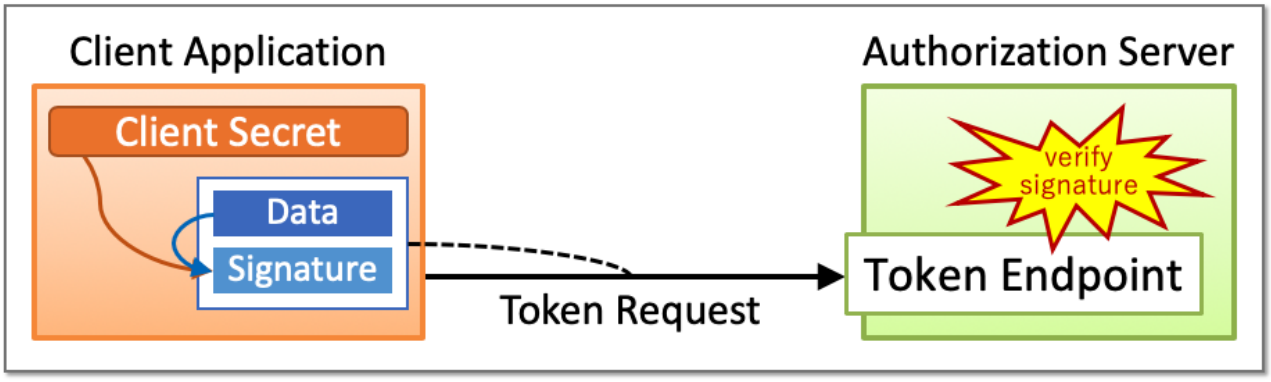
First, prepare some data.

Next, generate a signature for the data using a client secret.

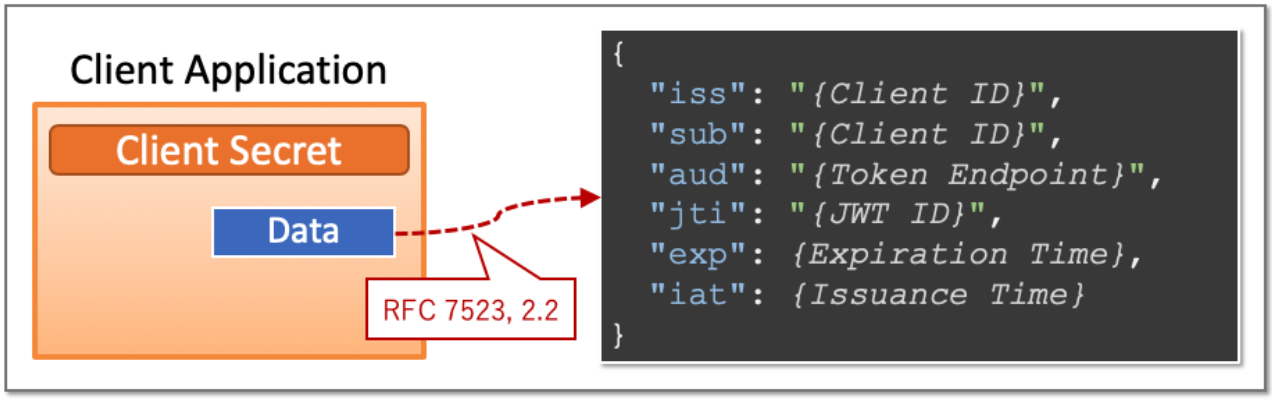
Then, include the data and the signature in a token request.



By verifying the signature, the authorization server can confirm that the client application which has sent the token request has the client secret, whereby the authorization server can authenticate the client.

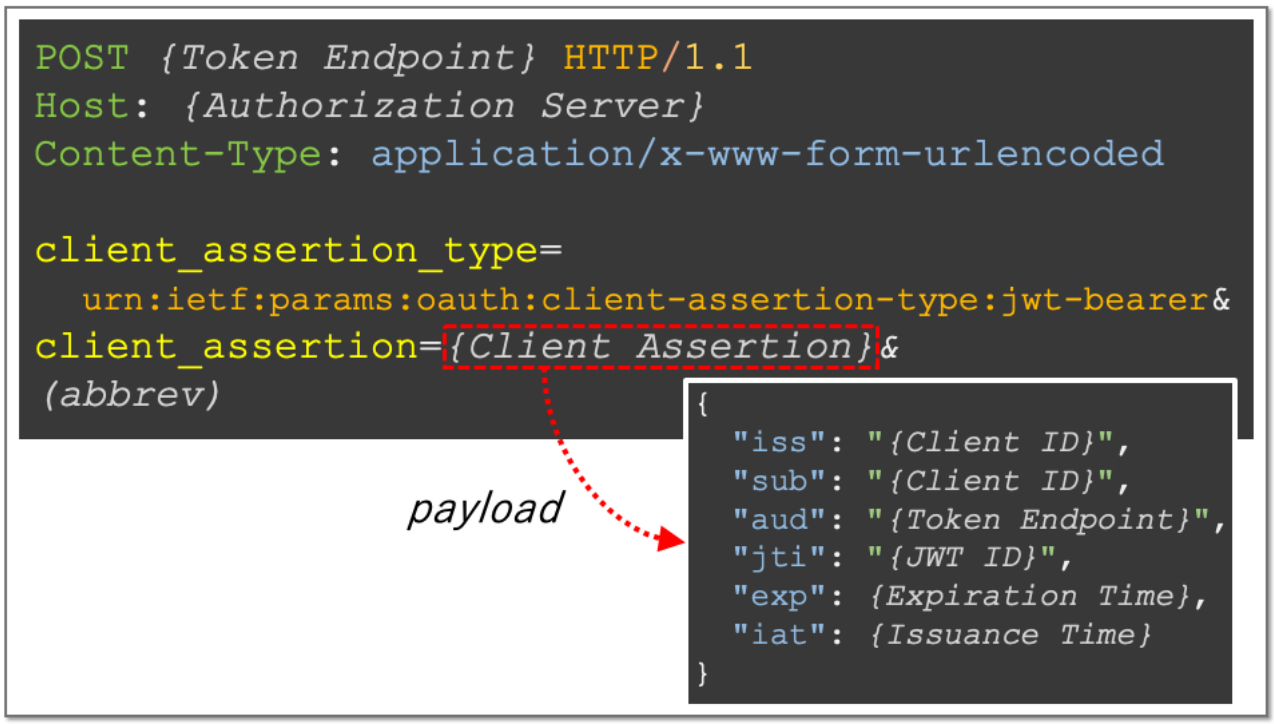


There is a rule for the format of the data. Details are written in RFC 7523, 2.2. Using JWTs for Client Authentication. To put it simply, it is JSON that includes an `iss` and other claims.



Signing this JSON is conducted by the way defined in RFC 7515 (JSON Web Signature). As a result of the signing, a JWT (RFC 7519) is generated. In the context of client authentication, the JWT is called **client assertion**.

The client assertion is included in a token request as the value of the `client_assertion` request parameter. At the same time, the `client_assertion_type` request parameter needs to be included. The value of the request parameter is a fixed string, `urn:ietf:params:oauth:client-assertion-type:jwt-bearer`.



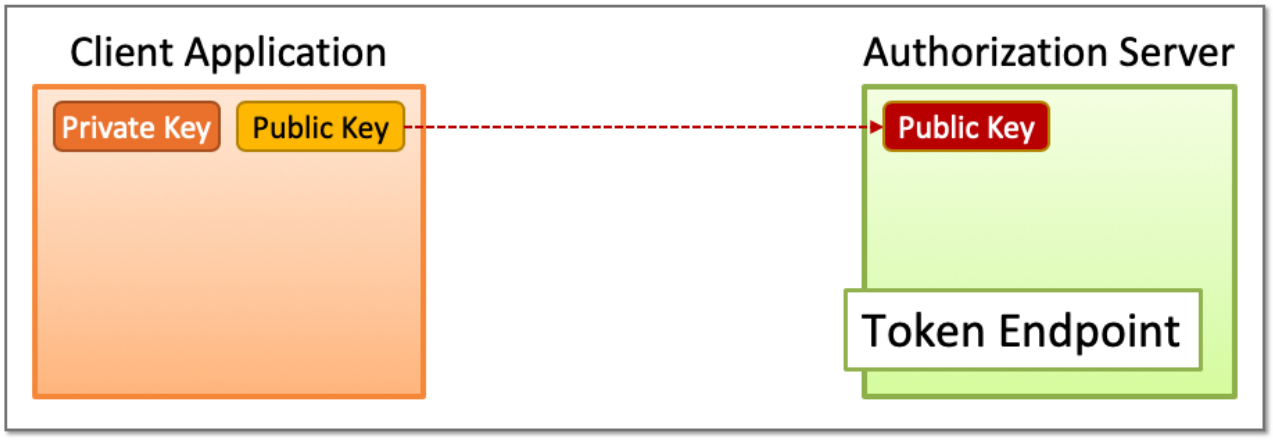
This client authentication method has a name, `client_secret_jwt` (OIDC Core, 9. Client Authentication).

1.6. private_key_jwt

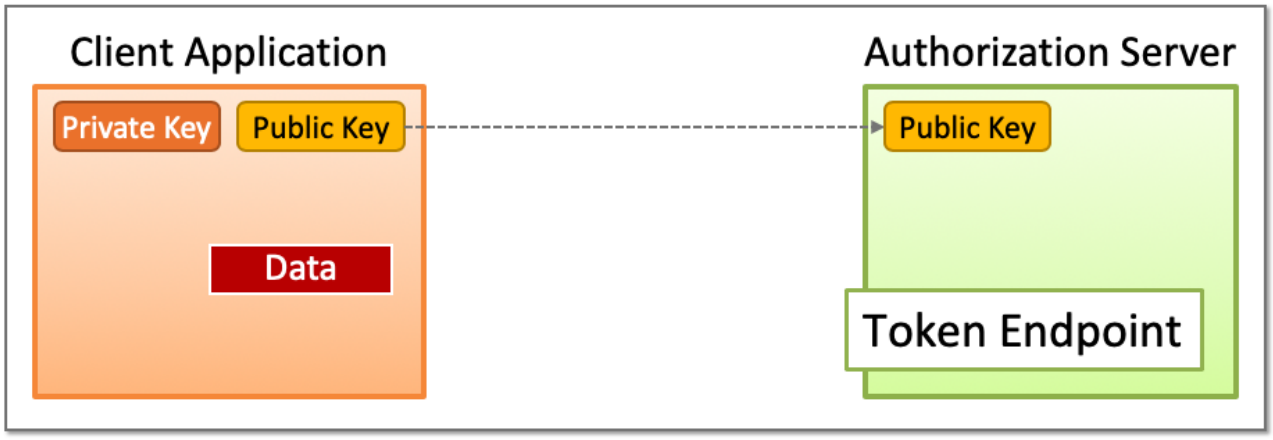
In the client authentication method explained in the previous section, the signature of the client assertion is generated using a shared key (i.e. client secret). On the other hand, there is another way which uses an asymmetric key.

First, prepare a pair of a private key and a public key on the client side.

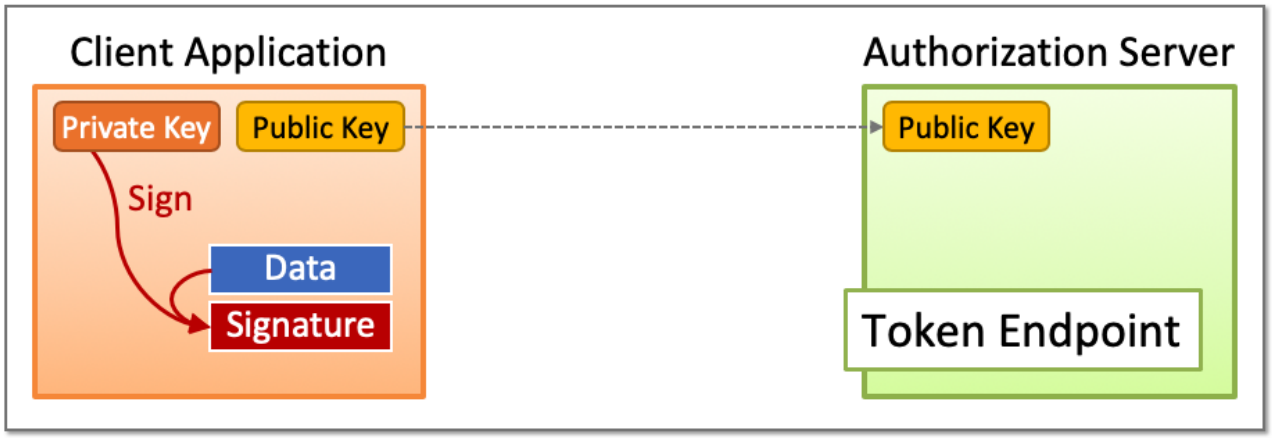
Then, make the public key accessible from the authorization server in some way or other (e.g. by `jwks` or `jwks_uri` client metadata).



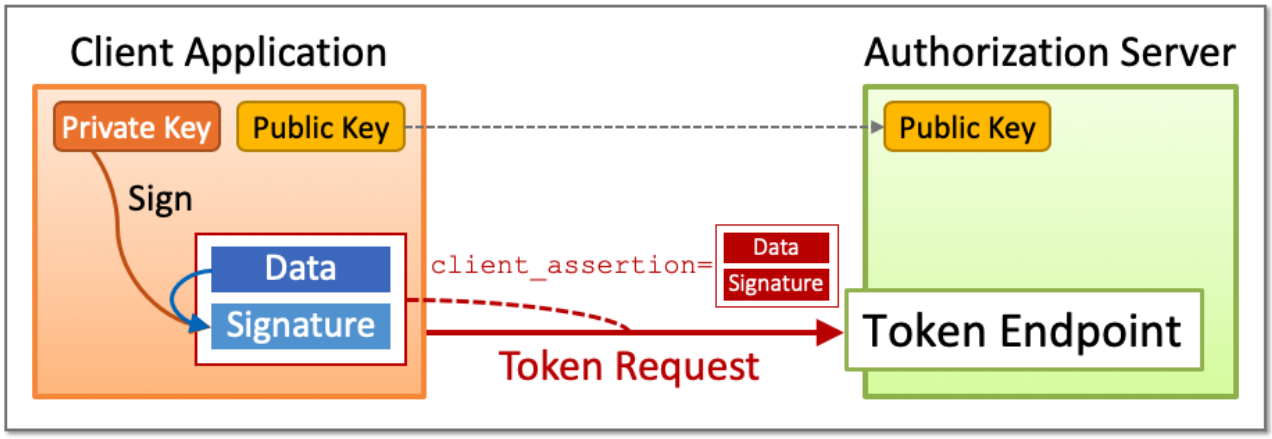
Prior to a token request, prepare JSON data which conforms to the specification described in RFC 7523, 2.2. Using JWTs for Client Authentication.



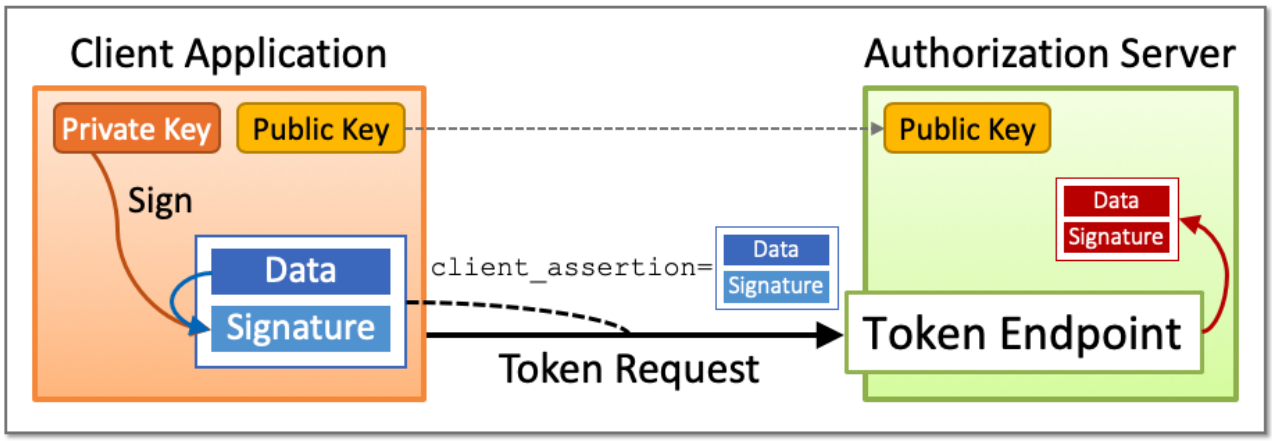
Sign the data with the private key.



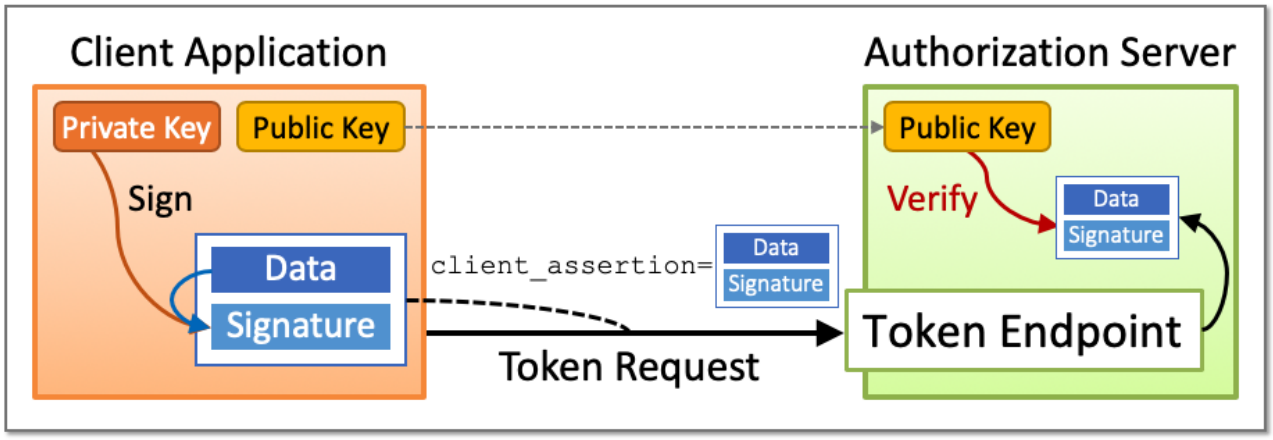
Make a token request including the generated client assertion as the value of the `client_assertion` request parameter.



The authorization server extracts the client assertion from the token request.



Then, it verifies the client assertion with the public key.

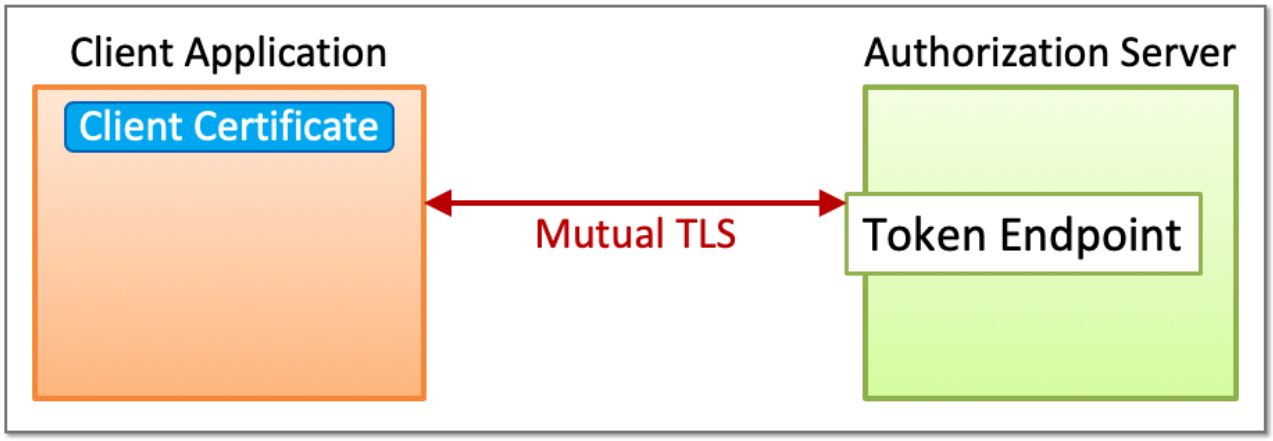


This client authentication method has a name, `private_key_jwt` (OIDC Core, 9. Client Authentication).

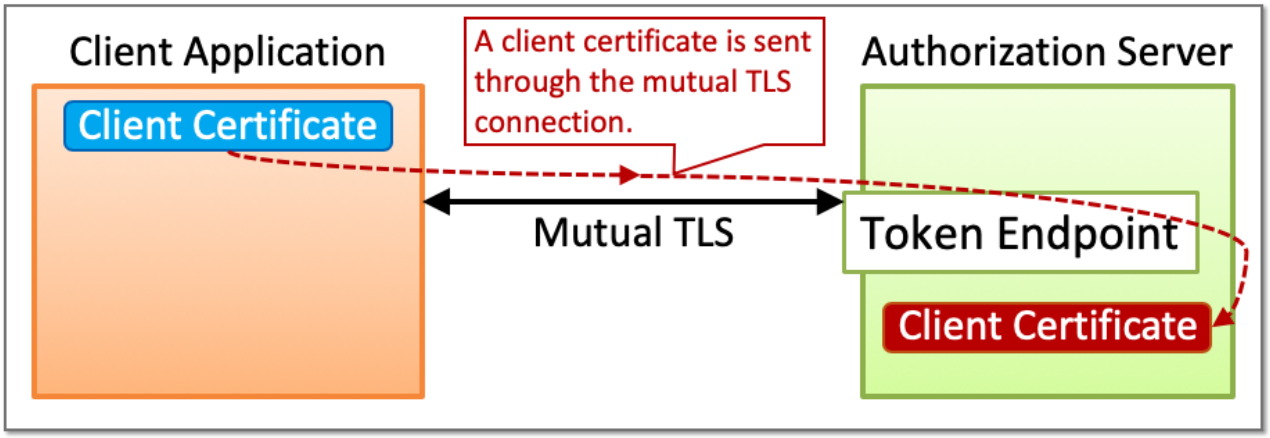
1.7. tls_client_auth

There is a specification titled "OAuth 2.0 Mutual TLS Client Authentication and Certificate-Bound Access Tokens" (hereinafter, MTLS). "2. Mutual TLS for OAuth Client Authentication" of the specification defines client authentication methods which utilize a client certificate.

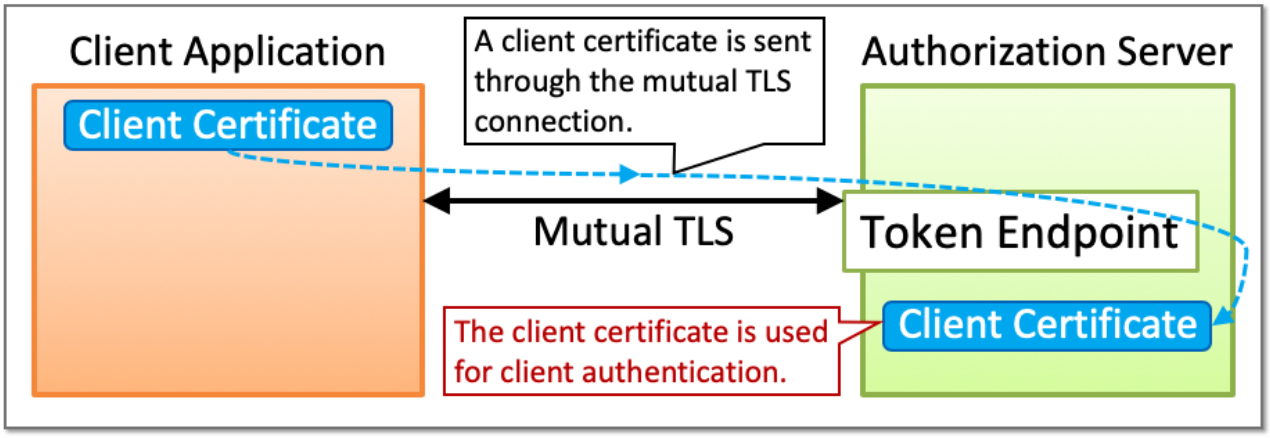
To use the client authentication methods defined in MTLS, the connection between a client application and a token endpoint must be Mutual TLS.



In a normal TLS connection, only the server presents its certificate. On the other hand, in a mutual TLS connection, the client presents its certificate, too. As a result, the server receives a client certificate.



The authorization server uses the client certificate for client authentication.



To use a PKI certificate in this client authentication method, a client must register information which identifies the subject of the certificate into the authorization server in advance. Information to identify the subject is one of the following.

Subject Information	Client Metadata
Subject Distinguished Name	`tls_client_auth_subject_dn`
Subject Alternative Name, DNS	`tls_client_auth_san_dns`
Subject Alternative Name, URI	`tls_client_auth_san_uri`
Subject Alternative Name, IP address	`tls_client_auth_san_ip`
Subject Alternative Name, Email	`tls_client_auth_san_email`

The authorization server confirms that the subject of the client certificate sent from a client matches the one registered in advance, whereby the authorization server can authenticate the client.

This client authentication method has a name, `tls_client_auth` (MTLS, 2.1.1. PKI Method Metadata Value).

Because a client certificate does not include a client ID for the OAuth 2.0 context, the client cannot be identified only by the client certificate. Therefore, when a certificate-based client authentication method is used, a client ID needs to be included in the request. Basically, the `client_id` request parameter is used for the purpose.

1.8. self_signed_tls_client_auth

Instead of a PKI certificate, a self-signed certificate also can be used for certificate-based client authentication. To use a self-signed certificate, a client must register the certificate into the server in advance.

This client authentication method has a name, `self_signed_tls_client_auth` (MTLS, 2.2.1. Self-Signed Method Metadata Value).

2. Metadata

The following are metadata related to client authentication.

2.1. Server Metadata

2.1.1. token_endpoint_auth_methods_supported

Client authentication methods supported at the token endpoint. An array containing names of client authentication methods such as `client_secret_basic` and `private_key_jwt` .

2.1.2. token_endpoint_auth_signing_alg_values_supported

Supported signature algorithms for client assertion for client authentication at the token endpoint. An array containing values listed in RFC 7518 (JSON Web Algorithms), 3.1. “alg” (Algorithm) Header Parameter Values for JWS, such as `HS256` and `ES256` .

alg	Algorithm
HS256	HMAC using SHA-256
HS384	HMAC using SHA-384
HS512	HMAC using SHA-512
RS256	RSASSA-PKCS1-v1_5 using SHA-256
RS384	RSASSA-PKCS1-v1_5 using SHA-384
RS512	RSASSA-PKCS1-v1_5 using SHA-512
ES256	ECDSA using P-256 and SHA-256
ES384	ECDSA using P-384 and SHA-384
ES512	ECDSA using P-521 and SHA-512
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512
none	No digital signature or MAC performed

JWS signature algorithms listed in RFC 7518

2.1.3. revocation_endpoint_auth_methods_supported

Client authentication methods supported at the revocation endpoint (RFC 7009). The content is the same as that for `token_endpoint_auth_methods_supported`.

2.1.4. revocation_endpoint_auth_signing_alg_values_supported

Supported signature algorithms for client assertion for client authentication at the revocation endpoint (RFC 7009). The content is the same as that for `token_endpoint_auth_signing_alg_values_supported`.

2.1.5. introspection_endpoint_auth_methods_supported

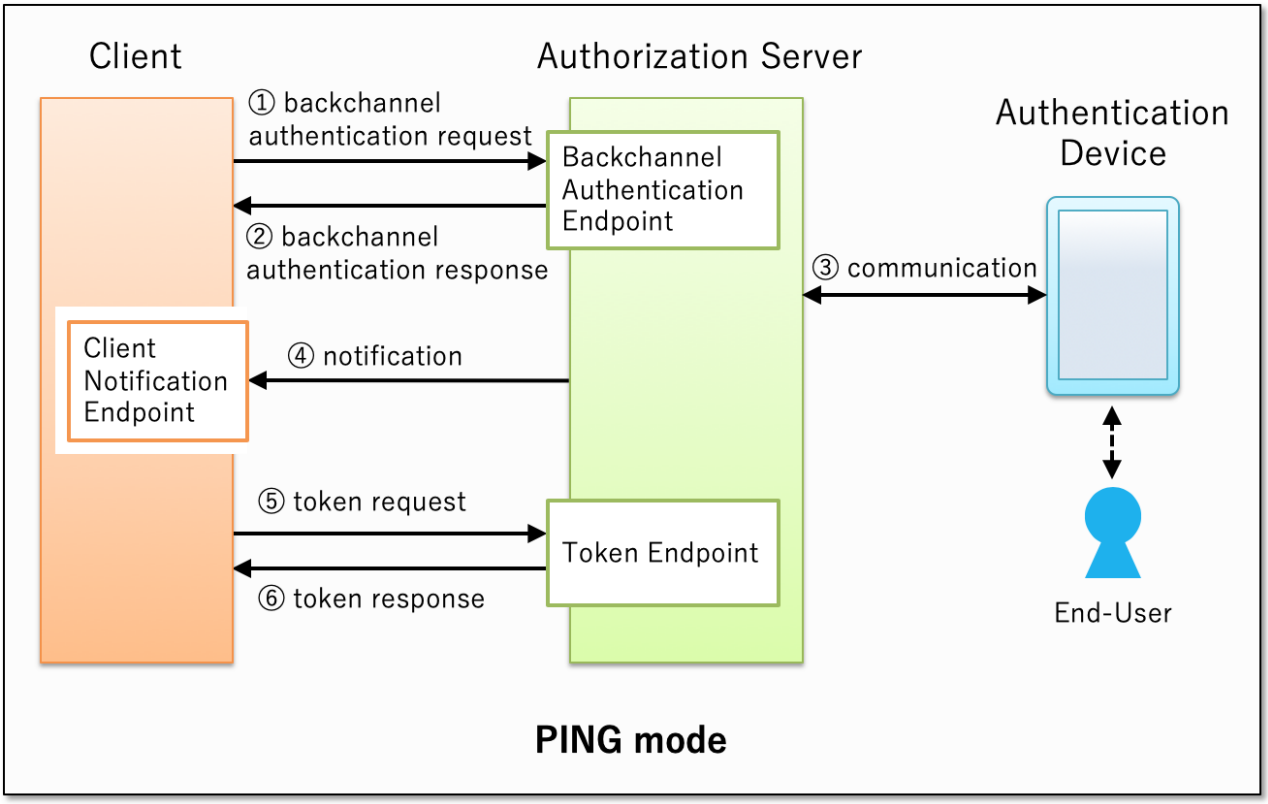
Client authentication methods supported at the introspection endpoint (RFC 7662). The content is the same as that for `token_endpoint_auth_methods_supported`.

2.1.6. introspection_endpoint_auth_signing_alg_values_supported

Supported signature algorithms for client assertion for client authentication at the introspection endpoint (RFC 7662). The content is the same as that for `token_endpoint_auth_signing_alg_values_supported`.

2.1.7 Metadata related to Backchannel Authentication Endpoint

The **backchannel authentication endpoint** defined in CIBA Core accepts requests from confidential clients only. Therefore, client authentication is always required when a client accesses the endpoint.



cf. CIBA Flow in Ping Mode

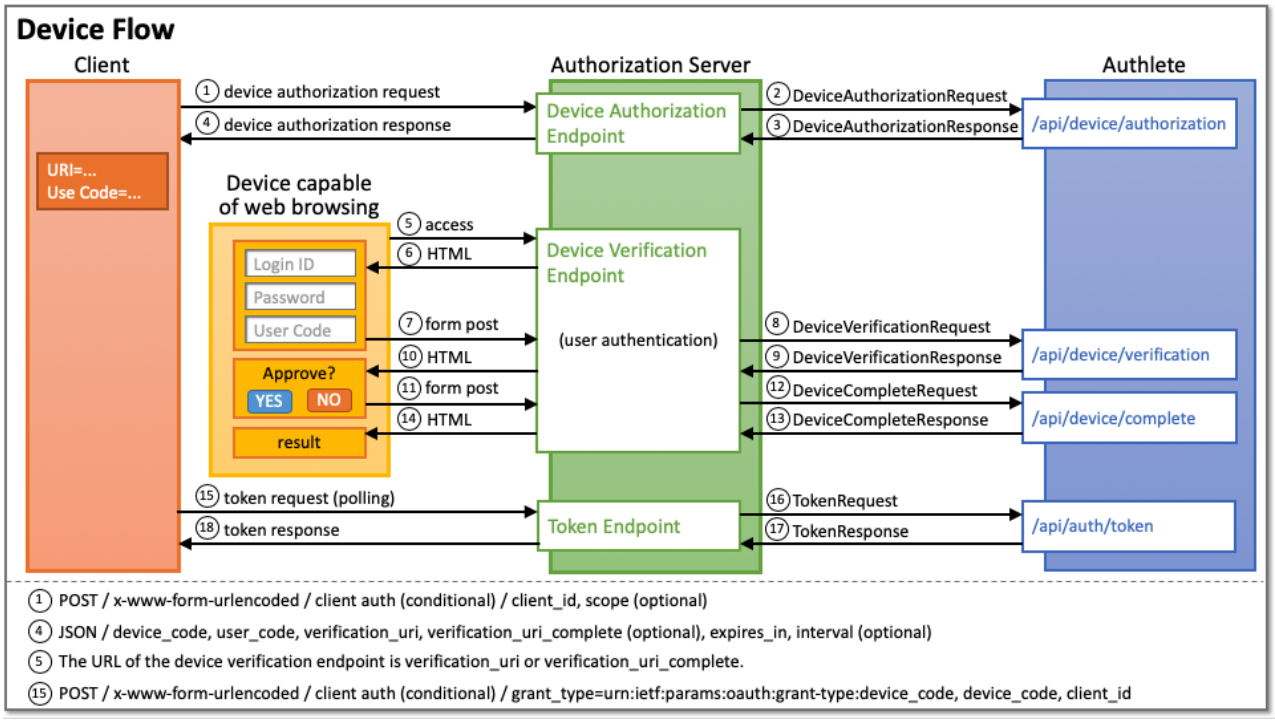
In RFC 8141 which lists metadata of an authorization server, metadata for supported client authentication methods (`*_endpoint_auth_methods_supported`) and supported signature algorithms for client assertion (`*_endpoint_auth_signing_alg_values_supported`) are defined for each endpoint such as a token endpoint, a revocation endpoint and an introspection endpoint. Therefore, during the discussion period to develop the CIBA Core specification, I suggested adding new metadata for the backchannel authentication endpoint ([Issue 102] CIBA: Metadata for client auth at backchannel endpoint).

However, the conclusion was that new metadata for the backchannel authentication endpoint should not be defined and the existing metadata for the token endpoint should be used for the backchannel authentication endpoint, too. The background opinion for this decision is that the old custom which has defined metadata for each endpoint is not good and should not be followed any longer. If you are interested in the discussion, please read "3.2. Client Authentication" in "**CIBA, a new authentication/authorization technology in 2019, explained by an implementer**".

2.1.8. Metadata related to Device Authorization Endpoint

OAuth 2.0 Device Authorization Grant (a.k.a. Device Flow), which will become an RFC soon, defines a new endpoint, **device authorization endpoint**. When a confidential client accesses the endpoint, client authentication is required as required at other

endpoints.



cf. Device Flow with Authlete APIs

The specification does not mention metadata related to client authentication at the device authorization endpoint. So, I posted a question to the OAuth Working Group mailing list to ask whether the working group had a plan to define a rule as to which client authentication method should be used at the device authorization endpoint ([OAUTH-WG] Client Authentication Method at Device Authorization Endpoint).

However, no discussion has been observed so far. It seems there are few people who care about details of client authentication method at the device authorization endpoint. Therefore, the implementation of Authlete decided to use the metadata for the token endpoint as the CIBA specification does.

2.2. Client Metadata

2.2.1. token_endpoint_auth_method

Client authentication method that a client has declared it will use at the token endpoint. For example, `client_secret_basic` and `private_key_jwt`. If the client type is public, client authentication is not required. In this case, `none` should be set.

2.2.2. token_endpoint_auth_signing_alg

Signature algorithm of client assertion that a client has declared it will use for client authentication at the token endpoint. One of the algorithms listed in 3.1. "alg" (Algorithm) Header Parameter Values for JWS of RFC 7519 (JSON Web Algorithms), such as `HS256` and `ES256`.

When the client authentication method is `client_secret_jwt`, the signature algorithm must be symmetric. To be concrete, it must be one of `HS256`, `HS384` and `HS512`. Note that from a security perspective, it is meaningless to choose an algorithm whose entropy is bigger than the entropy of the client secret. The following is the description in "16.19. Symmetric Key Entropy" in OIDC Core.

> In Section 10.1 and Section 10.2, keys are derived from the `client_secret` value. Thus, when used with symmetric signing or encryption operations, `client_secret` values MUST contain sufficient entropy to generate cryptographically strong keys. Also, `client_secret` values MUST also contain at least the minimum of number of octets required for MAC keys for the particular algorithm used. So for instance, for `HS256`, the `client_secret` value MUST contain at least 32 octets (and almost certainly SHOULD contain more, since `client_secret` values are likely to use a restricted alphabet).

On the other hand, when the client authentication method is `private_key_jwt`, the signature algorithm must be asymmetric. For instance, `ES256`.

3. Financial-grade API Requirements

3.1. Client Authentication Method

Financial-grade API (FAPI) requires higher security than traditional OAuth 2.0 and OpenID Connect. The specification puts restrictions on client authentication methods.

First, the traditional client authentication methods written in RFC 6749 (`client_secret_basic` and `client_secret_post`) are prohibited.

Client authentication methods permitted by FAPI Part 1, the security profile for Read-Only API, are the following four.

- `client_secret_jwt`
- `private_key_jwt`
- `tls_client_auth`
- `self_signed_tls_client_auth`

Client authentication methods permitted by FAPI Part 2 are as follows. Note that `client_secret_jwt` is excluded.

- `private_key_jwt`
- `tls_client_auth`
- `self_signed_tls_client_auth`

The summary table is as follows.

Client Authentication Method	Part 1	Part 2
client_secret_basic (traditional)	×	×
client_secret_post	×	×
client_secret_jwt (assertion-based)	○	×
private_key_jwt	○	○
tls_client_auth (certificate-based)	○	○
self_signed_tls_client_auth	○	○

3.2. Client Assertion Signature Algorithm

"8.6. JWS algorithm considerations" in FAPI Part 2 puts restrictions on signature algorithms for JWS. In the context of FAPI Read-and-Write, other algorithms than `ES256` and `PS256` are not permitted.

alg	Algorithm
HS256	HMAC using SHA-256
HS384	HMAC using SHA-384
HS512	HMAC using SHA-512
RS256	RSASSA-PKCS1-v1_5 using SHA-256
RS384	RSASSA-PKCS1-v1_5 using SHA-384
RS512	RSASSA-PKCS1-v1_5 using SHA-512
ES256	ECDSA using P-256 and SHA-256
ES384	ECDSA using P-384 and SHA-384
ES512	ECDSA using P-521 and SHA-512
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512
none	No digital signature or MAC performed

FAPI Part 2, 8.6. JWS algorithm considerations
1. shall use `PS256` or `ES256` algorithms

This requirement affects client authentication methods that utilize client assertion.

3.3. Key Size

The 5th clause in "5.2.2. Authorization Server" in FAPI Part 1 requires that the key size be 2048 bits or more when an RSA algorithm is used for client authentication. Likewise, the 6th clause requires that the key size be 160 bits or more when an elliptic curve algorithm is used for client authentication.

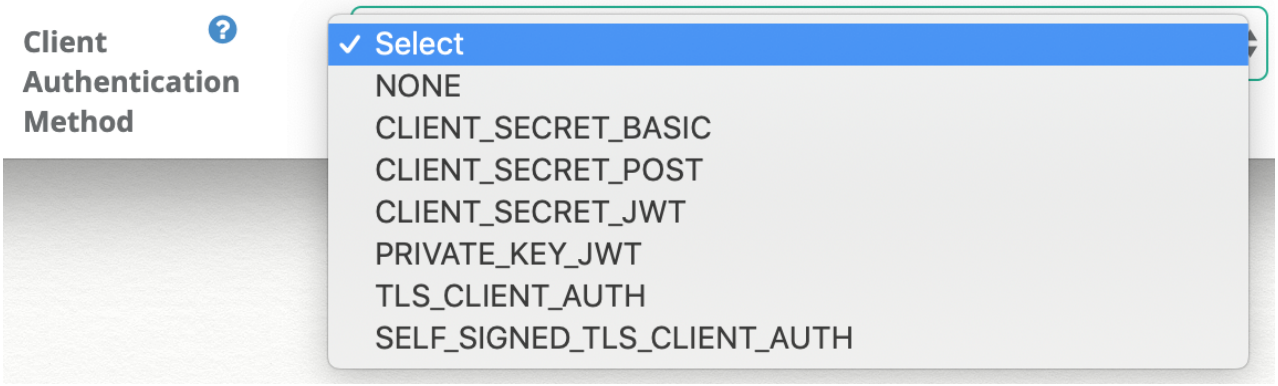
3.4. Other FAPI Requirements

If you are interested in technical details about FAPI, please read "Financial-grade API (FAPI), explained by an implementer".

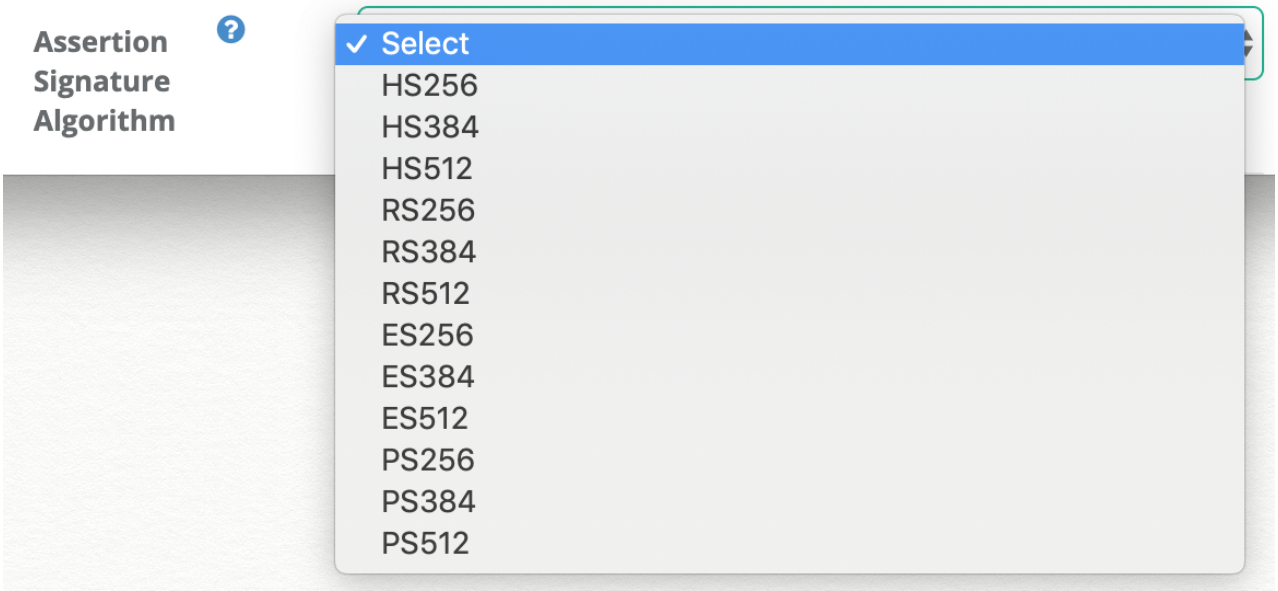
4. Authlete

Authlete, an implementation of OAuth 2.0 and OpenID Connect,

> supports all the client authentication methods explained in this article,



> supports all the client assertion signature algorithms,



> and is the only implementation in the world (as of July 18, 2019) that is (not just a sandbox but) ready for commercial deployment and has been certified by FAPI certification in both the two categories which are respectively for certificate-based client authentication (MTLS) and assertion-based client authentication (Private Key).

Certified Financial-grade API (FAPI) OpenID Providers

These deployments have been granted certifications for these Financial-grade API (FAPI) conformance profiles:

Organization	Implementation	FAPI R/W OP w/ MTLS	FAPI R/W OP w/ Private Key
Authlete	Authlete 2.1	1-Apr-2019	1-Apr-2019
ForgeRock	ForgeRock Financial 3.1.0-credence		1-Apr-2019
Ozone	Ozone Sandbox v3.1	6-Jun-2019	6-Jun-2019
Ping Identity	PingFederate 9.2.3	29-May-2019	

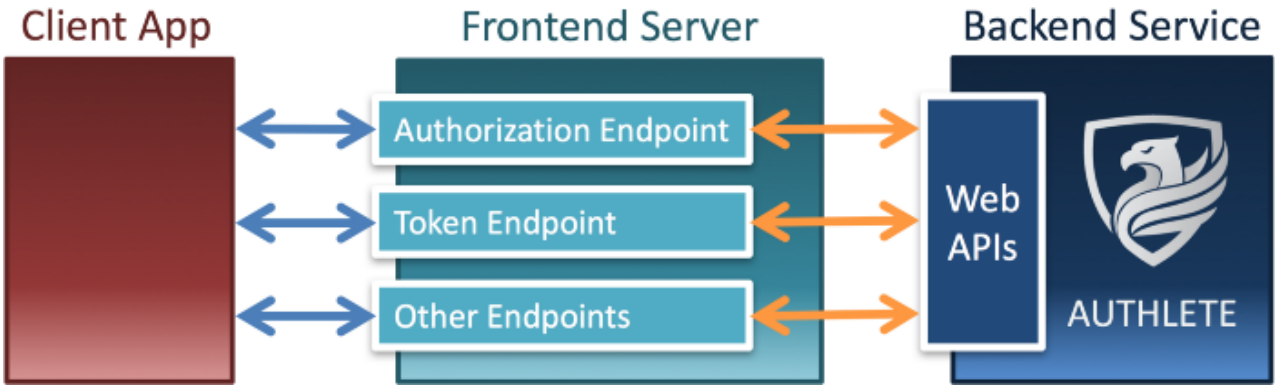
As of July 18, 2019

Finally

Recently, a lot of services publish APIs, and thanks to them, the world is becoming more convenient. On the other hand, due to improper implementations, we observe security incidents more often than before.

If you put high priority on security in publishing APIs of your services, please consider using Authlete, a certified implementation of Financial-grade API OpenID Provider.

Thanks to its architecture, Authlete can be combined with any end-user authentication mechanism, any API management solution and other API-related technologies without need to replace them.



In addition, Authlete team is eager to implement promising new specifications such as FAPI, CIBA, MTLS, JARM, etc. and thus Authlete can keep your service up-to-date.

Please contact us via the contact form or sales@authlete.com!