White Paper

A Comprehensive Commentary on Financial-grade API

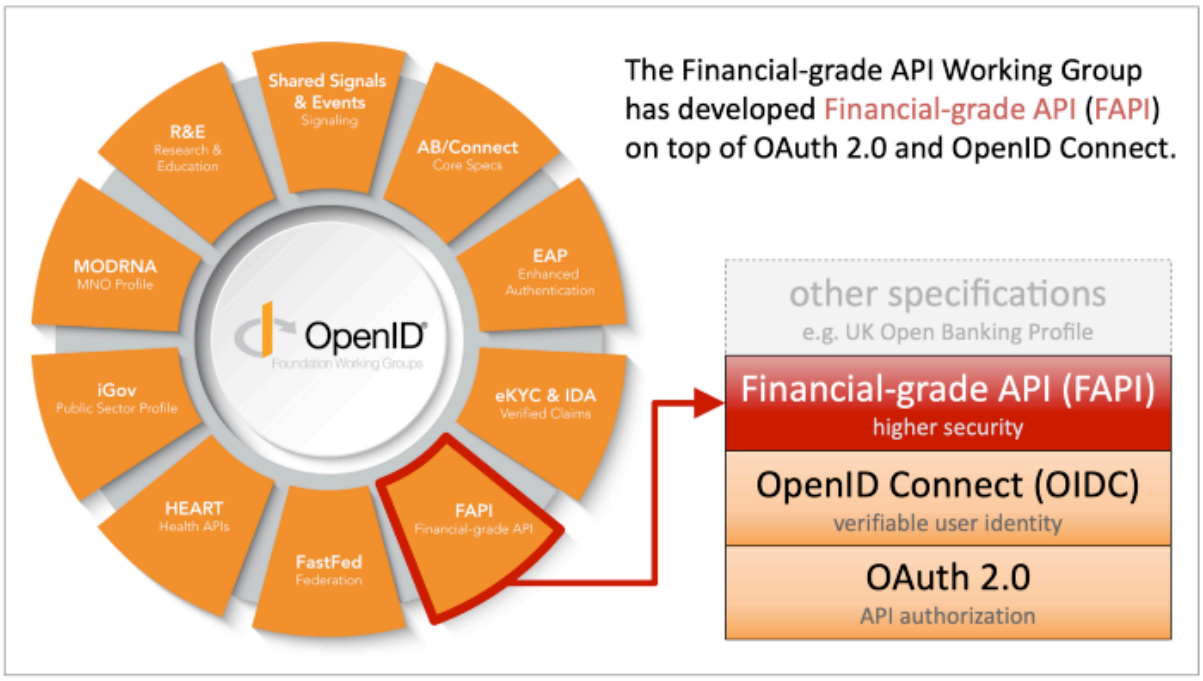
April 2021

Authlete, Inc.

Contents

What is Financial-grade API?

Financial-grade API (FAPI) is a technical specification that Financial-grade API Working Group of OpenID Foundation has developed. It uses **OAuth 2.0** and **OpenID Connect (OIDC)** as its base and defines additional technical requirements for the financial industry and other industries that require higher API security.



OpenID Foundation Working Groups and Financial-grade API Stack

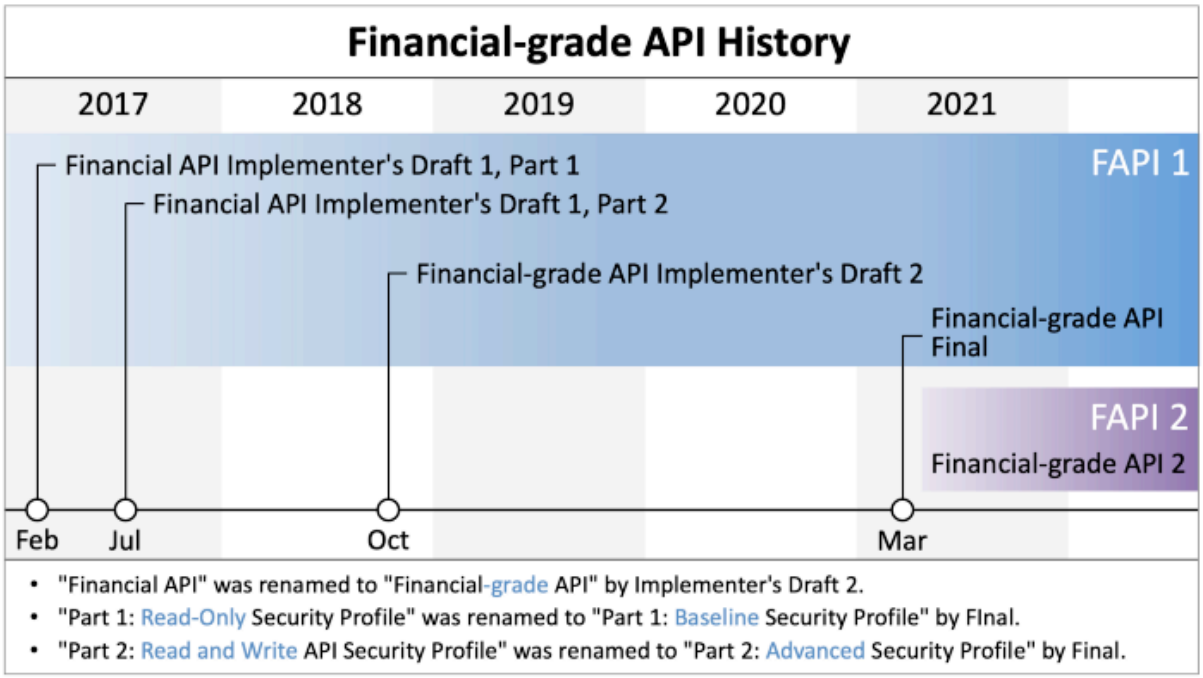
History of Standardization of FAPI

Implementer's Draft 1 — The initial version of the FAPI specification was published in 2017. The version is called Implementer's Draft 1 (ID1).

Implementer's Draft 2 — The second version was published in October, 2018. The version is called Implementer's Draft 2 (ID2). In this version, the FAPI specification was renamed from "Financial API" to "Financial-grade API" for wider adoption across various industries.

Final Version — The final version was published in March, 2021. In this version, the main two parts of the FAPI specification, "Part 1: Read-Only Security Profile" and "Part 2: Read and Write API Security Profile", were renamed to "Part 1: Baseline Security Profile" and "Part 2: Advanced Security Profile", respectively.

FAPI 2.0 — The FAPI WG has started to discuss the next version of the FAPI specification, which is called "FAPI 2.0". The FAPI FAQ published on March 31, 2021 (announcement) mentions FAPI 2.0. Authlete is mentioned in the answer to the question "Are there FAPI 2.0 implementations?" because Authlete has already implemented new technical components of FAPI 2.0 such as **PAR** (OAuth 2.0 Pushed Authorization Requests), **RAR** (OAuth 2.0 Rich Authorization Requests) and **DPoP** (OAuth 2.0 Demonstration of Proof-of-Possession at the Application Layer).



History of Financial-grade API

FAPI Specifications

The core parts of the FAPI specification are Part 1 and Part 2. Their previous and final versions are available here:

Implementer's Draft 1 (Part 1: February 2, 2017 / Part 2: July 17, 2017)

- Financial Services — Financial API — Part 1: Read Only API Security Profile
- Financial Services — Financial API — Part 2: Read and Write API Security Profile

Implementer's Draft 2 (October 17, 2018)

- Financial-grade API — Part 1: Read-Only API Security Profile
- Financial-grade API — Part 2: Read and Write API Security Profile
- Differences between ID1 and ID2

Final Version (March 12, 2021)

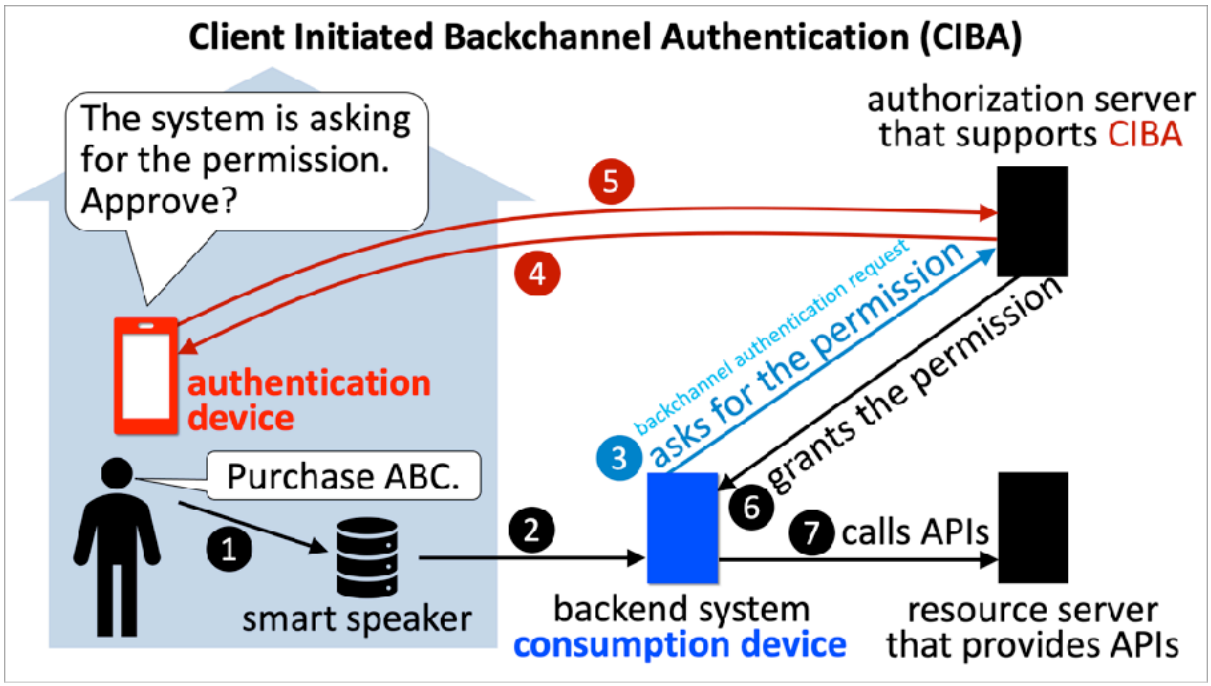
- Financial-grade API Security Profile 1.0 — Part 1: Baseline
- Financial-grade API Security Profile 1.0 — Part 2: Advanced
- Differences between ID2 and Final

In addition, another specification was released in August, 2019 that lists additional requirements applied when FAPI and **CIBA** (**Client Initiated Backchannel Authentication**) are used together. The specification is called **FAPI-CIBA Profile**.

- Financial-grade API: Client Initiated Backchannel Authentication Profile

For details about CIBA, please read the following article.

- "CIBA", a new authentication/authorization technology in 2019, explained by an implementer

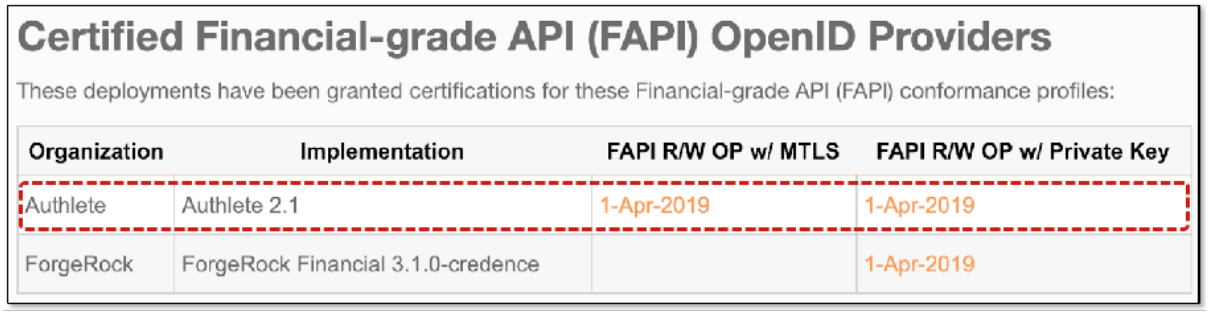


Concept of CIBA

FAPI Certification Program

Certification for FAPI OpenID Providers

The Certification Program for FAPI OpenID Providers officially started on April 1, 2019 (announcement). Two vendors were granted certification on the start day. **Authlete, Inc.**, is one of the two vendors.



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

Certified Financial-grade API OpenID Providers on April 1, 2019

Two years have passed since then, and now more than 30 solutions and deployments are listed as certified FAPI OPs.

Certification program for the FAPI Final version has not started yet as of this writing (April, 2021), but **Authlete 2.2 has already supports the FAPI Final version**. See the announcement and the release note published on February 4, 2021 for details.

Certification for FAPI-CIBA OpenID Providers

The Certification Program for FAPI-CIBA OpenID Providers started on September 16, 2019 (announcement). Authlete was the only solution that was granted certification on the start day.

Certified Financial-grade API Client Initiated Backchannel Authentication Profile (FAPI-CIBA) OpenID Providers

These deployments have been granted certifications for these Financial-grade API Client Initiated Backchannel Authentication Profile (FAPI-CIBA) conformance profiles:

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	16-Sep-2019 [view]	16-Sep-2019 [view]	16-Sep-2019 [view]	16-Sep-2019 [view]

Certified FAPI-CIBA Profile OpenID Providers on September 16, 2019

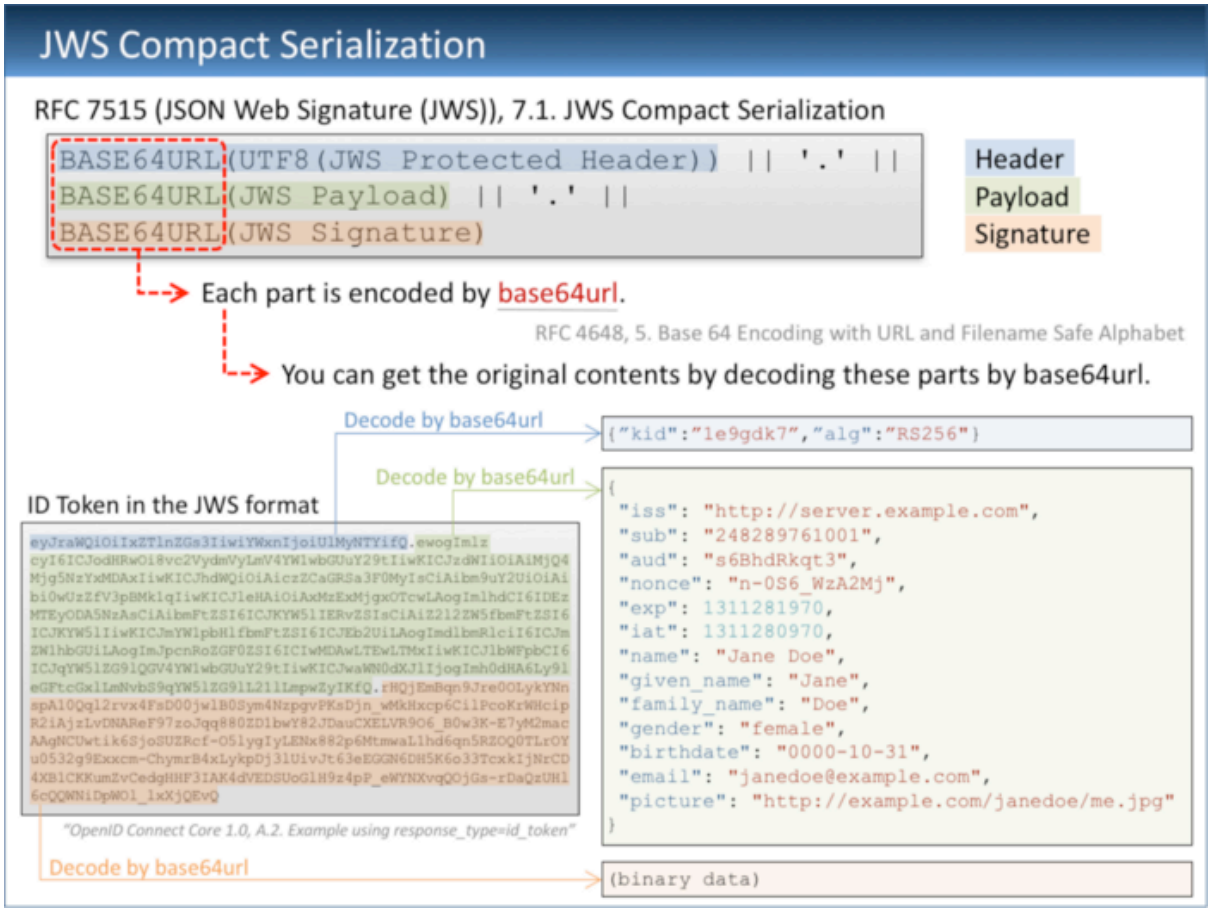
As of this writing (April, 2021), three solutions including Authlete are listed as certified FAPI-CIBA OPs.

Prior Knowledge to Understand FAPI

Basic Specifications

The format of the FAPI specification is a terse list of technical requirements, so the document is not long. In exchange, a lot of prior knowledge is required to read it smoothly. Especially, you have to learn **RFC 6749** and **RFC 6750** (the core of OAuth 2.0) and **OpenID Connect Core 1.0** (the core of OpenID Connect) by heart.

In addition, because specifications related to JWT (**JWS**, **JWE**, **JWK**, **JWA** and **JWT**) are prior knowledge to understand OIDC Core, they are of course prior knowledge to read the FAPI specification. Therefore, you need to understand them perfectly.



JWS Compact Serialization (RFC 7515 Section 7.1)

Furthermore, **PKCE** (RFC 7636) which was published in September, 2015 is now regarded as a part of the basic set of OAuth 2.0 specifications as well as RFC 6749 and RFC 6750.

The following is a list of specifications that you should read at least once before the FAPI specification.

- RFC 6749 — The OAuth 2.0 Authorization Framework
- RFC 6750 — The OAuth 2.0 Authorization Framework: Bearer Token Usage
- RFC 7515 — JSON Web Signature (JWS)
- RFC 7516 — JSON Web Encryption (JWE)
- RFC 7517 — JSON Web Key (JWK)
- RFC 7518 — JSON Web Algorithms (JWA)
- RFC 7519 — JSON Web Token (JWT)
- RFC 7523 — JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants
- RFC 7636 — Proof Key for Code Exchange by OAuth Public Clients
- OpenID Connect Core 1.0
- OpenID Connect Discovery 1.0
- OpenID Connect Dynamic Client Registration 1.0
- OAuth 2.0 Multiple Response Type Encoding Practices
- OAuth 2.0 Form Post Response Mode

Articles below may help understanding these specifications.

- The Simplest Guide To OAuth 2.0
- Diagrams And Movies Of All The OAuth 2.0 Flows
- Diagrams of All The OpenID Connect Flows
- Understanding ID Token

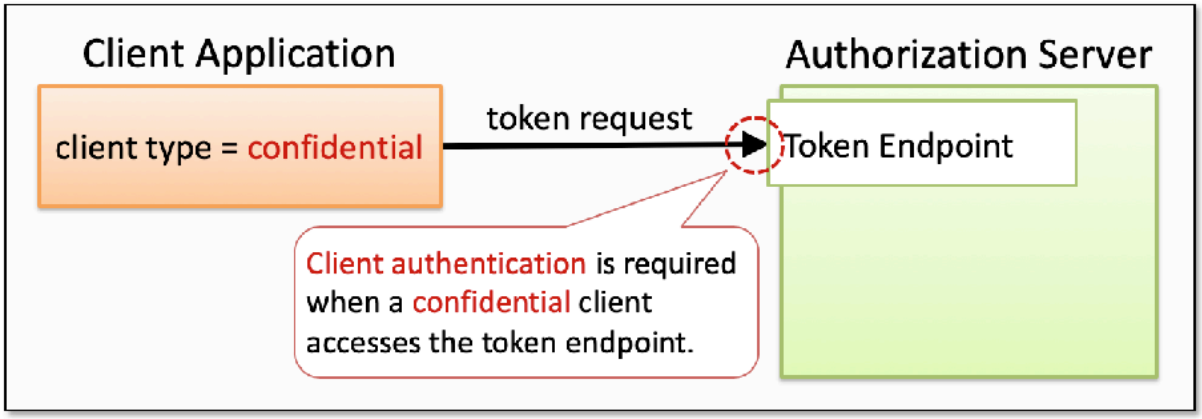
Mutual TLS

In general, "Mutual TLS" means that a client is also required to present its X.509 certificate in a TLS connection. However, in the context of FAPI, Mutual TLS means the following two which are defined in "**RFC 8705 OAuth 2.0 Mutual TLS Client Authentication and Certificate-Bound Access Tokens**" (**MTLS**).

- OAuth client authentication using a client certificate
- Tokens bound to a client certificate

OAuth Client Authentication using a Client Certificate

When a **confidential client** (RFC 6749, 2. Client Types) accesses a token endpoint (RFC 6749, 3.2. Token Endpoint), **client authentication** (RFC 6749, 2.3. Client Authentication) is required. Client authentication is a process where a client application proves it has its confidential authentication information.



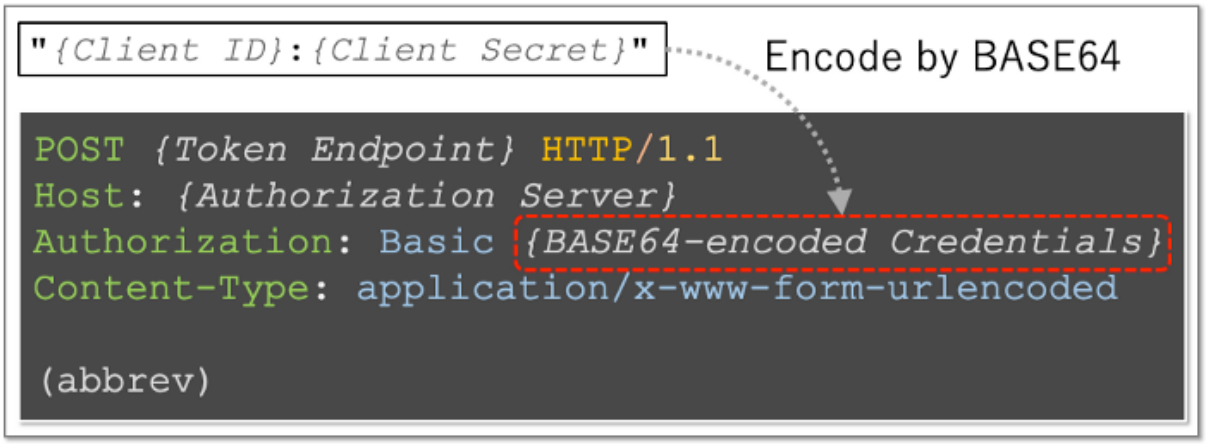
Client Authentication at Token Endpoint

There are several ways for client authentication. The following are client authentication methods listed in OIDC Core, 9. Client Authentication (except `none`).

- `client_secret_basic` — Basic Authentication using a pair of client ID and client secret
- `client_secret_post` — Embedding a pair of client ID and client secret in a request body

- `client_secret_jwt` — Passing a JWT signed by a key based on a client secret with a symmetric algorithm
- `private_key_jwt` — Passing a JWT signed by a private key with an asymmetric algorithm

In `client_secret_basic` and `client_secret_post`, a client application directly shows the server its client secret to prove that it has the confidential information.



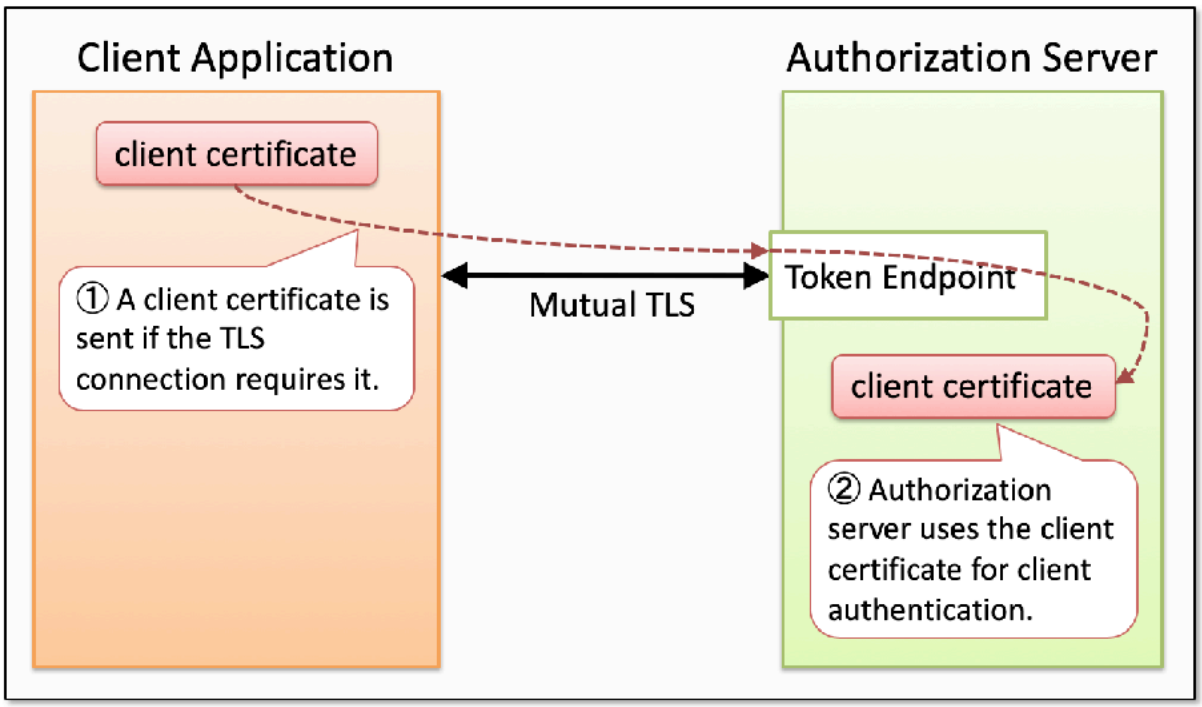
Client Authentication using Basic Authentication

In `client_secret_jwt`, a client application indirectly proves that it has the client secret by signing a JWT with the client secret and passing the JWT to the server. On the other hand, in `private_key_jwt`, signing is performed with an asymmetric private key and the server verifies the signature with the public key corresponding to the private key.

Apart from the above, "2. Mutual TLS for OAuth Client Authentication" of RFC 8705 introduces new client authentication methods below.

- `tls_client_auth` — Utilizing a PKI client certificate used in a TLS connection
- `self_signed_tls_client_auth` — Utilizing a self-signed client certificate used in a TLS connection

These two utilize the client certificate used in a TLS connection between the client and the token endpoint for client authentication.



Client Authentication using JWT
Client Certificate for Client Authentication

In `tls_client_auth`, the PKI client certificate used in a TLS connection established between a client and a server is used for client authentication. The server verifies the client certificate (this should be done even in a context irrelevant to OAuth) and then checks whether the Subject Distinguished Name or Subject Alternative Name matches the pre-registered one.

For this process, client applications that want to use `tls_client_auth` for client authentication must register Subject Distinguished Name or Subject Alternative Name into the server in advance. The specification newly defines the following client metadata for this purpose (RFC 8705, 2.1.2 Client Registration Metadata).

- `tls_client_auth_subject_dn`
- `tls_client_auth_san_dns`
- `tls_client_auth_san_uri`
- `tls_client_auth_san_ip`
- `tls_client_auth_san_email`

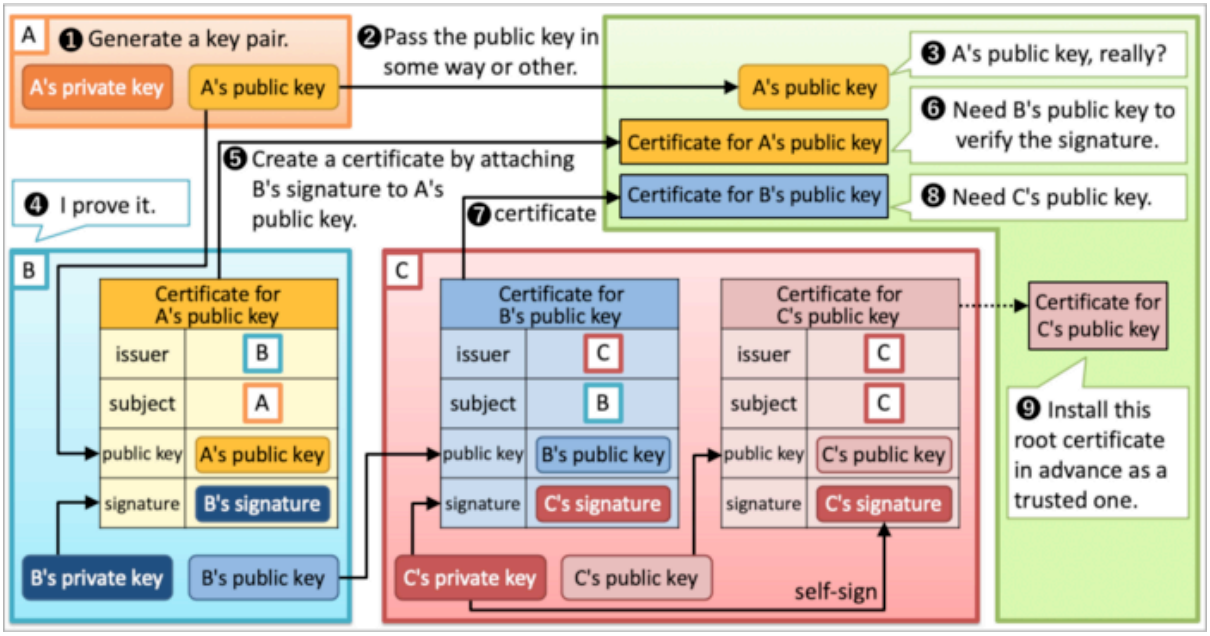
In `self_signed_tls_client_auth`, a self-signed client certificate is used instead of a PKI client certificate. To use this client authentication method, client applications have to register a self-signed client certificate into the server in advance.

The following table is the list of client authentication methods mentioned in the FAPI specification.

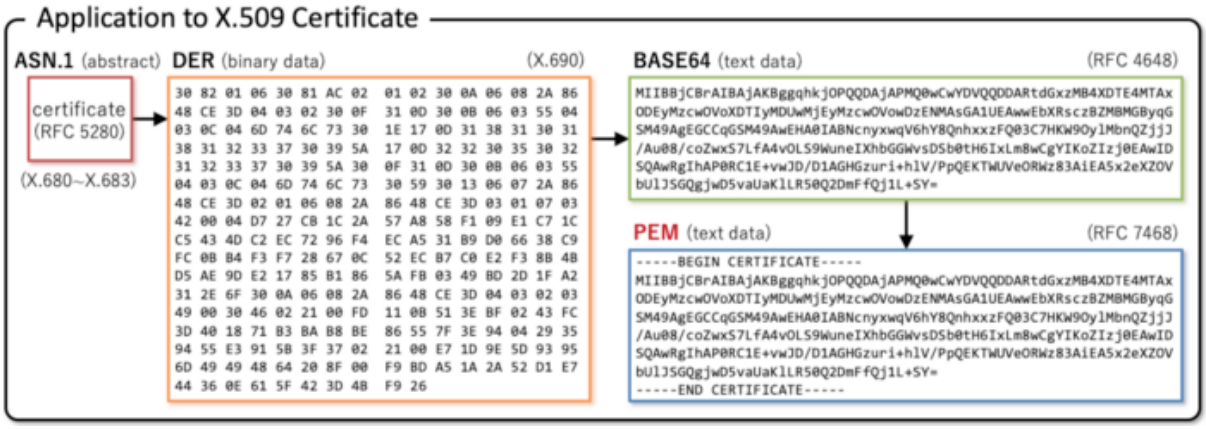
Client Authentication Method	Required Parameters	Remarks
`client_secret_basic`	client ID, client secret	basic authentication
`client_secret_post`	client ID, client secret	form parameters in POST
`client_secret_jwt`	JWT	signed with client secret
`private_key_jwt`	JWT	signed with private key
`tls_client_auth`	client certificate	PKI certificate in TLS
`self_signed_tls_client_auth`	client certificate	self-signed certificate in TLS

Client Authentication Methods

For detailed explanation about client authentication, please read "[OAuth 2.0 Client Authentication]". Also, if you are not familiar with X.509 certificate, please read "[Illustrated X.509 Certificate]".



X.509 Certificate Chain



X.509 Certificate in PEM Format

Certificate-Bound Tokens

Once a traditional access token is leaked, an attacker can access APIs with the access token. Traditional access tokens are just like a train ticket which anyone can use once it is stolen.

An idea to mitigate this vulnerability is to check whether the API caller bringing an access token matches the legitimate holder of the access token when an API call is made. This is just like the boarding procedure for international flights where passengers are required to show not only a plane ticket but also their passport.

This idea is called "**Proof of Possession**" (**PoP**) and FAPI lists "**Mutual TLS**" as an only possible option of PoP (— in the previous versions (ID1 & ID2), "**Token Binding**" was mentioned as a PoP mechanism but it was dropped by the final version). In this context, "Mutual TLS" means the specification defined in "3. Mutual-TLS Client Certificate-Bound Access Tokens" of RFC 8705.

Because Mutual TLS has several meanings as explained above and I actually experienced a problematic conversation like below,

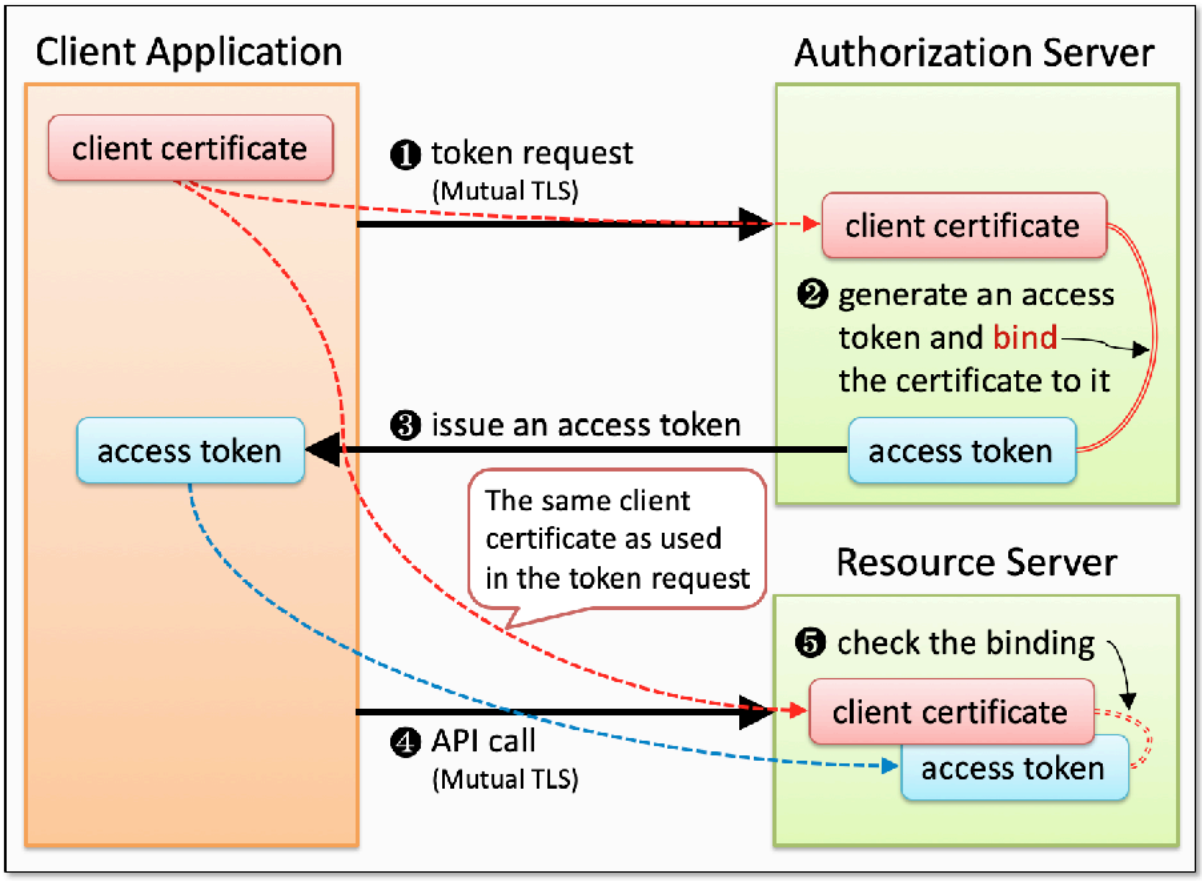
> ***Me**: The API management solution of your company does not support Mutual TLS (as a PoP mechanism).*
> ***The company**: Not correct. Our solution supports Mutual TLS (because it can be configured to request a client certificate for TLS communication).*

I've personally decided to call Mutual TLS as a PoP mechanism "**Certificate Binding**". This naming is not so bad (at least for me) because it sounds symmetrical

to Token Binding and because actual implementations will eventually become just binding a certificate to an access token and won't care whether the certificate has been extracted from a mutual TLS connection or has come from somewhere else.

In an implementation of Certificate Binding, when the token endpoint of an authorization server issues an access token, it calculates the hash value of the client certificate presented by the client application in the TLS connection and remembers the binding between the access token and the hash value (or embeds the hash value into the access token if the implementation of the access token is a self-contained JWT).

When the client application accesses an API of the target resource server, it uses the same client certificate that was previously used in the communication with the token endpoint. The implementation of the API extracts an access token and a client certificate from the request, calculates the hash value of the client certificate and checks the hash value matches the one that is associated with the access token. If they match, the API implementation accepts the request. If not, it rejects the request.



Certificate Binding

It is relatively easy to implement Certificate Binding because it can be implemented only if the client certificate is accessible. On the other hand, Token Binding is relatively hard because it is necessary to modify multiple layers such as TLS layer and HTTP layer. In addition, the future is uncertain as Chrome has removed the Token Binding feature although the community strongly tried to urge Chrome team to rethink it ("Intent to Remove: Token Binding"). However, anyway, related specifications were promoted to RFCs at the beginning of October, 2018.

- RFC 8471 — The Token Binding Protocol Version 1.0
- RFC 8472 — Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation
- RFC 8473 — Token Binding over HTTP

"OAuth 2.0 Token Binding" (its status is "expired") is a specification that defines rules to apply Token Binding to OAuth 2.0 tokens based on the specifications listed above.

NOTE: The final version of the FAPI specification dropped Token Binding due to its unlikeliness of future availability.

JARM

JARM is a new specification which was approved at the same timing with FAPI Implementer's Draft 2. JARM is referred to in FAPI Part 2.

- Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)

The specification defines new values for the `response_mode` request parameter as shown below.

1. `query.jwt`
2. `fragment.jwt`
3. `form_post.jwt`
4. `jwt`

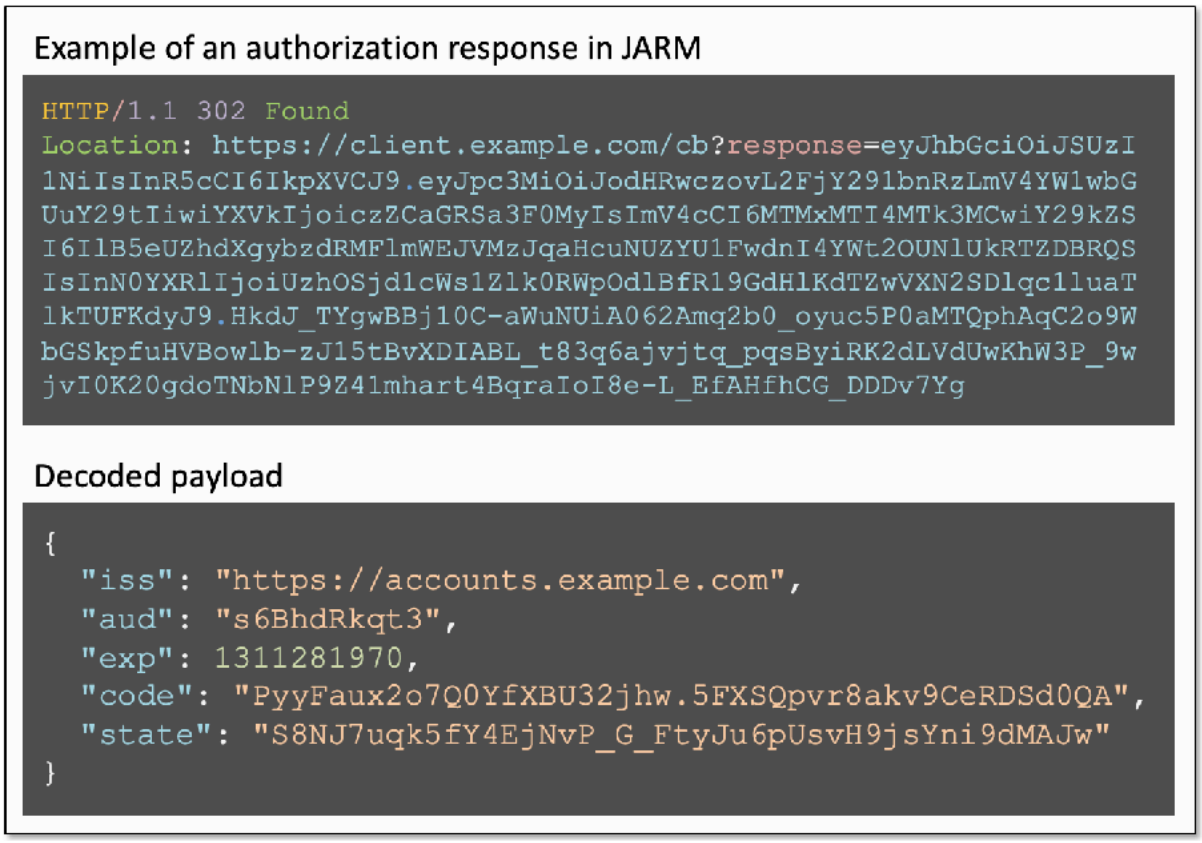
If one of the above is specified, response parameters of an authorization response are packed into a JWT and the JWT is returned as the value of a single `response` response parameter.

For example, a traditional authorization response in the authorization code flow looks like below. `code` and `state` response parameters are included separately.

```
HTTP/1.1 302 Found
Location: https://client.com/callback?code={CODE}&state={STATE}
```

On the other hand, if `response_mode=query.jwt` is added to an authorization request, the authorization response will become like below.

```
HTTP/1.1 302 Found
Location: https://client.com/callback?response={JWT}
```



JARM example

Because the JWT is signed by a key of the server, a client can confirm that the response has not been tampered by verifying the signature of the JWT.

Client Metadata for JARM

Before using JARM, client applications have to set a value to the `authorization_signed_response_alg` metadata in advance. The metadata represents an algorithm for signature of response JWTs. If the value of the `response_mode` request parameter is `*.jwt` although the metadata is not set, the authorization request fails because the specification requires response JWTs be always signed.

To encrypt response JWTs, algorithms have to be set in advance to the `authorization_encrypted_response_alg` metadata and the `authorization_encrypted_response_enc` metadata. To use an asymmetric algorithm, configuration about client's public key is necessary, too.

The screenshot below is client-side settings for JARM in Authlete's web console that is provided for client management.

Authorization Response Algorithms (in Developer Console provided by Authlete)

Server Metadata for JARM

Discovery information of authorization servers that support JARM includes one or more of query.jwt, fragment.jwt, form_post.jwt and jwt in the list of supported response modes (response_modes_supported). Also, discovery information includes the following metadata related to algorithms used for response JWTs.

`authorization_signing_alg_values_supported` — supported algorithms for signing

`authorization_encryption_alg_values_supported` — supported algorithms for key encryption

`authorization_encryption_enc_values_supported` — supported algorithms for payload encryption

Discovery information of authorization servers that support JARM completely will include data as shown below.

```
"response_modes_supported": [
  "query", "fragment", "form_post",
  "query.jwt", "fragment.jwt", "form_post.jwt", "jwt"
],

"authorization_signing_alg_values_supported": [
  "HS256", "HS384", "HS512", "RS256", "RS384", "RS512",
  "ES256", "ES384", "ES512", "PS256", "PS384", "PS512"
],

"authorization_encryption_alg_values_supported": [
  "RSA1_5", "RSA-OAEP", "RSA-OEAP-256",
  "A128KW", "A192KW", "A256KW",
  "dir",
  "ECDH-ES", "ECDH-ES+A128KW", "ECDH-ES+A192KW", "ECDH-ES+A256KW",
  "A128GCMKW", "A192GCMKW", "A256GCMKW",
  "PBES2-HS256+A128KW", "PBES2-HS384+A192KW", "PBES2-HS512+A256KW"
],

"authorization_encryption_enc_values_supported": [
  "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512",
  "A128GCM", "A192GCM", "A256GCM"
],
```

Server Metadata related to JARM

Part 1: Baseline

As introduction of prior knowledge was done, let's start the main part of this article. To begin with, "Part 1" which defines baseline security profile.

Requirements for Authorization Server

"5.2.2. Authorization server" in "Part 1" lists requirements for authorization server. Let's take a look one by one.

> ***Part 1: 5.2.2. Authorization server, 1.***
>
> *shall support confidential clients;*
>
> ***Part 1: 5.2.2. Authorization server, 2.***
>
> *should support public clients;*

The definition of "***confidential clients***" and "***public clients***" is described in "2.1. Client Types" of RFC 6749. I don't explain the difference between the client types here as it is prior knowledge for those who read the FAPI specification. However, the relationship between client types and OAuth 2.0 flows is often misunderstood even by those who are familiar with OAuth 2.0. It is only the combination of a "public client" and "client credentials flow" that RFC 6749 explicitly prohibits. Other combinations are allowed. Without this understanding, you would misread the FAPI specification.

Flow	Client Type	
	confidential	public
authorization code	allowed	allowed
implicit	allowed	allowed
resource owner password credentials	allowed	allowed
client credentials	allowed	prohibited

Combinations of Flow and Client Type (RFC 6749)

Flow	Client Type	
	confidential	public
CIBA POLL	allowed	prohibited
CIBA PING	allowed	prohibited
CIBA PUSH	allowed	prohibited

Combinations of Flow and Client Type (CIBA)

Just FYI. It is confidential clients only that are allowed to make backchannel authentication requests which are defined in CIBA.

> ***Part 1: 5.2.2. Authorization server, 3.***
>
> *shall provide a client secret that adheres to the requirements in section 16.19 of OIDC if a symmetric key is used;*

OIDC Core states that a value calculated based on a client secret must be used as the shared key when a symmetric algorithm is used for signing and encryption. If the entropy of the client secret is lower than the one required by the algorithm, the strength of the algorithm is weakened. Therefore, "16.19. Symmetric Key Entropy" requires that client secrets have entropy strong enough for used algorithms. For example, when `HS256` (HMAC using SHA-256) is used for signing algorithm of ID tokens, client secrets must have 256-bit entropy at minimum.

> ***Part 1: 5.2.2. Authorization server, 4.***
>
> *shall authenticate the confidential client using one of the following methods:*
>
> *1. Mutual TLS for OAuth Client Authentication as specified in section 2 of MTLS;*
>
> *2.`client_secret_jwt` or `private_key_jwt` as specified in section 9 of OIDC;*

Note that `client_secret_basic` and `client_secret_post` defined in RFC 6749 are not allowed as client authentication methods at the token endpoint.

Client Authentication Method	Use
client_secret_basic	✕
client_secret_post	✕
client_secret_jwt	○
private_key_jwt	○
tls_client_auth	○
self_signed_tls_client_auth	○

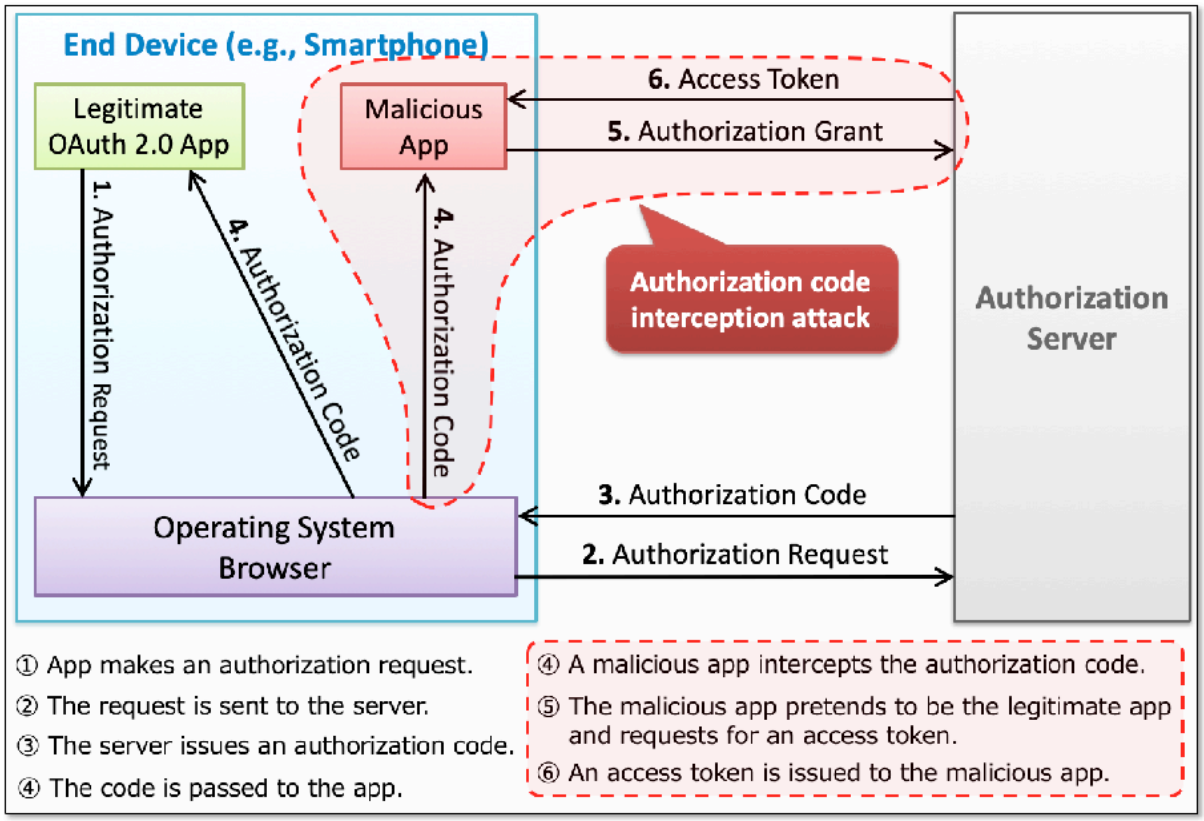
Client Authentication Methods allowed in FAPI Part 1

> ***Part 1: 5.2.2. Authorization server, 5.***
>
> *shall require and use a key of size 2048 bits or larger for RSA algorithms;*
>
> ***Part 1: 5.2.2. Authorization server, 6.***
>
> *shall require and use a key of size 160 bits or larger for elliptic curve algorithms;*

For example, when `private_key_jwt` is used as client authentication method and RSA is used for signing the JWT, the key size must be 2048 or bigger. Likewise, when an elliptic curve algorithm is used, the key size must be 160 at minimum.

> ***Part 1: 5.2.2. Authorization server, 7.***
>
> *shall require RFC7636 with `S256` as the code challenge method;*

It is required to implement RFC 7636 (PKCE) which is a countermeasure for "authorization code interception attack".



Authorization Code Interception Attack

RFC 7636 has added `code_challenge` and `code_challenge_method` request parameters to the authorization request and `code_verifier` request parameter to the token request. Because the default value of `code_challenge_method` is `plain`, authorization requests that comply with FAPI must include `code_challenge_method=S256` explicitly.

See "Proof Key for Code Exchange (RFC 7636)" for details about PKCE.

Part 1: 5.2.2. Authorization server, 8.

shall require redirect URIs to be pre-registered;

In RFC 6749, registration of redirect URIs is not required under some conditions. For FAPI, registration of redirect URIs is always required.

Part 1: 5.2.2. Authorization server, 9.

shall require the `redirect_uri` *parameter in the authorization request;*

In RFC 6749, the `redirect_uri` request parameter of an authorization request can be omitted under some conditions. For FAPI, the request parameter must be always included. OIDC has the same requirement.

Part 1: 5.2.2. Authorization server, 10.

shall require the value of `redirect_uri` *to exactly match one of the pre-registered redirect URIs;*

When an authorization server checks whether the value of the `redirect_uri` request parameter matches a pre-registered one, the rule described in "6. Normalization and Comparison" of RFC 3986 (Uniform Resource Identifier (URI): Generic Syntax) is applied unless the pre-registered one is an absolute URI.

On the other hand, FAPI (and OIDC also) requires that simple string comparison be always used to check whether the redirect URIs match.

Part 1: 5.2.2. Authorization server, 11.

shall require user authentication to an appropriate Level of Assurance for the operations the client will be authorized to perform on behalf of the user;

It is required that user authentication performed during authorization process satisfy an appropriate level of assurance. ID1 and ID2 required LoA (Level of Assurance) 2, which is defined in X.1254 (Entity authentication assurance framework. However, the Final version made the requirement more abstract (= changed the requirement from "LoA2" to "appropriate LoA").

Part 1: 5.2.2. Authorization server, 12.

shall require explicit approval by the user to authorize the requested scope if it has not been previously authorized;

Indeed.

> ***Part 1: 5.2.2. Authorization server, 13.***
>
> *shall reject an authorization code (Section 1.3.1 of RFC6749) if it has been previously used;*

Prohibiting reuse of authorization codes and ensuring that authorization codes have never been used previously are different things. If the current implementation of an authorization server uses randomly-generated strings as authorization codes and removes them from the database after they are used, the authorization codes have to be kept in the database even after they are used just only for the verification. If strings that represent authorization codes are generated randomly with high enough entropy, it is wasteful to keep authorization codes in the database even after their use.

A certain famous engineer says ***"Most implementations prevent reuse of authorization codes by deleting corresponding database records and don't check if they have been used previously, and such implementations are sufficient enough."***

> ***Part 1: 5.2.2. Authorization server, 14.***
>
> *shall return token responses that conform to Section 4.1.4 of RFC6749;*

This is not a FAPI-specific requirement. Every authorization server implementation that claims it supports OAuth 2.0 must conform to Section 4.1.4 of RFC 6749.

> ***Part 1: 5.2.2. Authorization server, 15.***
>
> *shall return the list of granted scopes with the issued access token if the request was passed in the front channel and was not integrity protected;*

In RFC 6749, the `scope` response parameter can be omitted unless requested scopes and granted ones are different (RFC 6749, 5.1. Successful Response). In FAPI, the `scope` response parameter is required (even if the requested scopes and granted

ones are equal) if the authorization request is passed in the front channel and is not integrity protected.

"Integrity protected" here means that a Request Object (OIDC Core Section 6 or JAR) is used.

> ***Part 1: 5.2.2. Authorization server, 16.***
>
> *shall provide non-guessable access tokens, authorization codes, and refresh token (where applicable), with sufficient entropy such that the probability of an attacker guessing the generated token is computationally infeasible as per RFC 6749 Section 10.10;*

ID2 requires that access tokens have a minimum of 128 bits of entropy, but the Final version avoids mentioning the exact size of the minimum entropy and just says "sufficient entropy".

> ***Part 1: 5.2.2. Authorization server, 17.***
>
> *should clearly identify the details of the grant to the user during authorization as in 16.18 of OIDC;*

Suppose that a client application requests `payment` scope. A typical authorization page will tell the user just that the client application is requesting the `payment` scope. However, recent regulations in financial industries require that details be explained to the user. For example, information about the purpose of the `payment` scope, the amount of money transferred, and so on. Generally speaking, recent regulations require that grant be more specific.

UK Open Banking has invented "**Lodging Intent**" for the purpose. In the mechanism, (a) a client application registers details of grant it wants into an authorization server in advance, (b) the authorization server issues an intent ID that represents the registered details, and (c) the client makes an authorization request with the intent ID. As a result, the authorization server can generate an authorization page which includes the details of the authorization request.

To make the lodging intent pattern available as standards, OpenID Foundation has developed two separate specifications; "**OAuth 2.0 Pushed Authorization**

Requests" (**PAR**) and "OAuth 2.0 Rich Authorization Requests" (**RAR**). These specifications will be mentioned again later.

> ***Part 1: 5.2.2. Authorization server, 18.***
>
> *should provide a mechanism for the end-user to revoke access tokens and refresh tokens granted to a client as in 16.18 of OIDC;*

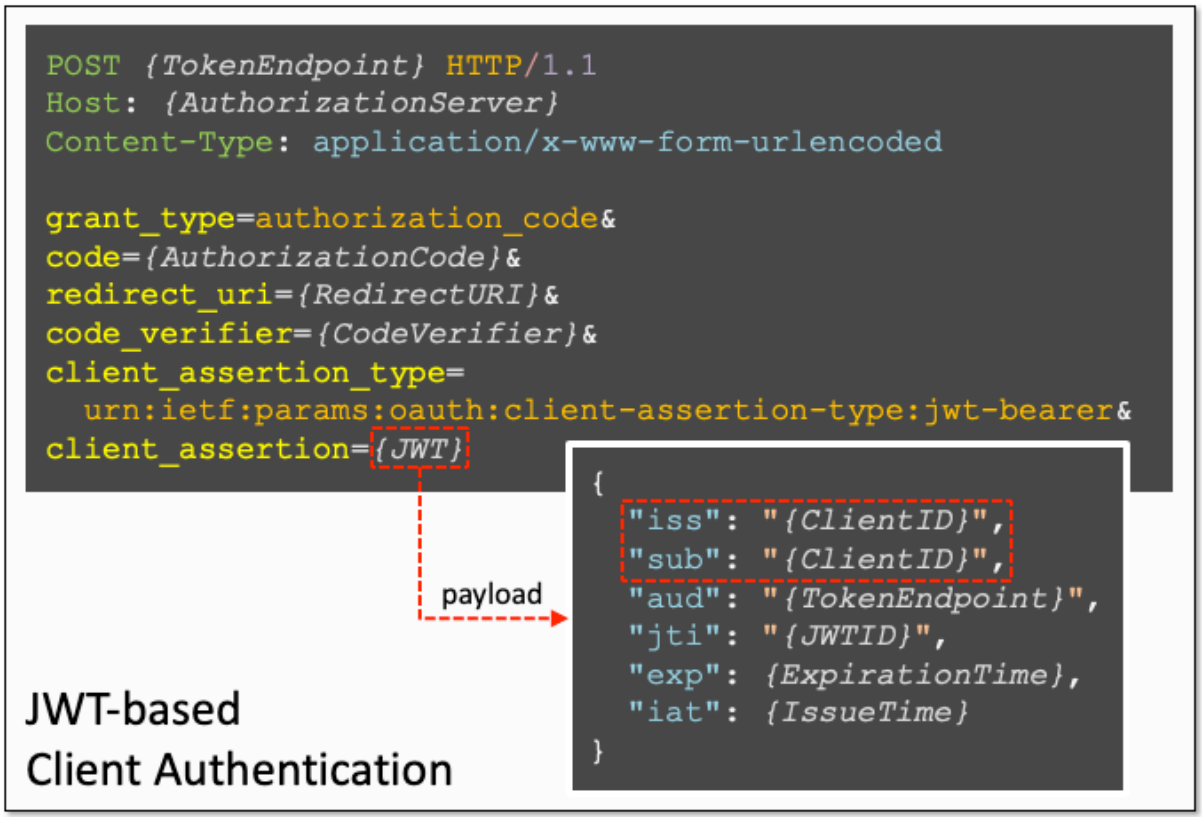
It should be noted that, **if the format of access tokens is self-contained-type (e.g. JWT), the access tokens cannot be revoked** unless the system implements and operates a mechanism like CRL (Certificate Revocation List) or OCSP (Online Certificate Status Protocol) of PKI (Public Key Infrastructure). If the system does not provide such mechanism, it means that the system has decided to give up revocation of access tokens. In this case, the duration of access tokens must be short enough to mitigate damage of access token leakage. See "**OAuth Access Token Implementation**" for further discussion.

> ***Part 1: 5.2.2. Authorization server, 19.***
>
> *shall return an* `invalid_client` *error as defined in 5.2 of RFC6749 when mismatched client identifiers were provided through the client authentication methods that permits sending the client identifier in more than one way;*

FAPI Part 1 requires MTLS (`tls_client_auth`, `self_signed_tls_client_auth`) or JWT (`client_secret_jwt`, `private_key_jwt`) for client authentication.

MTLS uses a client certificate but a certificate does not include the client identifier of the client which tries to authenticate itself with the certificate. Therefore, the `client_id` request parameter needs to be given explicitly.

On the other hand, JWT-based client authentication methods present a JWT as the value of the `client_assertion` request parameter and the JWT contains the client identifier as the value of the `iss` claim. Therefore, the `client_id` request parameter is not necessary. In addition, according to RFC 7523, 3. and OIDC Core, 9., the `sub` claim also holds the client identifier when a JWT is used for client authentication.



JWT-based Client Authentication and Client Identifiers

In MTLS, it is only the `client_id` request parameter that represents a client identifier. On the other hand, in JWT-based client authentication, both the `iss` claim and the `sub` claim hold a client identifier. The values of the claims must match. Also, if the `client_id` request parameter is redundantly given although JWT-based client authentication is used, the value of the request parameter must match the client identifier, too.

> ***Part 1: 5.2.2. Authorization server, 20.***
>
> *shall require redirect URIs to use the* `https` *scheme;*

This sentence added by FAPI Implementer's Draft 2 is short but has a big impact. Because of this sentence, developers **cannot use custom schemes in FAPI** any more. To process redirection on client side only without preparing an external Web server, developers have to use the method described in "7.2. Claimed "https" Scheme URI Redirection" of BCP 212 (OAuth 2.0 for Native Apps).

Part 1: 5.2.2. Authorization server, 21.

should issue access tokens with a lifetime of under 10 minutes unless the tokens are sender-constrained; and

This requirement was added by the FAPI Final version. "sender-constrained" here means that access tokens have to be bound to a client certificate (MTLS).

Part 1: 5.2.2. Authorization server, 22.

shall support OIDD, may support RFC8414 and shall not distribute discovery metadata (such as the authorization endpoint) by any other means.

This requirement was added by the FAPI Final version. OIDD here is short for "OpenID Connect Discovery 1.0". Therefore, authorization servers for FAPI must implement a "discovery endpoint" which is defined in OIDD Section 4.

Part 1: 5.2.2.1. Returning authenticated user's identifier

Further, if it is desired to provide the authenticated user's identifier to the client in the token response, the authorization server:

Section 5.2.2.1. lists requirements that an authorization server must follow when the authenticated user's identifier is requested. In other words, when an ID token is requested.

Part 1: 5.2.2.1. Returning authenticated user's identifier, 1.

shall support the authentication request as in Section 3.1.2.1 of OIDC;

"3.1.2.1. Authentication Request" of OIDC Core is the definition of a request to an authorization endpoint in the context of OpenID Connect. RFC 6749 calls a request to an authorization endpoint "authorization request". OIDC Core calls it "authentication request". Aside from the names, considering that the specification of an authorization endpoint is the main part of OIDC Core, the FAPI's requirement is almost equal to stating "***shall support OIDC Core***".

> ***Part 1: 5.2.2.1. Returning authenticated user's identifier, 2.***
>
> *shall perform the authentication request verification as in Section 3.1.2.2 of OIDC;*
>
> ***Part 1: 5.2.2.1. Returning authenticated user's identifier, 3.***
>
> *shall authenticate the user as in Section 3.1.2.2 and 3.1.2.3 of OIDC;*
>
> ***Part 1: 5.2.2.1. Returning authenticated user's identifier, 4.***
>
> *shall provide the authentication response as in Section 3.1.2.4 and 3.1.2.5 of OIDC depending on the outcome of the authentication;*
>
> ***Part 1: 5.2.2.1. Returning authenticated user's identifier, 5.***
>
> *shall perform the token request verification as in Section 3.1.3.2 of OIDC; and*
>
> ***Part 1: 5.2.2.1. Returning authenticated user's identifier, 6.***
>
> *shall issue an ID Token in the token response when `openid` was included in the requested `scope` as in Section 3.1.3.3 of OIDC with its `sub` value corresponding to the authenticated user and optional `acr` value in ID Token.*

Summary of the requirements above is "***shall follow OIDC Core specification***." Nothing special for FAPI.

> ***Part 1: 5.2.2.2. Client requesting openid scope***
>
> *If the client requests the `openid` scope, the authorization server*
>
> *1. shall require the `nonce` parameter defined in Section 3.1.2.1 of OIDC in the authentication request.*

OIDC Section 3.1.2.1 (Authorization Code Flow) states that `nonce` is optional. On the other hand, OIDC Section 3.2.2.1 (Implicit Flow) states that `nonce` is mandatory.

The FAPI requirement above requires `nonce` even in the authorization code flow if `openid` is included in `scope`.

> ***Part 1: 5.2.2.3. Clients not requesting openid scope***

If the client does not requests the `openid` scope, the authorization server

1. shall require the `state` parameter defined in Section 4 of RFC6749

In RFC 6749, the `state` parameter is optional. FAPI makes the parameter mandatory when `openid` is not included in `scope`.

Requirements for Public Client

"5.2.3. Public client" of "Part 1" lists requirements for public clients. Let's take a look one by one.

> ***Part 1: 5.2.3. Public client, 1.***
>
> *shall support RFC7636;*

RFC 7636 is PKCE.

> ***Part 1: 5.2.3. Public client, 2.***
>
> *shall use `S256` as the code challenge method for the RFC7636;*

This means *"**an authorization request must include `code_challenge_method=S256`**."*

> ***Part 1: 5.2.3. Public client, 3.***
>
> *shall use separate and distinct redirect URI for each authorization server that it talks to;*
>
> ***Part 1: 5.2.3. Public client, 4.***
>
> *shall store the redirect URI value in the resource owner's user-agents (such as browser) session and compare it with the redirect URI that the authorization response was received at, where, if the URIs do not match, the client shall terminate the process with error;*

These requirements are so clear that further explanation is not needed.

> ***Part 1: 5.2.3. Public client, 5.***
>
> *(withdrawn); and*

"(withdrawn)" here indicates that the requirement which existed in the previous FAPI versions has been withdrawn. You'll see more "withdrawn"s in following sections, too.

> ***Part 1: 5.2.3. Public client, 6.***
>
> *shall implement an effective CSRF protection.*

In normal cases, CSRF protection is implemented on server side. What is CSRF protection as a requirement for public clients? This is CSRF protection for redirect URIs. The following is an excerpt from "10.12. Cross-Site Request Forgery" of RFC 6749.

> *The client MUST implement CSRF protection for its redirection URI. This is typically accomplished by requiring any request sent to the redirection URI endpoint to include a value that binds the request to the user-agent's authenticated state (e.g., a hash of the session cookie used to authenticate the user-agent). The client SHOULD utilize the "state" request parameter to deliver this value to the authorization server when making an authorization request.*

In addition to the requirements from "Public client, 1" to "Public client, 6", "***if it is desired to obtain a persistent identifier of the authenticated user***", that is, if an ID token is requested, an authorization request by a public client:

> ***Part 1: 5.2.3. Public client, 7.***
>
> *shall include* `openid` *in the* `scope` *value; and*
>
> ***Part 1: 5.2.3. Public client, 8.***
>
> *shall include the* `nonce` *parameter defined in Section 3.1.2.1 of OIDC in the authentication request.*

On the other hand, "***If*** `openid` ***is not in the*** `scope` ***value***", an authorization request by a public client:

> ***Part 1: 5.2.3. Public client, 9.***
>
> *shall include the* `state` *parameter defined in section 4.1.1 of RFC6749;*

Part 1: 5.2.3. Public client, 10.

shall verify that the `scope` *received in the token response is either an exact match, or contains a subset of the* `scope` *sent in the authorization request; and*

Part 1: 5.2.3. Public client, 11.

shall only use Authorization Server metadata obtained from the metadata document published by the Authorization Server at its well known endpoint as defined in OIDD or RFC 8414.

Requirements for Confidential Client

"5.2.4. Confidential client" of "Part 1" lists requirements for confidential clients. The requirements are positioned as additions to the requirements for public clients. Therefore, confidential clients must follow not only the requirements in 5.2.4 but also the requirements in 5.2.3.

> ***Part 1: 5.2.4. Confidential client, 1.***
>
> *shall support the following methods to authenticate against the token endpoint:*
>
> *1. Mutual TLS for OAuth Client Authentication as specified in Section 2 of MTLS, and*
>
> *2.*`client_secret_jwt` *or* `private_key_jwt` *as specified in Section 9 of OIDC;*

Note that client authentication methods defined in RFC 6749 (`client_secret_basic` and `client_secret_post`) cannot be used.

> ***Part 1: 5.2.4. Confidential client, 2.***
>
> *shall use RSA keys with a minimum 2048 bits if using RSA cryptography;*
>
> ***Part 1: 5.2.4. Confidential client, 3.***
>
> *shall use elliptic curve keys with a minimum of 160 bits if using Elliptic Curve cryptography; and*
>
> ***Part 1: 5.2.4. Confidential client, 4.***
>
> *shall verify that its client secret has a minimum of 128 bits if using symmetric key cryptography.*

These requirements apply when encrypted JWTs are used.

Requirements for Protected Resources

"6.2.1. Protected resources provisions" of "Part 1" lists requirements for protected resources.

> ***Part 1: 6.2.1. Protected resource provisions, 1.***
>
> *shall support the use of the HTTP GET method as in Section 4.3.1 of RFC7231;*
>
> ***Part 1: 6.2.1. Protected resource provisions, 2.***
>
> *shall accept access tokens in the HTTP header as in Section 2.1 of OAuth 2.0 Bearer Token Usage RFC6750;*

That is, protected resource endpoints must support HTTP `GET` method and be able to accept an access token in the format of `Authorization: Bearer {AccessToken}`.

```
GET {ProtectedResourceEndpoint}?{Parameters} HTTP/1.1
Host: {ResourceServer}
Authorization: Bearer {AccessToken}
```

Request to Protected Resource Endpoint

> ***Part 1: 6.2.1. Protected resource provisions, 3.***
>
> *shall not accept access tokens in the query parameters stated in Section 2.3 of OAuth 2.0 Bearer Token Usage RFC6750;*

That is, protected resource endpoints must not accept a query parameter in the format of `access_token={AccessToken}`.

> ***Part 1: 6.2.1. Protected resource provisions, 4.***
>
> *shall verify that the access token is neither expired nor revoked;*

> ***Part 1: 6.2.1. Protected resource provisions, 5.***
>
> *shall verify that the scope associated with the access token authorizes access to the resource it is representing;*
>
> ***Part 1: 6.2.1. Protected resource provisions, 6.***
>
> *shall identify the associated entity to the access token;*
>
> ***Part 1: 6.2.1. Protected resource provisions, 7.***
>
> *shall only return the resource identified by the combination of the entity implicit in the access and the granted scope and otherwise return errors as in Section 3.1 of RFC6750;*

These are general steps of access token verification that protected resource endpoints are expected to take.

"3.1. Error Codes" of RFC 6750 defines three error codes. They are `invalid_request`, `invalid_token` and `insufficient_scope`. One point those who are not familiar with RFC 6750 may feel strange is that an error code is embedded not in the response body but in the `WWW-Authenticate` HTTP header.

```
HTTP/1.1 400 Bad Request
WWW-Authenticate: Bearer error="{error code}",
  error_description="{description of the error}",
  error_uri="{URI of the page describing the error in detail}",
  scope="{scopes required to access the protected resource}"
Cache-Control: no-store
Pragma: no-cache
```

RFC 6750 Error Response

> ***Part 1: 6.2.1. Protected resource provisions, 8.***
>
> *shall encode the response in UTF-8 if applicable;*
>
> ***Part 1: 6.2.1. Protected resource provisions, 9.***
>
> *shall send the Content-type HTTP header Content-Type: application/json; if applicable;*

Protected resource endpoints in FAPI are expected to return their responses in JSON format.

> ***Part 1: 6.2.1. Protected resource provisions, 10.***
>
> *shall send the server date in HTTP `Date` header as in Section 7.1.1.2 of RFC7231;*

The format of `Date` header is defined in "7.1.1.1. Date/Time Formats" of RFC 7231. Below is an example.

```
Date: Sun, 06 Nov 1994 08:49:37 GMT
```

> ***Part 1: 6.2.1. Protected resource provisions, 11.***
>
> *shall set the response header `x-fapi-interaction-id` to the value received from the corresponding FAPI client request header or to a RFC4122 UUID value if the request header was not provided to track the interaction, e.g., `x-fapi-interaction-id: c770aef3-6784-41f7-8e0e-ff5f97bddb3a`;*

This is a requirement specific to FAPI. Responses from FAPI protected resource endpoints must include an `x-fapi-interaction-id` header.

When an incoming request has `x-fapi-interaction-id`, the same value of the header must be included in the response. Otherwise, the protected resource endpoint must generate a new value for `x-fapi-interaction-id`.

> ***Part 1: 6.2.1. Protected resource provisions, 12.***
>
> *shall log the value of `x-fapi-interaction-id` in the log entry; and*

This is also specific to FAPI. This requirement doesn't have any impact on request and response formats, but this can make it easy to correlate server-side logs and client-side logs.

> ***Part 1: 6.2.1. Protected resource provisions, 13.***
>
> *shall not reject requests with a* `x-fapi-customer-ip-address` *header containing a valid IPv4 or IPv6 address.*
>
> ***Part 1: 6.2.1. Protected resource provisions, 14.***
>
> *should support the use of Cross Origin Resource Sharing (CORS) [CORS] and or other methods as appropriate to enable JavaScript clients to access the endpoint if it decides to provide access to JavaScript clients.*

For example, if a protected resource endpoint wants to allow JavaScript clients to access it from anywhere, the endpoint should include an `Access-Control-Allow-Origin: *` header in responses.

Requirements for Clients to Protected Resources

"6.2.2. Client provisions" of "Part 1" lists requirements for clients to follow in accessing protected resources.

> ***Part 1: 6.2.2. Client provisions, 1.***
>
> *shall send access tokens in the HTTP header as in Section 2.1 of OAuth 2.0 Bearer Token Usage RFC6750; and*

That is, clients send an access token in the format of `Authorization: Bearer {AccessToken}`.

> ***Part 1: 6.2.2. Client provisions, 2.***
>
> *(withdrawn);*
>
> ***Part 1: 6.2.2. Client provisions, 3.***
>
> *may send the last time the customer logged into the client in the* `x-fapi-auth-date` *header where the value is supplied as a HTTP-date as in Section 7.1.1.1 of RFC7231, e.g.,* `x-fapi-auth-date: Tue, 11 Sep 2012 19:43:31 GMT`*;*
>
> ***Part 1: 6.2.2. Client provisions, 4.***
>
> *may send the customer's IP address if this data is available in the* `x-fapi-customer-ip-address` *header, e.g.,* `x-fapi-customer-ip-address: 2001:DB8::1893:25c8:1946` *or* `x-fapi-customer-ip-address: 198.51.100.119`*; and*
>
> ***Part 1: 6.2.2. Client provisions, 5.***
>
> *may send the* `x-fapi-interaction-id` *request header, in which case the value shall be a RFC4122 UUID to the server to help correlate log entries between client and server, e.g.,* `x-fapi-interaction-id: c770aef3-6784-41f7-8e0e-ff5f97bddb3a`*.*

These are FAPI-specific HTTP headers. It is up to clients whether to send the headers or not.

HTTP header	Value
`x-fapi-auth-date`	The last login time of the customer
`x-fapi-customer-ip-address`	The IP address of the customer
`x-fapi-interaction-id`	The unique identifier of the API call

FAPI-specific HTTP Headers

Security Considerations

"7. Security considerations" of "Part 1" lists security considerations. Summary is as follows.

7.1. — Follow BCP 195. Use TLS 1.2 or newer. Follow RFC 6125.

7.2. — Part 1 doesn't authenticate authorization request and response.

7.3. — Part 1 doesn't assure message integrity of authorization request.

7.4.1. — Part 1 doesn't discuss encryption of authorization request.

7.4.2. — Be careful not to leak information through logs.

7.4.3. — Be careful not to leak information through referrer. Make duration of access tokens short.

7.5. — Native applications shall follow BCP 212 but must not support "Private-Use URI Scheme Redirection" and "Loopback Interface Redirection". They must use `https` for the scheme of redirect URI as introduced in "Claimed https Scheme URI Redirection".

7.6. — Both FAPI implementation and underlying OAuth/OIDC implementation must be complete and correct. See OpenID Certification.

7.7. — Use a separate `issuer` per brand if multiple brands need to be supported.

"Part 2" provides solutions for security considerations listed in "Part 1", for example, by making "request object" mandatory. "Part 2" is recommended when higher security than "Part 1" is needed.

Part 2: Advanced

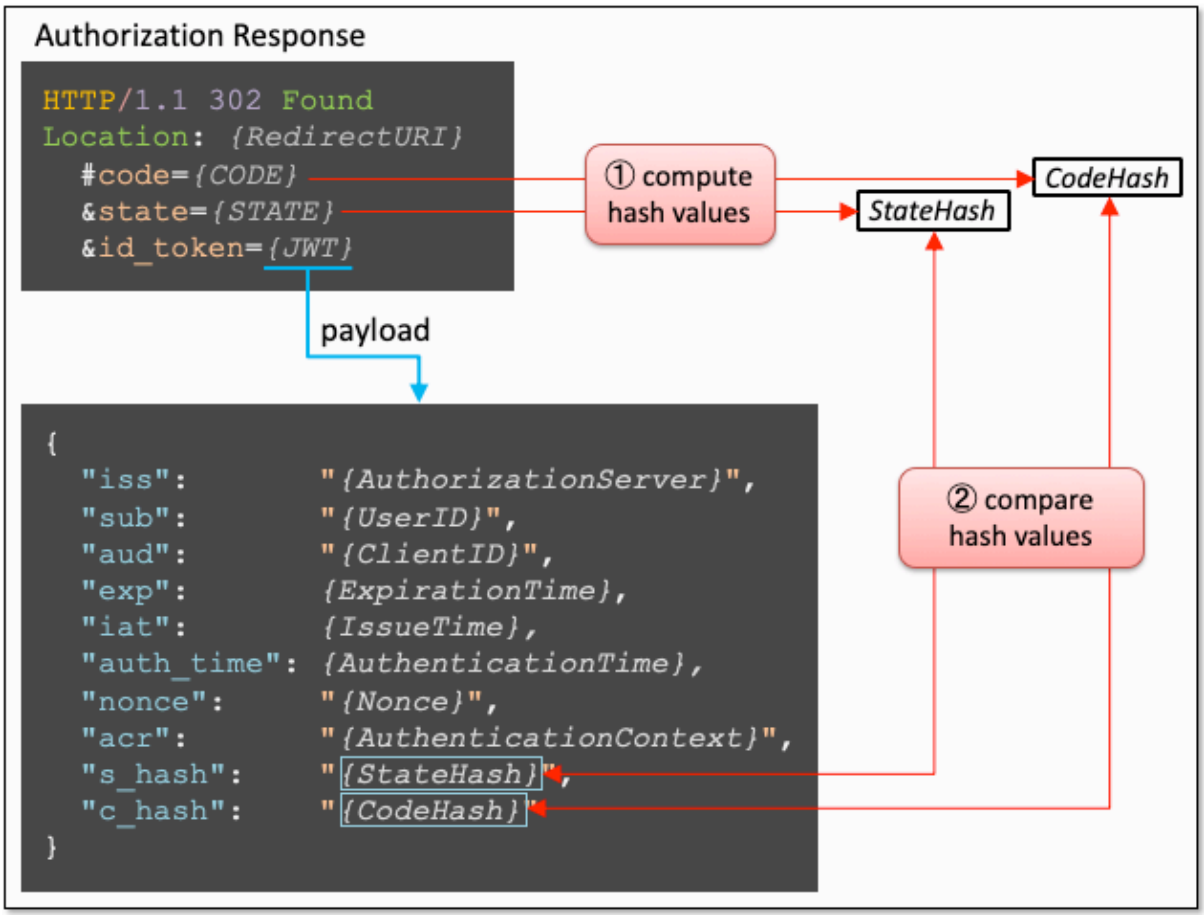
Next, let's read "Part 2" which defines advanced security profile.

Detached Signature

"5.1.1. ID Token as Detached Signature" of "Part 2" states that it uses "**ID token**" as "**detached signature**".

An ID token is signed by an authorization server, so even if an attacker tampered the content of the ID token, it could be detected. A client application that has received an ID token can confirm that the ID token has not been tampered by verifying the signature of the ID token.

If an authorization server embeds hash values of response parameters (such as `code` and `state`) into an ID token, a client application can confirm that the values of the response parameters have not been tampered by computing hash values of the response parameter values and comparing them to the hash values embedded in the ID token. In the context, the ID token is regarded as a detached signature.



ID Token as Detached Signature

For the `code` response parameter that represents an authorization code, `c_hash` has already been defined in OIDC Core as a claim that represents the hash value of `code`. Likewise, `at_hash` has been defined as a claim that represents the hash value of `access_token`.

What is missing is a claim that represents the hash value of the `state` response parameter. So, "5.1.1. ID Token as Detached Signature" defines `s_hash` for that purpose.

> ***s_hash***
> *State hash value. Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the* `state` *value, where the hash algorithm used is the hash algorithm used in the* `alg` *header parameter of the ID Token's JOSE header. For instance, if the* `alg` *is* `HS512`*, hash the state value with SHA-512, then*

> *take the left-most 256 bits and base64url encode them. The* ***`s_hash`*** *value is a case sensitive string.*

Because "Part 2" uses ID tokens as detached signatures, even if client applications don't need ID tokens in their application layer, they have to send authorization requests that require an ID token. To be exact, they have to include `id_token` in the `response_type` request parameter. This is the reason that the second requirement in "5.2.2. Authorization Server" is saying *"**shall require the `response_type` value `code id_token`**"*.

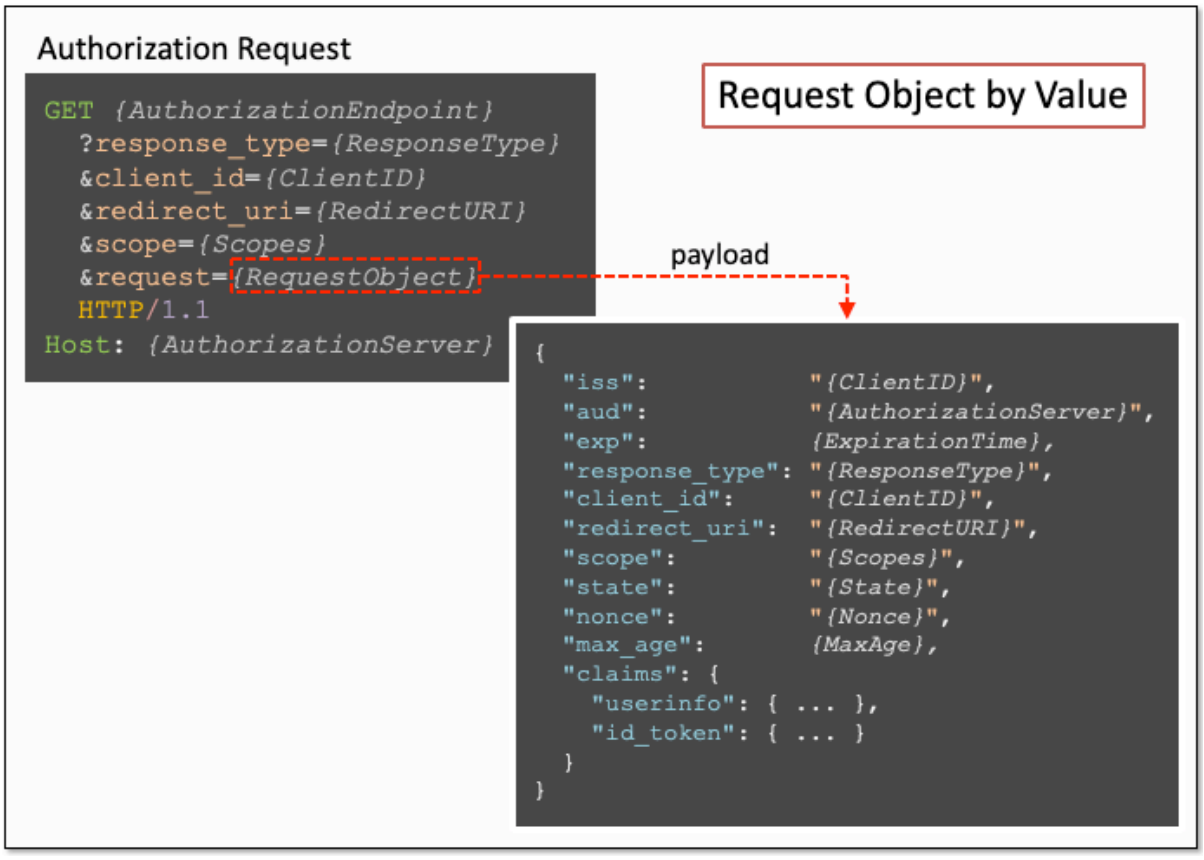
However, since Implementer's Draft 2, **ID tokens don't have to be used as detached signatures when JARM is used**. It is because the entire set of response parameters is packed into a JWT.

Requirements for Authorization Server

"5.2.2. Authorization server" of "Part 2" lists requirements for authorization server.

> ***Part 2: 5.2.2. Authorization server, 1.***
>
> *shall require a JWS signed JWT request object passed by value with the* `request` *parameter or by reference with the* `request_uri` *parameter;*

The `request` and `request_uri` parameters are defined in "6. Passing Request Parameters as JWTs" of OIDC Core. To use these parameters, the first step is **to pack request parameters into a JWT**. This JWT is called "**request object**". An authorization request (1) passes the request object as the value of the `request` parameter directly or (2) puts the request object somewhere accessible from the authorization server and passes the URI pointing to the location as the value of the `request_uri` parameter.



Passing a Request Object by Value

Signing a request object is not mandatory in OIDC Core, but signing is mandatory in FAPI Part 2. If request objects are signed, authorization servers can confirm that the request parameters have not been tampered by verifying signatures of the request objects.

To be honest, what surprised me most when I read the FAPI specification for the first time (many years ago) is this requirement. It's because I knew from my experience it is hard to implement the request object feature on authorization server side. As the feature is hard to implement and optional in OIDC, there are many authorization server implementations that claim they support OIDC but don't support request object. **Be careful not to choose an authorization server implementation that doesn't support request object if you want to build a system that supports FAPI Part 2**.

> ***Part 2: 5.2.2. Authorization server, 2.***
>
> *shall require*
>
> *1. the response_type values code id_token, or*
>
> *2. the* `response_type` *value* `code` *in conjunction with the* `response_type` *value* `jwt`*;*

To use ID token as detached signature, even if an ID token is not needed in the application layer, `id_token` must be included in the `response_type` request parameter.

But, as mentioned in the previous section, `id_token` doesn't have to be included in the `response_type` request parameter when JARM is used. "When JARM is used" is, to be concrete, "when the `response_mode` request parameter is included and its value is one of `query.jwt`, `fragment.jwt`, `form_post.jwt` and `jwt`".

NOTE: ID2 requires that `response_type` be either `code id_token` or `code id_token token` when JARM is not used, but the Final version has removed `code id_token token`.

Part 2: 5.2.2. Authorization server, 3.

(moved to 5.2.2.1);

Part 2: 5.2.2. Authorization server, 4.

(moved to 5.2.2.1);

Part 2: 5.2.2. Authorization server, 5.

shall only issue sender-constrained access tokens;

In ID2, this clause was *"**shall only issue authorization code, access token, and refresh token that are holder of key bound;**"*. However, because the requirement was impractical, it was changed to the current one. See FAPI Issue 202 for details if you are interested.

Part 2: 5.2.2. Authorization server, 6.

shall support MTLS as mechanism for constraining the legitimate senders of access tokens;

In ID2, this clause was *"**shall support [OAUTB] or [MTLS] as a holder of key mechanisms;**"*. However, OAUTB (Token Binding) was removed from the Final version due to its unlikeliness of future availability.

Part 2: 5.2.2. Authorization server, 7.

(withdrawn);

Part 2: 5.2.2. Authorization server, 8.

(moved to 5.2.2.1);

Part 2: 5.2.2. Authorization server, 9.

(moved to 5.2.2.1);

Part 2: 5.2.2. Authorization server, 10.

> *shall only use the parameters included in the signed request object passed via the* `request` *or* `request_uri` *parameter;*
>
> *shall require that all parameters are present inside the signed request object passed in the* `request` *or* `request_uri` *parameter;*

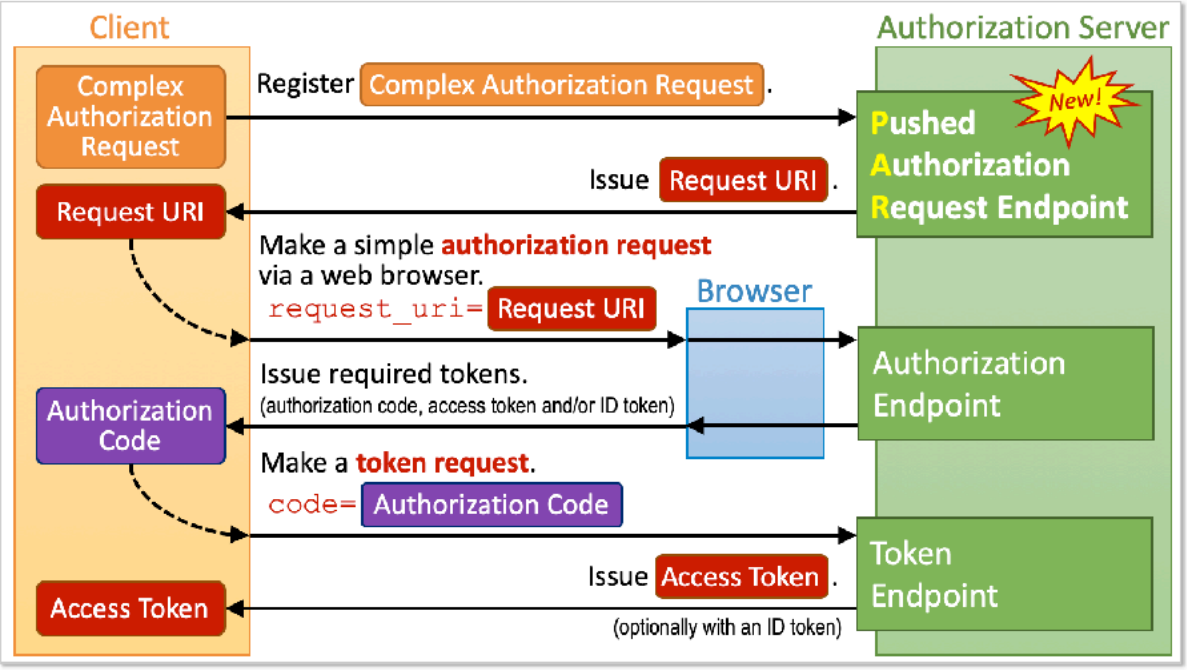
In ID2, this requirement was *"**shall require that all parameters are present inside the signed request object passed in the `request` or `request_uri` parameter;**"*. The expression was changed but the point remains the same. A request object must include all request parameters to conform to FAPI Part 2.

This is different from OIDC Core which allows request parameters to be put inside or outside a request object and merges them.

> ***Part 2: 5.2.2. Authorization server, 11.***
>
> *may support the pushed authorization request endpoint as described in PAR;*

The "**pushed authorization request endpoint**" is a new endpoint defined in "**OAuth 2.0 Pushed Authorization Requests**" (**PAR**). A client application can register an authorization request at the endpoint and obtain a **Request URI** which represents the registered authorization request. The client specifies the issued Request URI as the value of the `request_uri` request parameter when sending an authorization request to the authorization endpoint.

The following diagram excerpted from "**Illustrated PAR: OAuth 2.0 Pushed Authorization Requests**" shows the authorization code flow which utilizes the pushed authorization request endpoint.



Authorization Code Flow with Pushed Authorization Request Endpoint

HISTORY: The 7th section of ID2 showed an idea about pre-registration of an authorization request. The section named the endpoint for the pre-registration "request object endpoint". The specification of PAR was developed based on the idea. As a result, the FAPI Final version has withdrawn the 7th section.

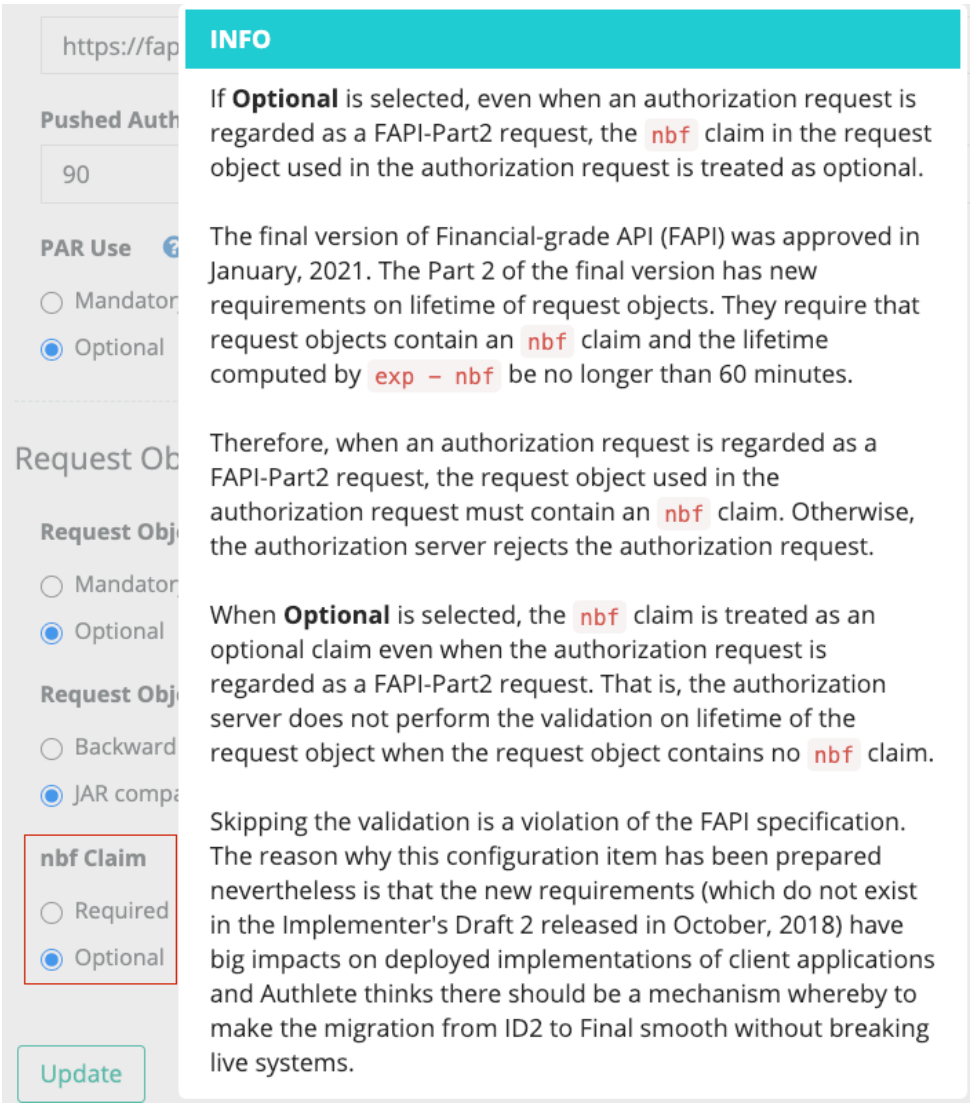
> ***Part 2: 5.2.2. Authorization server, 12.***
>
> *(withdrawn)*
>
> ***Part 2: 5.2.2. Authorization server, 13.***
>
> *shall require the request object to contain an* `exp` *claim that has a lifetime of no longer than 60 minutes after the* `nbf` *claim;*

OIDC Core does not require that request objects include the `exp` claim. In contrast, FAPI Part 2 requires `exp` as a mandatory claim.

Furthermore, the Final version has added a requirement *"**a lifetime of no longer than 60 minutes after the `nbf` claim**"*. Because of this requirement, the `nbf` claim has become mandatory.

The new requirement is a breaking change from a viewpoint of client applications because authorization servers now reject authorization requests whose request object does not include the `nbf` claim. As a matter of fact, some test cases in the official conformance suite had to be updated for the new requirement.

Authorization server implementations may provide a mechanism to mitigate the impact of the breaking change. For example, Authlete has defined `Service.nbfOptional` flag that indicates whether the `nbf` claim in the request object is optional even when the authorization request is regarded as a FAPI-Part2 request. The value of the flag can be changed by "nbf Claim" in the Service Owner Console.



Service Configuration: nbf Claim

> ***Part 2: 5.2.2. Authorization server, 14.***
>
> *shall authenticate the confidential client using one of the following methods (this overrides FAPI Security Profile 1.0 - Part 1: clause 5.2.2-4):*
>
> *1.* `tls_client_auth` *or* `self_signed_tls_client_auth` *as specified in section 2 of MTLS, or*
>
> *2.*`private_key_jwt` *as specified in section 9 of OIDC;*

It should be noted that `client_secret_jwt` is not allowed in Part 2. This is different from Part 1.

Client Authentication Method	Part 1	Part 2
client_secret_basic	✗	✗
client_secret_post	✗	✗
client_secret_jwt	○	✗
private_key_jwt	○	○
tls_client_auth	○	○
self_signed_tls_client_auth	○	○

Client Authentication Methods in FAPI

> ***Part 2: 5.2.2. Authorization server, 15.***
>
> *shall return the* `aud` *claim in the request object to be, or to be an array containing, the OP's Issuer Identifier URL;*

This requirement was added by the Final version. Client applications have to put the `aud` claim in request objects. The value of *"**OP's Issuer Identifier URL**"* can be found in the discovery document as the value of the `issuer` metadata (cf. OpenID Connect Discovery 1.0, 3. OpenID Provider Metadata).

Part 2: 5.2.2. Authorization server, 16.

shall not support public clients;

This requirement is a new one added by the Final version, but it is said that it has been logically impossible to support public clients in the context of FAPI Part 2 since older FAPI versions.

Part 2: 5.2.2. Authorization server, 17.

shall require the request object to contain an `nbf` claim that is no longer than 60 minutes in the past; and

The 13th requirement implies that the `nbf` claim is mandatory. This 17th requirement states it explicitly.

Part 2: 5.2.2. Authorization server, 18.

shall require PAR requests, if supported, to use PKCE (RFC7636) with `S256` as the code challenge method.

"PAR" here is "OAuth 2.0 Pushed Authorization Requests".

5.2.2.1. ID Token as detached signature

In addition, if the `response_type`value `code id_token` is used, the authorization server.

Section 5.2.2.1. lists requirements for authorization servers which are applied when an ID token is used as a detached signature.

5.2.2.1. ID Token as detached signature, 1.

shall support OIDC;

5.2.2.1. ID Token as detached signature, 2.

shall support signed ID Tokens;

5.2.2.1. ID Token as detached signature, 3.

should support signed and encrypted ID Tokens;

From a viewpoint of OIDC, these requirements are not new. By definition, ID tokens are always signed. Encryption of ID tokens is optional.

Part 2: 5.2.2.1. ID Token as detached signature, 4.

shall return ID Token as a detached signature to the authorization response;

This requires that an authorization server issue an ID token, but because the condition written at the top of Section 5.2.2.1 requires that `id_token` be included in `response_type` and so an ID token is issued as a general consequence, this requirement doesn't have to exist.

Part 2: 5.2.2.1. ID Token as detached signature, 5.

shall include state hash, `s_hash`, in the ID Token to protect the `state` value if the client supplied a value for `state`. `s_hash` may be omitted from the ID Token returned from the Token Endpoint when `s_hash` is present in the ID Token returned from the Authorization Endpoint; and

When JARM is used, this requirement doesn't have to be followed.

Part 2: 5.2.2.1. ID Token as detached signature, 6.

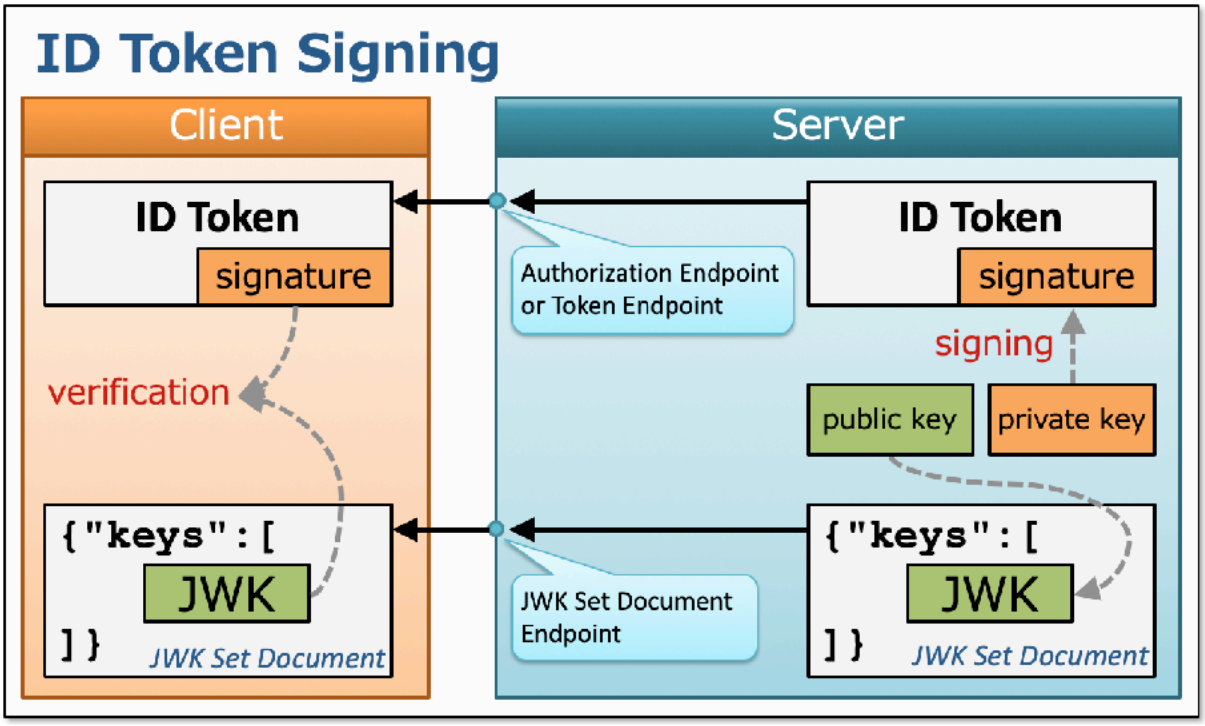
should not return sensitive PII in the ID Token in the authorization response, but if it needs to, then it should encrypt the ID Token.

PII is short for "Personally Identifiable Information".

The feature of ID token encryption has existed since OIDC Core. When the encryption algorithm for ID tokens is an asymmetric one, the authorization server must either (1) manage public keys of client applications directly in its database or

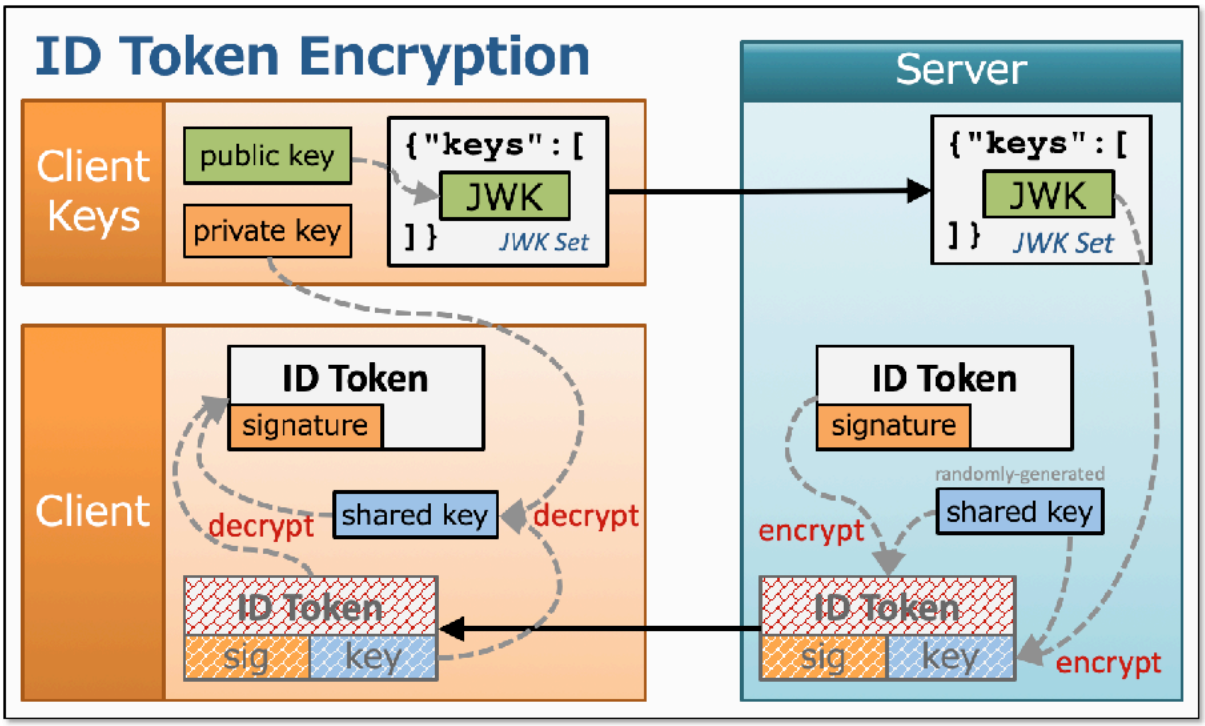
(2) fetch JWK Set documents from the locations pointed to by clients' `jwks_uri` metadata and extract public keys from the documents.

For signing ID tokens, it is server-side keys only that an authorization server has to handle.



ID Token Signing

In contrast, if an authorization server wants to support encryption of ID tokens, the authorization server has to handle client-side keys, too.



ID Token Encryption

This is the reason that not a small number of authorization server implementations don't support ID token encryption.

5.2.2.2. JARM

In addition, if the `response_type` value `code` is used in conjunction with the `response_mode` value `jwt`, the authorization server

5.2.2.2. JARM, 1.

shall create JWT-secured authorization responses as specified in JARM, Section 4.3.

This clause does not include any FAPI-specific requirements. It just says that JARM implementations should function as the JARM specification requires.

When `response_type` does not contain `id_token`, the authorization response will include no ID token. Therefore, an ID token cannot be used as a detached signature.

In this case, JARM has to be used to assure that the authorization response has not been tampered.

Requirements for Confidential Client

The FAPI Final version has renamed Part 2: Section 5.2.3 from "Public client" to "Confidential client".

> ***Part 2: 5.2.3. Confidential client, 1.***
>
> *shall support MTLS as mechanism for sender-constrained access tokens;*

That is, the authorization server must issue certificated-bound access tokens as defined in Section 3 of RFC 8705.

> ***Part 2: 5.2.3. Confidential client, 2.***
>
> *shall include the* `request` *or* `request_uri` *parameter as defined in Section 6 of OIDC in the authentication request;*

As listed in the list of requirements for authorization servers, either the `request` parameter or the `request_uri` parameter must be included. Note that OIDC Core says "***If one of these parameters is used, the other MUST NOT be used in the same request***."

> ***Part 2: 5.2.3. Confidential client, 3.***
>
> *shall ensure the Authorization Server has authenticated the user to an appropriate Level of Assurance for the client's intended purpose;*

This requirement states just that the user shall be authenticated appropriately. The FAPI Final version removed the requirement "***by requesting the `acr` claim as an essential claim***" which once existed in the clause.

There is a long history on this requirement.

In ID2, this requirement was ***"shall request user authentication at LoA 3 or greater by requesting the `acr` claim as an essential claim as defined in section 5.5.1.1 of [OIDC];".***

When a client wants to require claims as essential ones, the `acr_values` request parameter cannot be used. Instead, a client must use the `claims` request parameter, pass JSON as its value, and include `{"essential":true}` inside the JSON. The following is an example of JSON that needs to be given as the value of the `claims` request parameter in order to mark `urn:mace:incommon:iap:silver` as an essential ACR.

```
{
  "id_token":
    {
      "acr":
        {
          "essential": true,
          "values": ["urn:mace:incommon:iap:silver"]
        }
    }
}
```

Claims for Essential ACR

BTW, this requirement is loosened in **UK Open Banking** which is based on FAPI Part 2. That is, clients don't have to require ACRs as essential. Probably, it is not intentional. I guess that the snapshot of FAPI specification which was referred to when **Open Banking Profile** (**OBP**) was developed didn't contain the sentence, ***"by requesting the `acr` claim as an essential claim".***

The official Financial-grade API conformance test suite (conformance-suite) developed and maintained by FinTechLabs.io contains test cases for OBP. When FinTechLabs ran the OBP test cases using Authlete to test the test suite itself, they encountered an error. Because Authlete strictly follows FAPI specification, Authlete reported ***"`acr` claim is not required as essential."*** However, the expected behavior in the context of OBP is to ignore the FAPI requirement.

The right approach for the error was to amend OBP (to make OBP compliant with the latest FAPI specification). However, I was given explanation like "***if the official conformance test suite did it, all the existing OBP implementations wouldn't be able to pass the official tests. Changing the tests at this timing might cause delay in the officially-announced schedule of Open Banking.***"

Therefore, I decided to tweak Authlete and added OPEN_BANKING option in addition to FAPI option.

Supported Service Profiles (in Service Owner Console provided by Authlete)

If OPEN_BANKING is enabled, Authlete dare not to check if the `acr` claim is required as essential even in the context of FAPI Part 2. The code snippet below is the actual implementation excerpted from Authlete's source code.

```
private boolean isAcrEssentialRequired()
{
    boolean required = false;

    required |= mFapiRequirement.isAcrEssentialRequired();

    // Open Banking Profile (OBP) is based on FAPI, but the specification
    // does not require that the 'acr' claim be requested as essential.
    // We here intentionally disable the check for the value of 'essential'
    // when OBP is enabled because otherwise the official conformance test
    // suite for OBP would fail.
    if (isOpenBankingProfileEnabled())
    {
        required = false;
    }

    return required;
}
```

Code to judge whether acr should be required as an essential claim

As a result of this, Authlete is listed as a platform vendor that has passed "Open Banking Security Profile Conformance".

Authlete (Platform Vendor)	v1.1.2	v2.0.4	mtls	code id_token	29 Aug 2018	Download	PASS

Authlete listed in Open Banking Security Profile Conformance

However, again, the FAPI Final has removed the requirement *"**by requesting the `acr` claim as an essential claim**"*, so Authlete no longer checks whether ACRs are requested as essential ones. Therefore, the flag OPEN_BANKING is not meaningful any more.

Part 2: 5.2.3. Confidential client, 4.

(moved to 5.2.3.1);

Part 2: 5.2.3. Confidential client, 5.

(withdrawn);

Part 2: 5.2.3. Confidential client, 6.

(withdrawn);

Part 2: 5.2.3. Confidential client, 7.

(moved to 5.2.3.1);

Part 2: 5.2.3. Confidential client, 8.

shall send all parameters inside the authorization request's signed request object

Part 2: 5.2.3. Confidential client, 9.

shall additionally send duplicates of the `response_type`, `client_id`, *and* `scope` *parameters/values using the OAuth 2.0 request syntax as required by Section 6.1 of the OpenID Connect specification if not using PAR;*

If request parameters are all put into a request object, either the `request` parameter or the `request_uri` parameter is sufficient. However, if parameters that are mandatory in OAuth 2.0 / OIDC Core (e.g. `client_id` and `response_type`) are omitted, the request is no longer compliant with OAuth 2.0 / OIDC Core. Therefore, parameters that are mandatory in OAuth 2.0 / OIDC Core must be put outside the request object duplicately even if they exist inside the request object.

The FAPI Final version has added a condition *"**if not using PAR**"*. This implies that the set of request parameters don't have to be compliant with OAuth 2.0 / OIDC when PAR is used. This incompatibility comes from **JWT Secured Authorization Request** (**JAR**). See "**Implementer's note about JAR (JWT Secured Authorization Request)**" for details.

response_type	OAuth 2.0	OIDC	JAR
response_type outside request object	mandatory	mandatory	ignored
response_type inside request object	optional	optional; if present, it must match the outside one.	mandatory

response_type requirement in OAuth 2.0, OIDC and JAR

Part 2: 5.2.3. Confidential client, 10.

shall send the `aud` *claim in the request object as the OP's Issuer Identifier URL;*

Part 2: 5.2.3. Confidential client, 11.

shall send the `exp` *claim in the request object that has a lifetime of no longer than 60 minutes;*

The same requirements can be found in Section 5.2.2. Authorization server.

Part 2: 5.2.3. Confidential client, 12.

(moved to 5.2.3.1);

Part 2: 5.2.3. Confidential client, 13.

(moved to 5.2.3.1);

Part 2: 5.2.3. Confidential client, 14.

shall send a `nbf` claim in the request object;

The same requirement can be found in Section 5.2.2. Authorization server.

Part 2: 5.2.3. Confidential client, 15.

shall use RFC7636 with `S256` as the code challenge method if using PAR; and

That is, an authorization request must include `code_challenge_method=S256` request parameter when PAR is used.

Part 2: 5.2.3. Confidential client, 16.

shall additionally send a duplicate of the `client_id` parameter/value using the OAuth 2.0 request syntax to the authorization endpoint, as required by Section 5 of JAR, if using PAR.

The PAR specification requires that authorization servers handle request objects based on the rules defined in JAR. The JAR specification has made the `response_type` request parameter optional, but the `client_id` remains mandatory. See "**Implementer's note about JAR (JWT Secured Authorization Request)**" for details.

Part 2: 5.2.3.1. ID Token as detached signature

In addition, if the `response_type` value `code id_token` is used, the client

Section 5.2.3.1. lists requirements for client applications which are applied when an ID token is used as a detached signature.

Part 2: 5.2.3.1. ID Token as detached signature, 1.

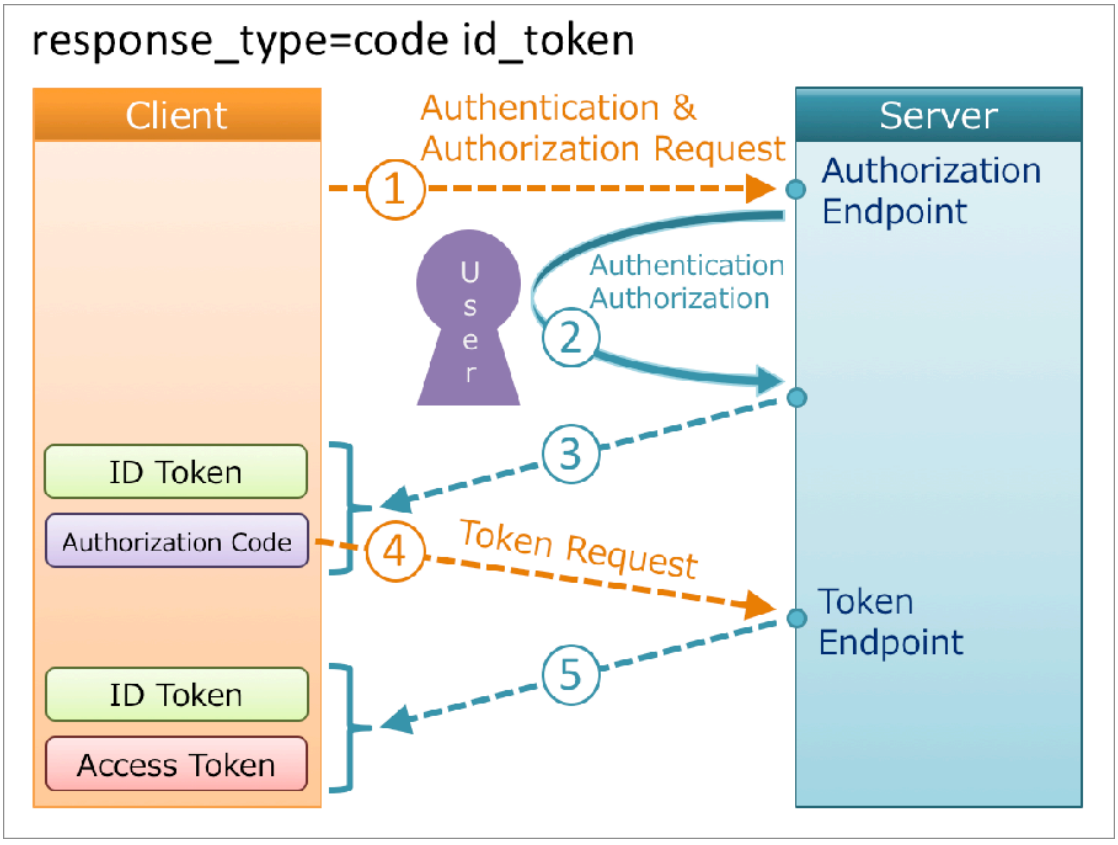
shall include the value `openid` *into the* `scope` *parameter in order to activate OIDC support;*

This is not a FAPI-specific requirement. OIDC Core requires that an OIDC request include `openid` in the `scope` parameter. See the explanation about the `scope` parameter written in Section 3.1.2.1. Authentication Request in OIDC Core for details.

Part 2: 5.2.3.1. ID Token as detached signature, 2.

shall require JWS signed ID Token be returned from endpoints;

Nothing new from OIDC's viewpoint. By definition, ID tokens are always signed. And when `response_type` is `code id_token` and `scope` contains `openid`, both the authorization endpoint and the token endpoint return an ID token. See "**Diagrams of All The OpenID Connect Flows**" for details about what the endpoints return.



response_type=code id_token

> ***Part 2: 5.2.3.1. ID Token as detached signature, 3.***
>
> *shall verify that the authorization response was not tampered using ID Token as the detached signature;*

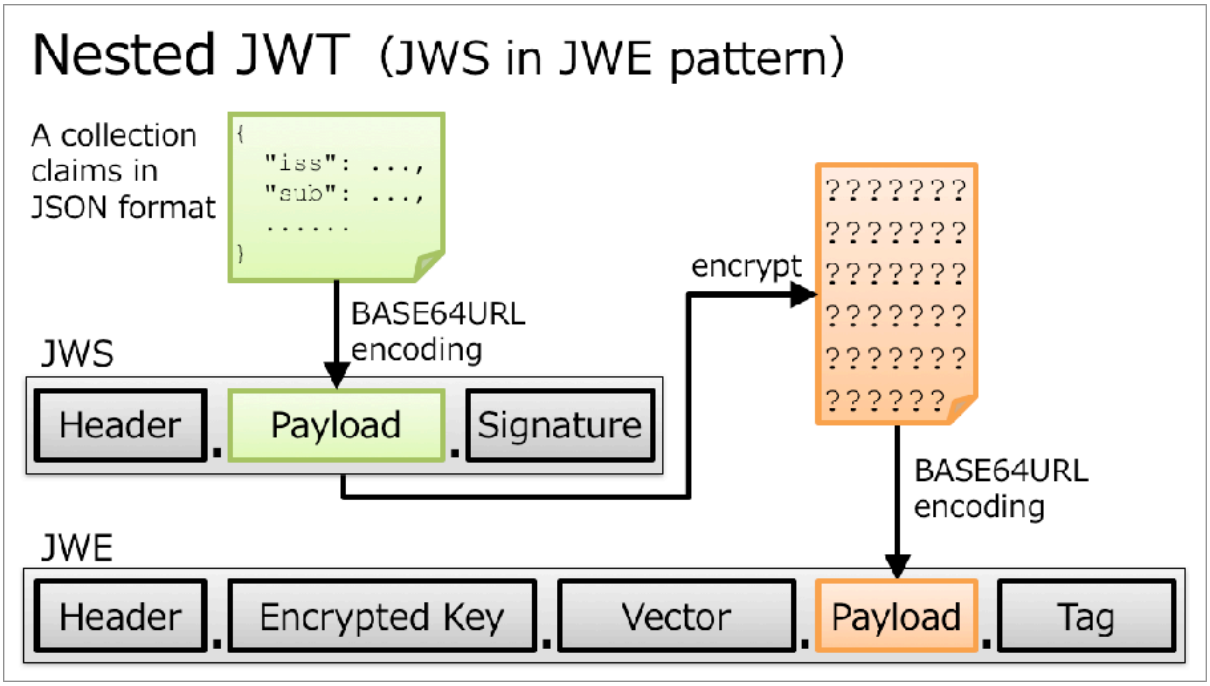
That is, client applications have to compute hash values of response parameters outside the issued ID token and compare the values to the hash values in the ID token.

> ***Part 2: 5.2.3.1. ID Token as detached signature, 4.***
>
> *shall verify that* `s_hash` *value is equal to the value calculated from the* `state` *value in the authorization response in addition to all the requirements in 3.3.2.12 of OIDC; and*
>
> *NOTE: This enables the client to verify that the authorization response was not tampered with, using the ID Token as a detached signature.*

This requirement particularly mentions the `state` parameter and the `s_hash` claim in the ID token although they are just one of parameter/hash pairs that have to be considered.

> ***Part 2: 5.2.3.1. ID Token as detached signature, 5.***
>
> *shall support both signed and signed & encrypted ID Tokens.*

By definition, ID tokens are always signed. When ID tokens are encrypted, the order of signing and encrypting is "sign then encrypt". As a result, an encrypted ID token takes the form of "Nested JWT" as illustrated below.



Nested JWT (JWS in JWE pattern)

See "**Understanding ID Token**" for details about the structure of ID tokens.

Part 2: 5.2.3.2. JARM

In addition, if the `response_type` *value* `code` *is used in conjunction with the* `response_mode` *value* `jwt`*, the client*

Part 2: 5.2.3.2. JARM, 1.

shall verify the authorization responses as specified in JARM, Section 4.4.

See "Section 4.4. Processing rules" of JARM for details about the verification steps.

Part 2: 5.2.4.

(withdrawn)

Part 2: 5.2.5.

(withdrawn)

Part 2: 6.2.1. Protected resource provisions, 1.

shall support the provisions specified in clause 6.2.1 Financial-grade API Security Profile 1.0 - Part 1: Baseline; and

Part 2: 6.2.1. Protected resource provisions, 2.

shall adhere to the requirements in MTLS.

Part 2: 6.2.2. Client provisions

The client supporting this document shall support the provisions specified in clause 6.2.2 of Financial-grade API Security Profile 1.0 - Part 1: Baseline.

Simply put, Section 6 of Part 2 states that protected resource endpoints and client applications shall use certificate-bound access tokens and follow requirements in Part 1.

Part 2: 7. *(Withdrawn)*

The 7th section of ID2 was "Request object endpoint". The section was removed by the FAPI Final version because it was replaced with "OAuth 2.0 Pushed Authorization Requests" (PAR). See "Illustrated PAR: OAuth 2.0 Pushed Authorization Requests" for overview of PAR.

Security Considerations

"8. Security considerations" of "Part 2" lists security considerations. Summary is as follows.

8.1 — This specification references security considerations in Section 10 of RFC 6749 and RFC 6819.

8.2 — Protected resource endpoints shall accept only certificate-bound access tokens.

8.3.1 — Clients should use a different redirect URI per authorization server.

8.3.2 — Authorization codes and client secrets are passed to attackers if developers are deceived into using a fake token endpoint.

8.3.3 — Hybrid flow or JARM can be used as a countermeasure for IdP mix-up attack.

8.3.4 — (removed)

8.3.5 — Because an access token is bound to an X.509 certificate, stolen access tokens cannot be used without corresponding certificates.

8.4.1 — RFC 6749 doesn't assure message integrity of authorization request and response.

8.4.2 — Using request objects prevents authorization request parameter injection attack.

8.4.3 — Using hybrid flow or JARM prevents authorization response parameter injection attack.

8.5 — Cipher suites for TLS 1.2 are restricted.

"8.5. TLS considerations" of "Part 2" permits only the following cipher suites for TLS communication when the TLS version in use is below 1.3.

1. `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`
2. `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
3. `TLS_DHE_RSA_WITH_AES_256_GCM_SHA384`
4. `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

But, from a viewpoint of interoperability of web browsers, additional cipher suites allowed in the latest BCP 195 are permitted for authorization endpoints.

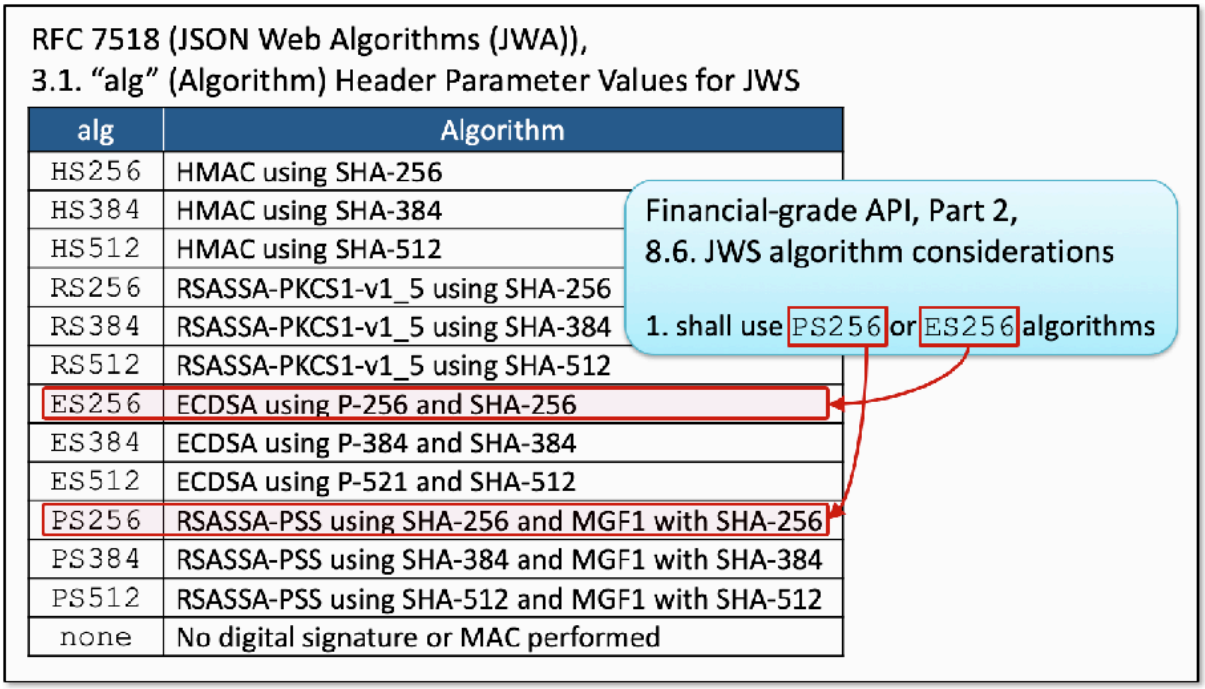
Because I couldn't find any good reasons to exclude the following cipher suites,

5. `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
6. `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`

I created Issue 216 (TLS_ECDHE_ECDSA cipher suites) to suggest adding them to the list of permitted cipher suites. 1 year and 4 months later, the issue was closed with the reason that FAPI now allows TLS 1.3.

8.6 — `PS256` and `ES256` only are allowed for JWS signature algorithm.

Signing algorithms of JWS are listed in "3.1. "alg" (Algorithm) Header Parameter Values for JWS" of RFC 7518 (JSON Web Algorithms). Among the 13 algorithms, "8.6. Algorithm considerations" of "Part 2" permits `PS256` and `ES256` only. The section explicitly states that RSASSA-PKCS1-v1_5 (e.g. `RS256`) should not be used and `none` must not be used.



JWS algorithms permitted by Financial-grade API, Part 2

FYI: JWT is used at the following places in an authorization server implementation.

specification	usage
RFC 7523	`client_assertion` parameter
OIDC Core, 2	ID token
OIDC Core, 6	request object
OIDC Core, 5.3	UserInfo response
JARM	authorization response
CIBA Core, 7.1.1	backchannel authentication request

JWT Usage in Authorization Server Implementation

8.6.1 — `RSA1_5` encryption algorithm must not be used.

This requirement about encryption algorithms was added by the FAPI Final version. FAPI prohibits `RSA1_5`. The algorighm identifier is defined in "4.1. "alg" (Algorithm) Header Parameter Values for JWE" of RFC 7518 (JSON Web Algorithms). The identifier represents RSAES-PKCS1-v1_5.

8.7 — Use certified FAPI implementations.

8.8 —Don't allow privileged actions without an access token.

8.9 — Keys for signature verification should be accessible via the `jwks_uri` or `jwks` client metadata (cf. RFC 7591) and the `jwks_uri` server metadata (cf. RFC 8414).

8.10 — A compromise of any client that shares the same key with other clients would result in a compromise of all the clients.

8.11 — JWK sets should not contain multiple keys with the same `kid`, but other key attributes may be used to select one among multiple key candidates.

How Authlete Implements FAPI

This chapter picks up some topics related to FAPI implementation.

Baseline or Advanced?

When a client application requests an access token and accesses APIs with the access token, which security profile should apply, FAPI Part 1 or FAPI Part 2, or neither of them?

Some implementations may configure themselves statically and others may make the decision dynamically at runtime. **The FAPI specification mentions nothing about how to determine which security profile should apply.**

A simple approach would be *"**Regard all authorization requests as FAPI Part 2 requests.**"* Actually, UK Open Banking has adopted this approach. A hard-coded implementation like this may be acceptable if the system development is a one-time work.

However, this approach is not appropriate for a generic authorization server implementation. It's because a hard-coded implementation hinders flexibility of system design too much. Therefore, in a generic implementation, it is better to judge dynamically at runtime whether an authorization request is for FAPI Part 1 or for FAPI Part 2 (or for normal OAuth 2.0 / OIDC).

If so, how to judge dynamically? The conclusion everyone will eventually reach after thinking will be just one. It is **judged by checking the requested scopes**.

(Note: Another possible way would be to utilize the `resource` request parameter defined in "RFC 8707 Resource Indicators for OAuth 2.0".)

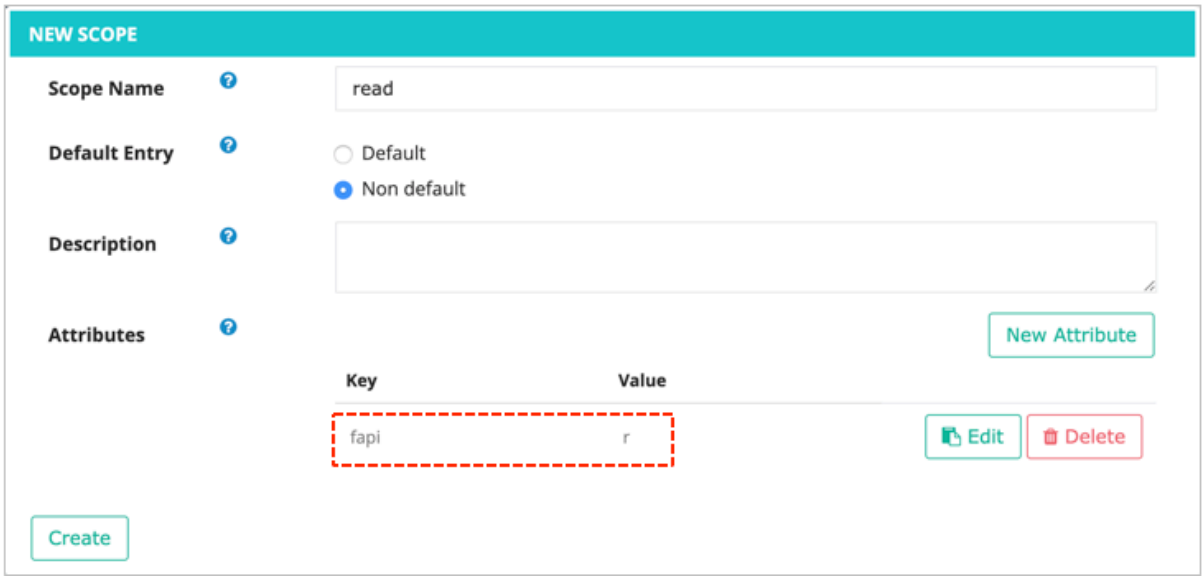
For example, (1) prepare scopes named `read` and `write`, (2) adopt a rule where the `read` scope requires FAPI Part 1 requirements be satisfied and the `write` scope requires FAPI Part 2 requirements be satisfied, and (3) implement APIs so that they interpret the scopes accordingly. If APIs are implemented in this way, the implementation of an authorization endpoint can change its behavior dynamically by (a) applying FAPI Part 2 requirements when the `scope` request parameter includes the `write` scope, (b) applying FAPI Part 1 requirements when the `scope`

request parameter does not include the `write` scope but includes the `read` scope, and (c) applying normal OAuth 2.0 and OIDC requirements when the `scope` request parameter includes neither the `read` scope nor the `write` scope.

How to implement the scope-based switch? For instance, one approach might be to regard scopes whose name starts with `read` as scopes for FAPI Part 1. However, this approach imposes heavy restrictions on scope names. If that is the case, what approach has Authlete adopted?

As the first step, Authlete implemented a generic mechanism to set arbitrary attributes to each scope. On the mechanism, Authlete treats the attribute name `fapi` in a special way. An attribute having name `fapi` and value `r` represents Read-Only (= Baseline). Likewise, an attribute having name `fapi` and value `rw` represents Read-and-Write (Advanced).

The web console for FAPI-aware Authlete (version 2.0+) provides UI for scope attributes. The screenshot below defines a scope named `read` with an attribute of `fapi=r`.



Scope Settings for FAPI Read-Only

Authlete's /auth/authorization API that parses an authorization request checks scopes listed in the `scope` request parameter in the authorization request and regards the request as a request for FAPI Part 2 if the scope list includes a scope that has an attribute of `fapi=rw`. If the scope list does not include any scope having an attribute of `fapi=rw` but includes a scope having an attribute of `fapi=r`, the

authorization request is regarded as a request for FAPI Part 1. In other cases, the authorization request is treated as a normal OAuth 2.0 / OIDC request.

NOTE: In ID2, the names of FAPI Part 1 and Part 2 were “Read-Only Security Profile” and “Read and Write Security Profile”. The FAPI Final version renamed them to “Baseline Security Profile” and “Advanced Security Profile”. The values of `r` and `rw` for the `fapi` attribute were determined based on the old names.

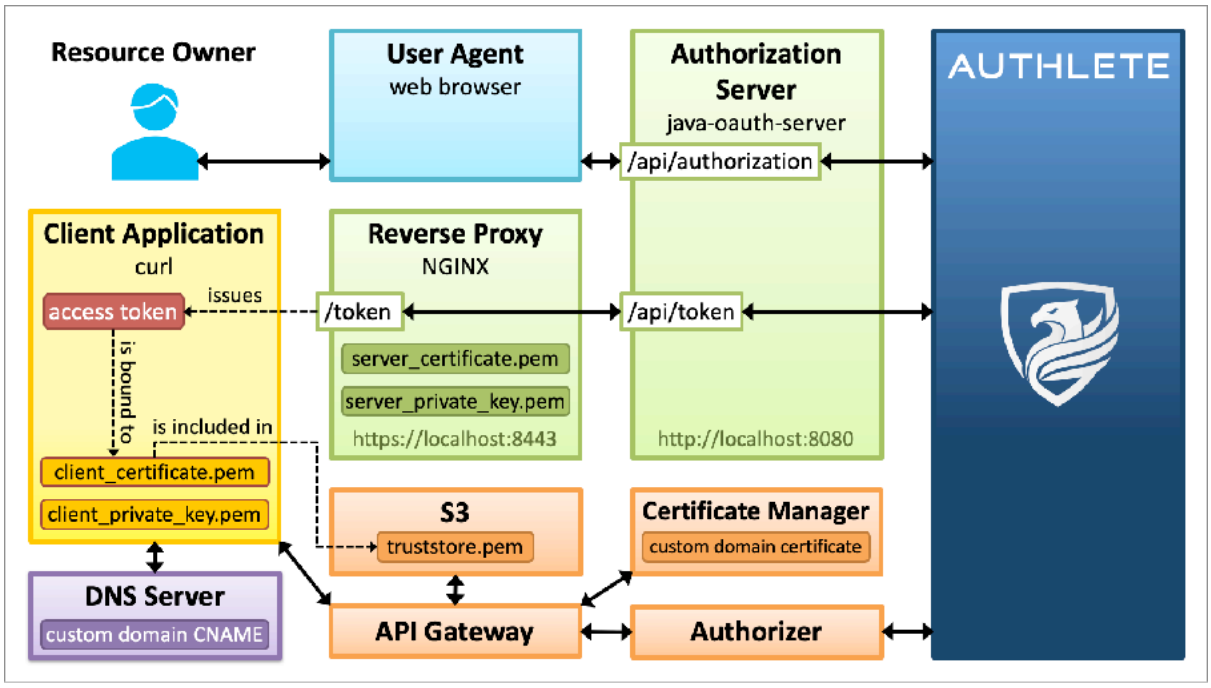
Mutual TLS

"Mutual TLS" has three meanings as listed below (as already explained previously).

1. TLS communication using a client certificate
2. Client authentication using a client certificate
3. Certificate binding

The first part is handled by API management solutions. On the other hand, the second and the third parts don't necessarily have to be handled by the API management layer. Rather, a better system architecture would handle them in a different layer that is independent of the API management layer.

Because of its unique architecture, Authlete doesn't take on any task in the API management layer. That is, Authlete does nothing for the first part. On the other hand, Authlete supports the second and the third parts. Thanks to this, with Authlete, systems can support MTLS (RFC 8705 OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens) required by FAPI on any API management solution that developers want to use. I actually tried MTLS on Amazon API Gateway and wrote an article titled "**Financial-grade Amazon API Gateway**" to explains how to achieve it.



Example of Component Deployment for MTLS on Amazon API Gateway

Any API management solution can support MTLS by using Authlete as long as the solution provides a mechanism which enables developers to access the client certificate used in TLS communication.

Existing API management solutions may try to implement MTLS directly. However, it would take time, and above all, it is not a good system design to support the functionality directly in the API management layer. At the time of this writing, if you use an API management solution provided by one of giant cloud vendors, Authlete is the best answer for MTLS.

The video below is a session in "Financial APIs Workshop" that took place in Tokyo on July 24, 2018. In the video, Justin Richer, one of the most famous software engineers in the community and the author of "OAuth 2 in Action", is explaining Authlete's MTLS implementation.

The material and transcript of the presentation are available at "Authlete FAPI Enhancements".

Access Token Duration

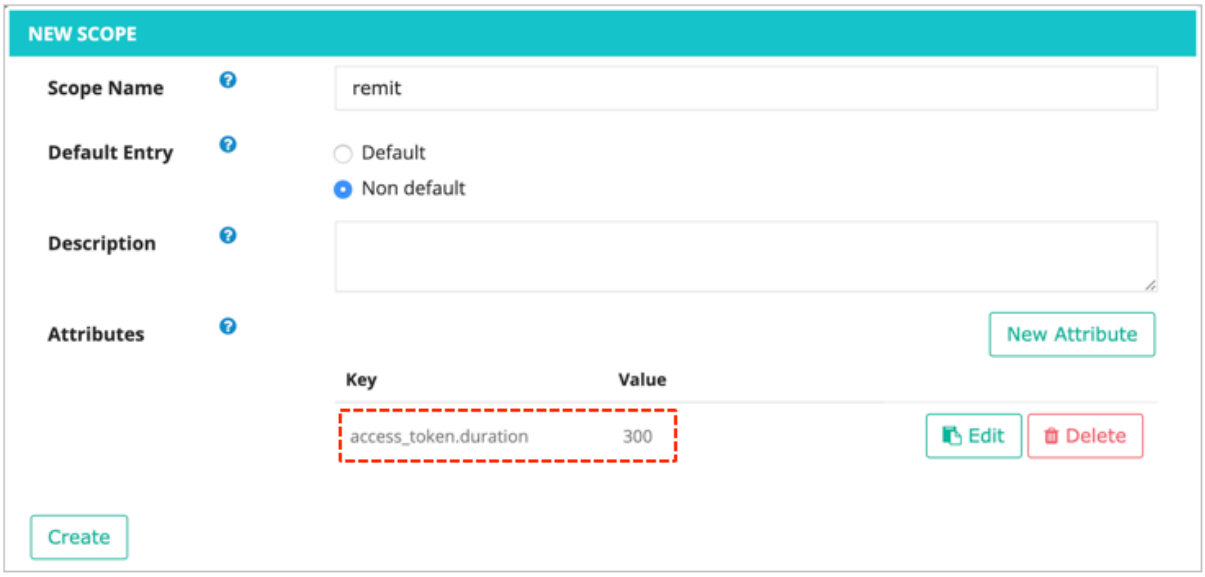
This is not related to FAPI, but I explain this feature here because I'm often consulted about the feature in the context of Bank API by customers who want to make duration of access tokens for remittance shorter than that of access tokens for other purposes.

The functionality can be achieved by making access token duration shorter when an authorization request contains a scope for remittance. For example, if an API for remittance requires a scope named `remit`, the authorization server would shorten access token duration when an authorization request contains the scope.

Authlete supports the functionality by treating a scope attribute named `access_token.duration` in a special way.

Authlete checks all scope attributes of requested scopes, picks up the smallest value among values of `access_token.duration` attributes, and uses it as the duration of an access token being issued. If none of the requested scopes has an `access_token.duration` attribute, Authlete uses the default value of access token duration set per authorization server instance. If the default value is smaller than the smallest value of `access_token.duration` attributes, the default value is used.

The screenshot below shows how to set `access_token.duration=300`as a scope attribute.



Scope Settings for Access Token Duration

Likewise, duration of refresh tokens can be set by utilizing `refresh_token.duration` attributes.

NOTE: Authlete 2.1 and newer versions support "access token duration per client". See "How Authlete determines token duration" on Authlete Knowledge Base for details.

Access Token with Transaction Information

This feature is not related to FAPI, either, but I explain it here as I'm often consulted about it in the context of Bank API by customers who want to associate detailed transaction information with an access token. I hear that some regulations in Europe require an access token be issued per transaction under some conditions.

This functionality cannot be achieved by "scope attribute" which was explained in "Access Token Duration" because the functionality requires data be handled per access token, not per scope.

Since old days, Authlete has provided a mechanism to set arbitrary key-value pairs to an access token. This feature can be utilized to associate transaction information with an access token. Technical details about this feature are explained in "**Extra Properties**". See also "How to add extra properties to an access token" in Authlete Knowledge Base.

However, note that it is not a smart way to associate detailed information such as amount of money with an access token directly. Instead, a recommended way is to (1) store detailed transaction information into another database and (2) associate the unique identifier of the database record with an access token.

Authorization Details

Since the version 2.2, Authlete supports "**OAuth 2.0 Rich Authorization Requests**" (**RAR**). The standard specification adds a request/response parameter, `authorization_details`.

The `authorization_details` parameter is used to enable an access token to hold detailed information about authorization. For example, detailed information about payment such as "How much?", "To whom?", etc.

According to the specification, the `authorization_details` parameter can be used anywhere the `scope` parameter is used. For instance, (a) in the authorization request, (b) in the token response, (c) in the introspection response, and so on.

```
GET /authorize?response_type=code
   &client_id=s6BhdRkqt3
   &state=af0ifjsldkj
   &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
   &code_challenge_method=S256
   &code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bwc-uCHaoeK1t8U
   &authorization_details=%5B%7B%22type%22%3A%22account%5Finfo
   rmation%22%2C%22actions%22%3A%5B%22list%5Faccounts%22%2C%22
   read%5Fbalances%22%2C%22read%5Ftransactions%22%5D%2C%22loca
   tions%22%3A%5B%22https%3A%2F%2Fexample%2Ecom%2Faccounts%22%
   5D%7D%2C%7B%22type%22%3A%22payment%5Finitiation%22%2C%22act
   ions%22%3A%5B%22initiate%22%2C%22status%22%2C%22cancel%22%5
   D%2C%22locations%22%3A%5B%22https%3A%2F%2Fexample%2Ecom%2Fp
   ayments%22%5D%2C%22instructedAmount%22%3A%7B%22currency%22%
   3A%22EUR%22%2C%22amount%22%3A%22123%2E50%22%7D%2C%22credito
   rName%22%3A%22Merchant123%22%2C%22creditorAccount%22%3A%7B%
   22iban%22%3A%22DE02100100109307118603%22%7D%2C%22remittance
   InformationUnstructured%22%3A%22RefNumberMerchant%22%7D%5D HTTP/1.1
Host: server.example.com
```

(a) authorization_details in Authorization Request (RAR Section 3)

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

{
   "access_token": "2YotnFZFEjr1zCsicMWpAA",
   "token_type": "example",
   "expires_in": 3600,
   "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
   "authorization_details": [
     {
         "type": "https://www.someorg.com/payment_initiation",
         "actions": [
            "initiate",
            "status",
            "cancel"
         ],
         "locations": [
            "https://example.com/payments"
         ],
         "instructedAmount": {
            "currency": "EUR",
            "amount": "123.50"
         },
         "creditorName": "Merchant123",
         "creditorAccount": {
            "iban": "DE02100100109307118603"
         },
         "remittanceInformationUnstructured": "Ref Number Merchant"
     }
   ]
}
```

(b) authorization_details in Token Response (RAR Section 7)

(c) authorization_details in Introspection Response (RAR Section 8.2)

RAR is an open standard to describe details about authorization and tie the information to an access token. RAR is to be adopted as a component of FAPI 2.0 (cf. "Are there FAPI 2.0 implementations?" in FAPI FAQ).

Authlete's Extra Properties can be used for the same purpose. One functional difference is that Extra Properties can choose to expose or hide extra properties. Hidden extra properties never appear in any OAuth/OIDC responses but can be retrieved by Authlete's introspection API (/auth/introspection API). There are some use cases where you want to tie information to an access token but hide the

information from the client application and the user. In such cases, Extra Properties is useful.

Conclusion

Authlete has already supports the FAPI 1.0 Final version and known technical specifications of FAPI 2.0 as mentioned in FAPI FAQ. You can try FAPI through Authlete API Tutorials with an Authlete account (signup).

Please feel free to contact us, Authlete, Inc., via the web form or email if you are interested in using the world's first certified FAPI implementation.

About Authlete

Authlete, Inc is based in Tokyo and London and comprised of a team of experts who have a wealth of experience specialized in authorization and identity management and are actively involved in providing specifications of open standards serving a variety of industries, such as UK Open Banking.

For more information, please visit www.authlete.com.