クライアント認証

Online Session on July 1, 2020

Co-founder, Authlete, Inc.
Takahiko Kawasaki <taka@authlete.com>
@darutk @authlete @authlete_jp

前提知識

- RFC 6749 (The OAuth 2.0 Authorization Framework)
- RFC 7515 ～ RFC 7519 (JWS, JWE, JWK, JWA, JWT)

大前提知識

HTTP, HTML, JSON,
x-www-form-urlencoded,
Base64, Base64URL,
公開鍵暗号の概念

OAuth & OIDC 勉強会【入門編】（前提知識おさらい）

https://www.authlete.com/ja/resources/videos/20200317/

https://www.youtube.com/watch?v=PKPj_MmLq5E

資料をダウンロードできます！

OAuth & OIDC 勉強会【アクセストークン編】

https://www.authlete.com/ja/resources/videos/20200422/

https://www.youtube.com/watch?v=8WPPrzL-FQc&list=PLxDcFnLrbxvafUBu2GtX35G-iiktmZhWa

OAuth & OIDC 勉強会【認可リクエスト編】

https://www.authlete.com/ja/resources/videos/20200527/

https://www.youtube.com/watch?v=8lEC2KfjN_E&list=PLxDcFnLrbxvZs5eWfxDFdPjrzZ2R7h2Uw

クライアントタイプ


OAuth 2.0
OpenID Connect
Authorization Focused • Reliable and Scalable • Developer Friendly
Faster Time to Market • Choice of Hosting Options • Broad Usage
Integrates with any Authentication methods
API Security
OpenID
AUTHLETE

RFC 6749, 2.1. Client Types

confidential

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

public

Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

クライアント認証情報（client credentials）の機密性を維持できるクライアントをコンフィデンシャルクライアント（confidential client）、維持することができないクライアントをパブリッククライアント（public client）と呼ぶ。

クライアントタイプとフロー利用可否の関係

フロー		クライアントタイプ	
		コンフィデンシャル	パブリック
RFC 6749	認可コード	○利用可能	○利用可能
	インプリシット	○利用可能	○利用可能
	リソースオーナー パスワードクレデンシャルズ	○利用可能	○利用可能
	クライアント クレデンシャルズ	○利用可能	×利用不可
RFC 8628	デバイス	○利用可能	○利用可能
CIBA	POLL	○利用可能	×利用不可
	PING	○利用可能	×利用不可
	PUSH	○利用可能	×利用不可

✓ 禁止されている組み合わせは次の二つのみ。

1. クライアントクレデンシャルズフロー ＋ パブリッククライアント
2. CIBA フロー (POLL/PING/PUSH) ＋ パブリッククライアント

✓ 「インプリシットフローを使うのはパブリッククライアントのみ」というのは誤解。

RFC 6749, 3.1.2.2. Registration Requirements

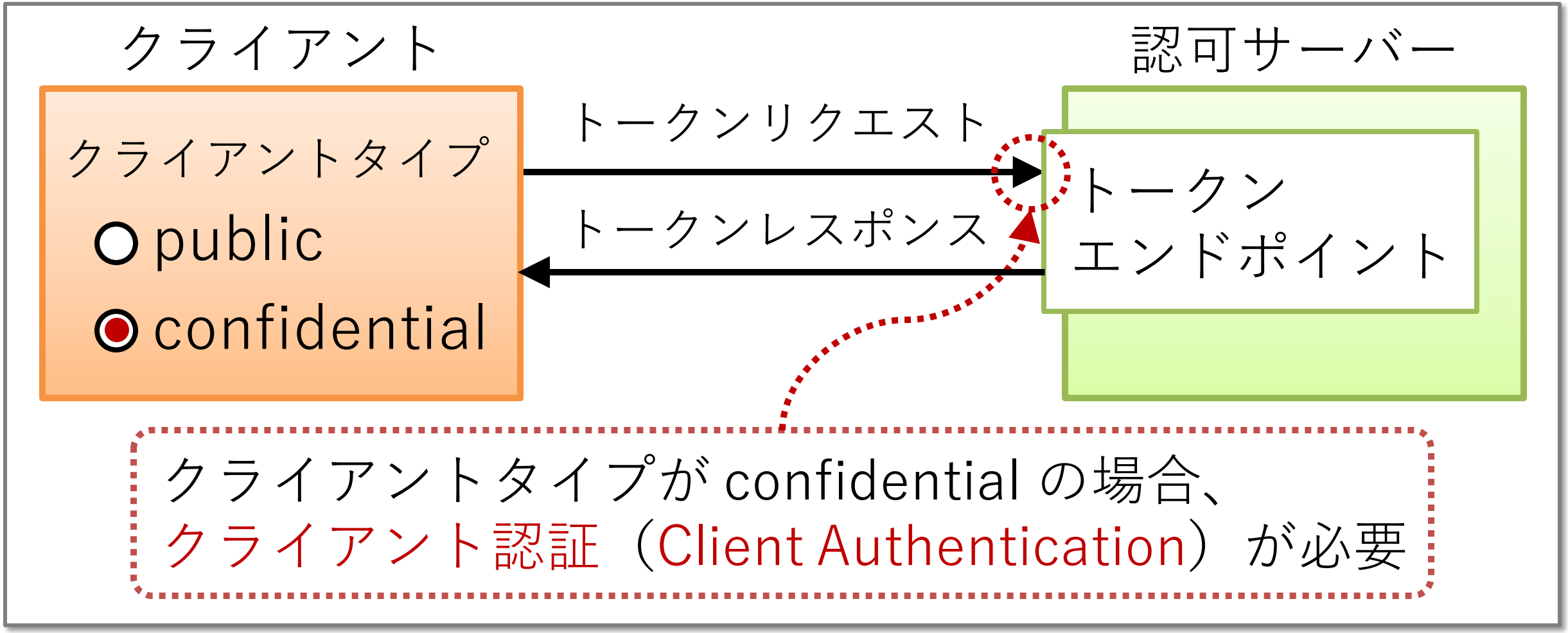
> The authorization server MUST require the following clients to register their redirection endpoint:
> - Public clients.
> - Confidential clients utilizing the implicit grant type.

✓ 一つのクライアントが複数のフローを使ってもよい。

OpenID Connect Dynamic Client Registration 1.0, 2. Client Metadata

> grant_types
> OPTIONAL. JSON array containing a list of the OAuth 2.0 Grant Types that the Client is declaring that it will restrict itself to using.

直接送信系クライアント認証



✓ ほとんどの場合、アクセストークンはトークンエンドポイントから発行される。

✓ 例外は、認可エンドポイントからアクセストークンが発行される場合（response_type に token が含まれる場合）と、CIBA の PUSH モード。

✓ トークンエンドポイント以外でもクライアント認証を要求されることがある。

1. RFC 7009　リボケーションエンドポイント
2. RFC 8628　デバイス認可エンドポイント
3. CIBA　　バックチャネル認証エンドポイント　← クライアント認証必須

◆ RFC 7662 や RFC 8414 を読むと、イントロスペクションエンドポイントでもクライアント認証が行われるように見えるが、やっていることはリソースサーバー認証であり、似て非なるものである。

詳細はこちら →

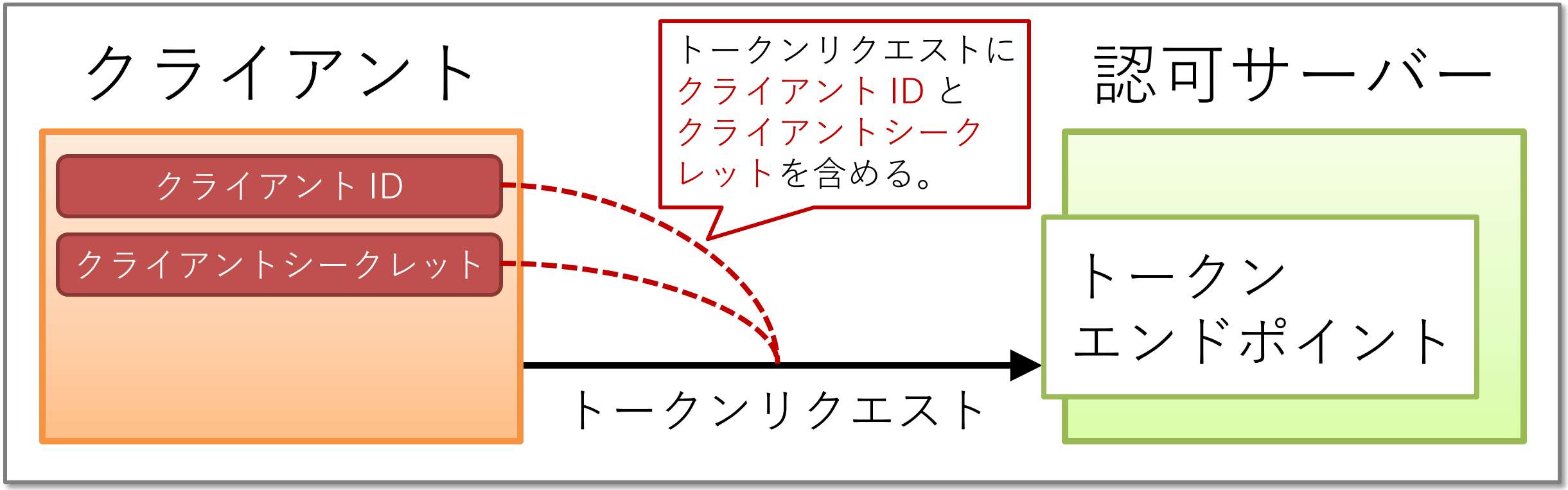
OAuth & OIDC 勉強会【アクセストークン編】#2 3:44～
https://www.youtube.com/watch?v=wAdVvJX9-Vw&t=224
https://www.authlete.com/ja/resources/videos/20200422/　（資料ダウンロード）

✓ 認可エンドポイントではクライアント認証は要求されない。

インプリシットフローは認可エンドポイントしか使わない。そのため、クライアントタイプがコンフィデンシャルでも、インプリシットフローではクライアント認証はおこなわれない。

➢ 「インプリシットフローを使うのはパブリッククライアントのみ」という誤解の一因

昔ながらのクライアント認証方式（RFC 6749 で言及されているもの）



- ✓ 認可サーバーがクライアントIDとクライアントシークレットを生成し、あらかじめクライアントアプリケーションに渡しておく。
- ✓ トークンリクエストにクライアントIDとクライアントシークレットを含める。

クライアントIDとクライアントシークレットによるクライアント認証その１

client_secret_post

✓ リクエストパラメーター client_id, client_secret を使う。

```
POST トークンエンドポイント HTTP/1.1
Host: 認可サーバー
Content-Type: application/x-www-form-urlencoded

client_id=クライアントID&
client_secret=クライアントシークレット&
（他のリクエストパラメーター群は省略）
```

クライアントIDとクライアントシークレットによるクライアント認証その 2

client_secret_basic

✓ BASIC 認証（RFC 7617）を使う。

クライアントID:クライアントシークレット

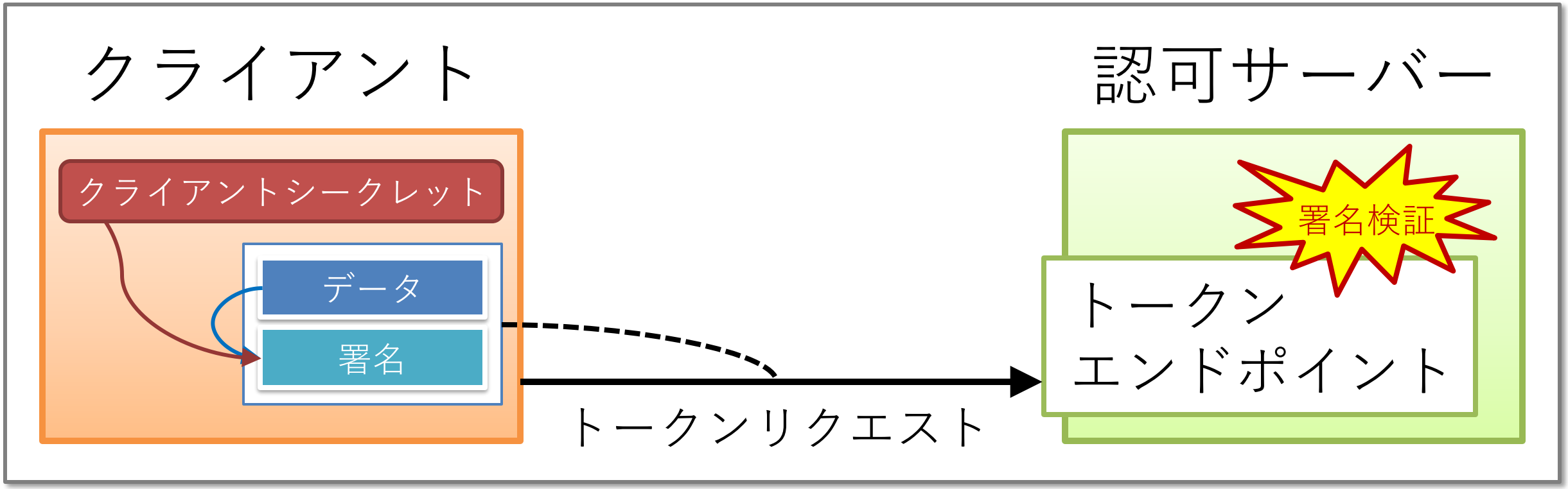
BASE64 エンコード

```
POST トークンエンドポイント HTTP/1.1
Host: 認可サーバー
Authorization: Basic BASE64エンコードされたクライアント認証情報
Content-Type: application/x-www-form-urlencoded

（リクエストパラメーター群は省略）
```

JWT系クライアント認証

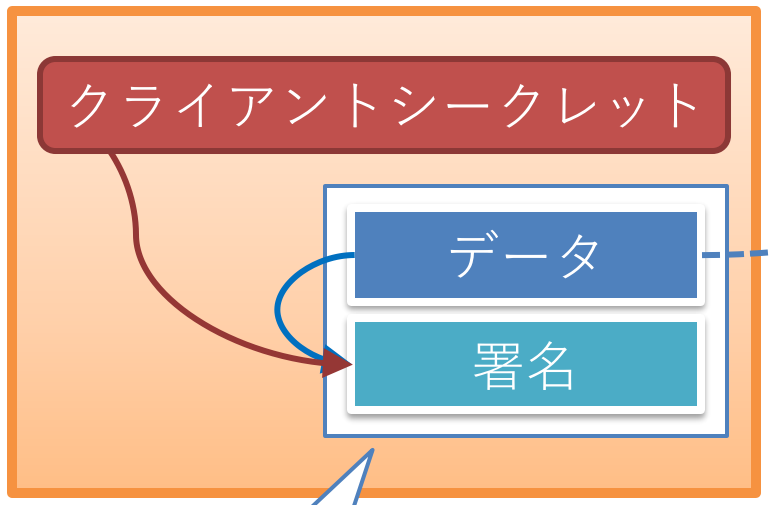
クライアントシークレットを直接トークンリクエストに含めなくても、
クライアントシークレットを持っていることを証明する方法がある。

✓ クライアントシークレットでデータに署名し、サーバー側でその署名を検証する。



署名対象となるデータの形式には標準仕様が存在する。

✓ RFC 7523 : JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants / 2.2. Using JWTs for Client Authentication




```
{
  "iss": "クライアントID",
  "sub": "クライアントID",
  "aud": "トークンエンドポイント",
  "jti": "JWT ID",
  "exp": 有効期限終了時刻,
  "iat": 発行時刻
}
```

クライアント認証で使われる JWT はクライアントアサーションと呼ばれる。

```
POST トークンエンドポイント HTTP/1.1
Host: 認可サーバー
Content-Type: application/x-www-form-urlencoded

client_assertion_type=
  urn:ietf:params:oauth:client-assertion-type:jwt-bearer&
client_assertion=クライアントアサーション&
 (他省略)
```

ペイロード部

```
{
  "iss": "クライアントID",
  "sub": "クライアントID",
  "aud": "トークンエンドポイント",
  "jti": "JWT ID",
  "exp": 有効期限終了時刻,
  "iat": 発行時刻
}
```

クライアントアサーションの aud について（1/2）

RFC 7523, 3. JWT Format and Processing Requirements, 3

> The JWT MUST contain an "aud" (audience) claim containing a value that identifies the authorization server as an intended audience. The token endpoint URL of the authorization server MAY be used as a value for an "aud" element to identify the authorization server as an intended audience of the JWT. The authorization server MUST reject any JWT that does not contain its own identity as the intended audience.

✓ RFC 7523 は aud の値としてトークンエンドポイントの URL を使っても良い（MAY）としている。
しかし、認可サーバーを audience として指定できるなら、それ以外も可。実装依存。

OpenID Connect Core 1.0, 9. Client Authentication

> aud
>
> REQUIRED. Audience. The aud (audience) Claim. Value that identifies the Authorization Server as an intended audience. The Authorization Server MUST verify that it is an intended audience for the token. The Audience SHOULD be the URL of the Authorization Server's Token Endpoint.

✓ OIDC Core は aud の値としてトークンエンドポイントの URL を使うべき（SHOULD）としている。

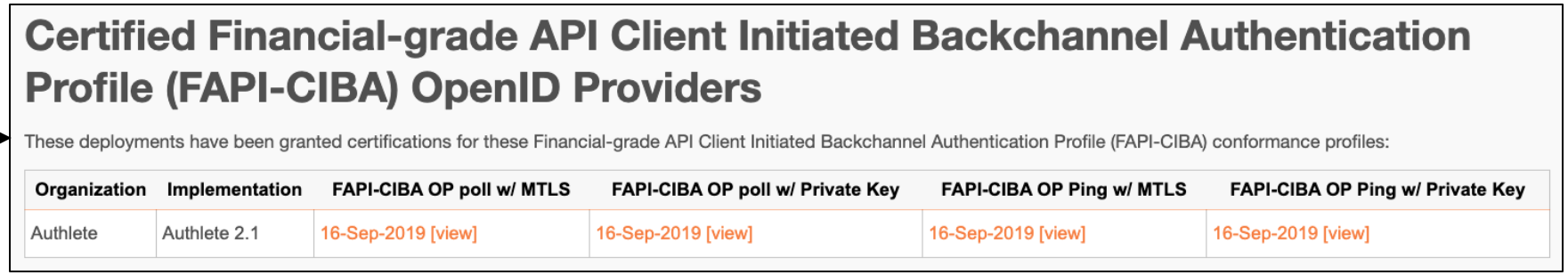
クライアントアサーションの aud について（2/2）

OpenID Connect Client Initiated Backchannel Authentication Flow – Core 1.0, 7.1. Authentication Request

> Note that there's some potential ambiguity around the appropriate audience value to use when JWT client assertion based authentication is employed. To address that ambiguity the Issuer Identifier of the OP SHOULD be used as the value of the audience. In order to facilitate interoperability the OP MUST accept its Issuer Identifier, Token Endpoint URL, or Backchannel Authentication Endpoint URL as values that identify it as an intended audience.

- ✓ CIBA Core は、バックチャネル認証エンドポイントで JWT 系クライアント認証を使う場合、クライアントアサーションの aud の値は OpenID Provider の発行者識別子（Issuer Identifier）とすべき（SHOULD）としている。
- ✓ 相互運用性のため、OpenID Provider は、発行者識別子、トークンエンドポイントの URL、バックチャネル認証エンドポイントの URL を、aud の値として受け入れなければならない（MUST）。

これ↑にちゃんと対応しないと**FAPI-CIBA プロファイル**の Certification を取得できない。

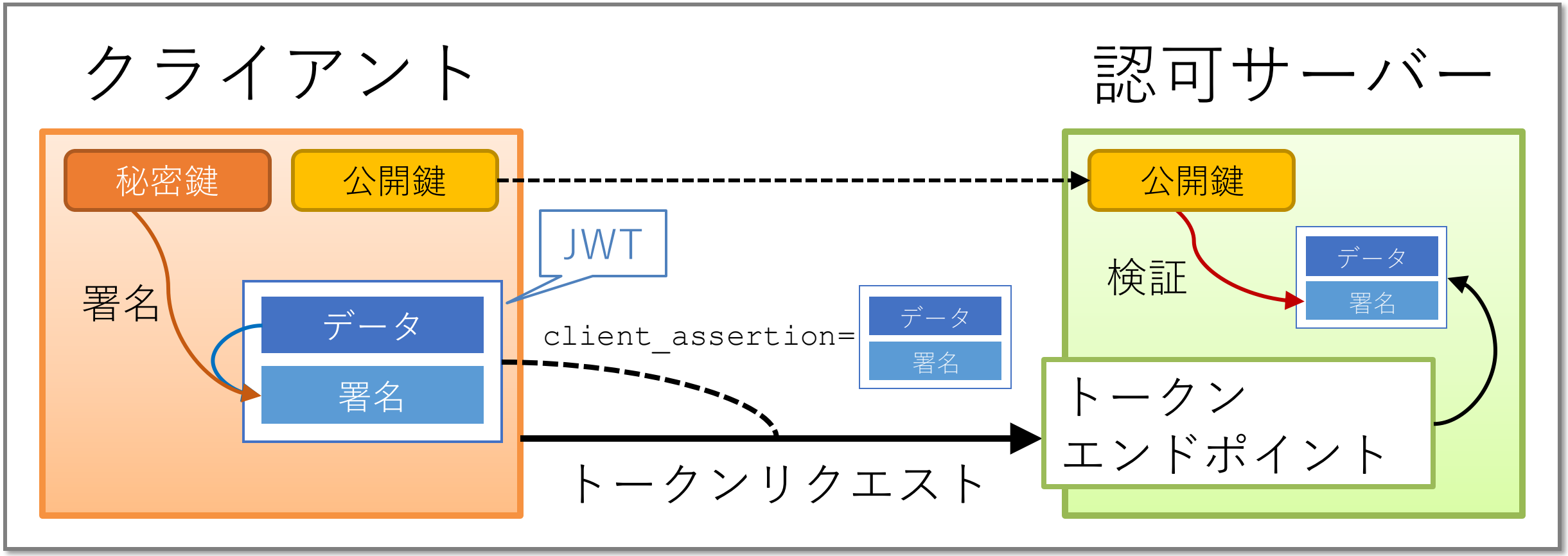


Certified Financial-grade API Client Initiated Backchannel Authentication Profile (FAPI-CIBA) OpenID Providers

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]

クライアントアサーションの署名にクライアントシークレットという共有鍵ではなく、非対称鍵を用いる方法がある。



クライアントアサーションによるクライアント認証方式

クライアントアサーションによるクライアント認証その１

`client_secret_jwt`

✓ 対称鍵系（共有鍵系）。クライアントシークレットを署名鍵として使う。

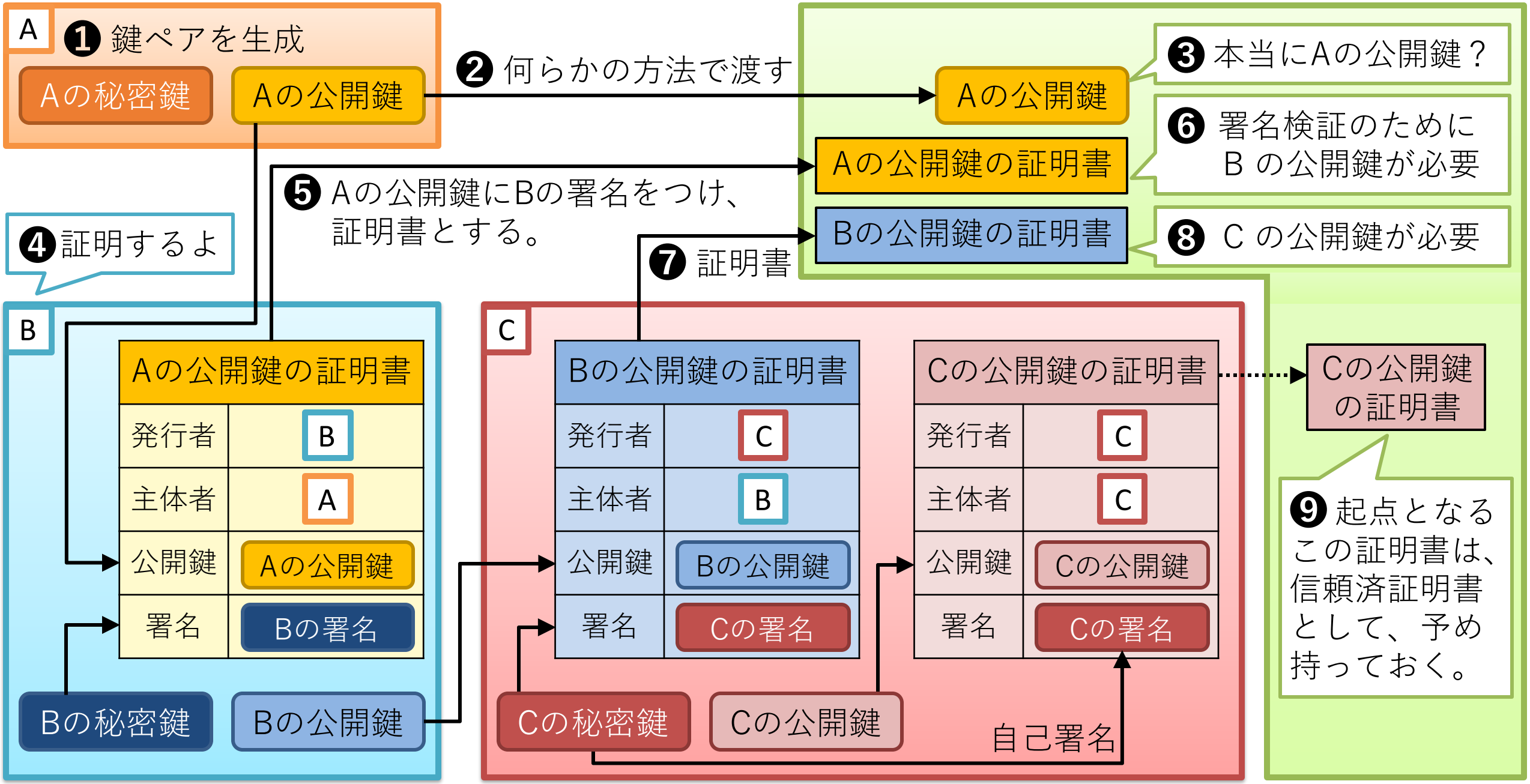
クライアントアサーションによるクライアント認証その２

`private_key_jwt`

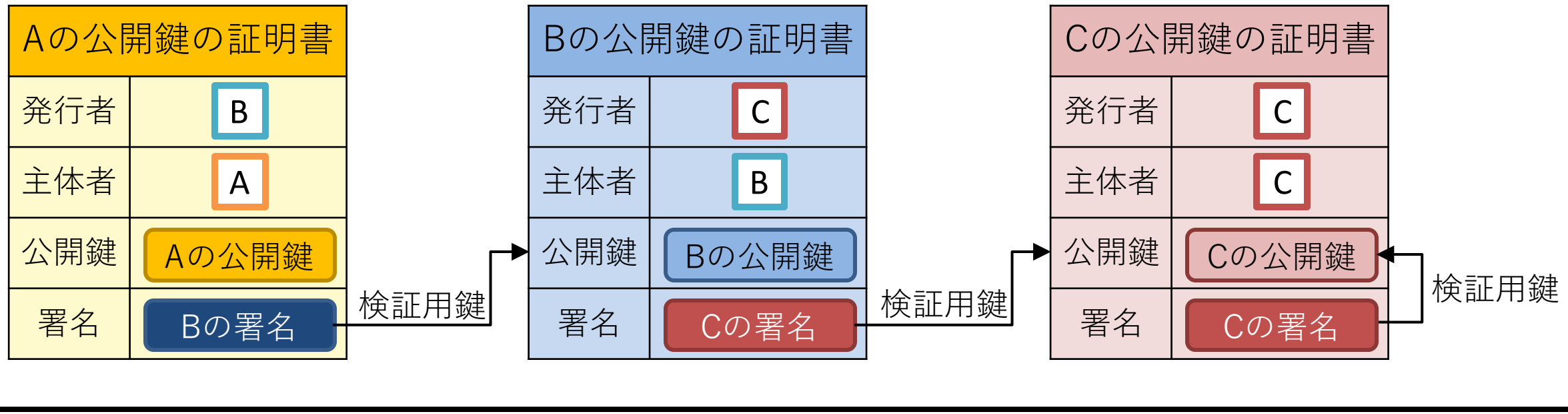
✓ 非対称鍵系（公開鍵暗号系）。秘密鍵を署名鍵として使う。

X.509 証明書


OAuth 2.0
OpenID Connect
Authorization Focused • Reliable and Scalable • Developer Friendly
Faster Time to Market • Choice of Hosting Options • Broad Usage
Integrates with any Authentication methods
API Security
OpenID
AUTHLETE


A
❶ 鍵ペアを生成
Aの秘密鍵
Aの公開鍵
❷ 何らかの方法で渡す
Aの公開鍵
❸ 本当にAの公開鍵？
Aの公開鍵の証明書
❻ 署名検証のために
B の公開鍵が必要
❺ Aの公開鍵にBの署名をつけ、
証明書とする。
❹ 証明するよ
Bの公開鍵の証明書
❽ C の公開鍵が必要
❼ 証明書
B
Aの公開鍵の証明書
発行者 B
主体者 A
公開鍵 Aの公開鍵
署名 Bの署名
Bの秘密鍵
Bの公開鍵
C
Bの公開鍵の証明書
発行者 C
主体者 B
公開鍵 Bの公開鍵
署名 Cの署名
Cの秘密鍵
Cの公開鍵の証明書
発行者 C
主体者 C
公開鍵 Cの公開鍵
署名 Cの署名
Cの公開鍵
Cの公開鍵
自己署名
Cの公開鍵
の証明書
❾ 起点となる
この証明書は、
信頼済証明書
として、予め
持っておく。

証明書チェーン



中間証明書
intermediate certificate

ルート証明書
root certificate

RFC 5280 : Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

4.1. Basic Certificate Fields

```
Certificate ::= SEQUENCE {
  tbsCertificate      TBSCertificate,
  signatureAlgorithm  AlgorithmIdentifier,
  signatureValue      BIT STRING }

TBSCertificate  ::=  SEQUENCE  {
  version         [0]  EXPLICIT Version DEFAULT v1,
  serialNumber         CertificateSerialNumber,
  signature            AlgorithmIdentifier,
  issuer               Name,                              ←発行者
  validity             Validity,                          ←有効期限
  subject              Name,                              ←主体者
  subjectPublicKeyInfo SubjectPublicKeyInfo,              ←公開鍵
  issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                  -- If present, version MUST be v2 or v3
  subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                  -- If present, version MUST be v2 or v3
  extensions      [3]  EXPLICIT Extensions OPTIONAL
                  -- If present, version MUST be v3
  }

Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

CertificateSerialNumber  ::=  INTEGER
```

```
Validity ::= SEQUENCE {
  notBefore      Time,
  notAfter       Time }

Time ::= CHOICE {
  utcTime        UTCTime,
  generalTime    GeneralizedTime }

UniqueIdentifier  ::=  BIT STRING

SubjectPublicKeyInfo  ::=  SEQUENCE  {
  algorithm         AlgorithmIdentifier,
  subjectPublicKey  BIT STRING  }

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
  extnID      OBJECT IDENTIFIER,
  critical    BOOLEAN DEFAULT FALSE,
  extnValue   OCTET STRING
              -- contains the DER encoding of an ASN.1 value
              -- corresponding to the extension type identified
              -- by extnID
  }
```

subject による主体指定は柔軟性にかけるので、Subject Alternative Name（主体者別名）拡張を用いて指定するケースが多い。

✓ X.509 v3 証明書の構造を ASN.1（Abstract Syntax Notation One）という記法で記述したもの。

X.680 シリーズとして関連仕様群が定義されている。

https://www.itu.int/rec/T-REC-X.680/

抽象的なデータ構造	データ構造の具体的な表現方法	形式変換	データ修飾
ASN.1	Distinguished Encoding Rules (DER)	BASE64	PEM
	XML Encoding Rules (XER)	BASE64URL	etc.
	etc.	etc.	

X.509 証明書への適用

ASN.1 (抽象)

証明書の内容 (RFC 5280)

(X.680~X.683)

DER (バイナリデータ) (X.690)

```
30 82 01 06 30 81 AC 02  01 02 30 0A 06 08 2A 86
48 CE 3D 04 03 02 30 0F  31 0D 30 0B 06 03 55 04
03 0C 04 6D 74 6C 73 30  1E 17 0D 31 38 31 30 31
38 31 32 33 37 30 39 5A  17 0D 32 32 30 35 30 32
31 32 33 37 30 39 5A 30  0F 31 0D 30 0B 06 03 55
04 03 0C 04 6D 74 6C 73  30 59 30 13 06 07 2A 86
48 CE 3D 02 01 06 08 2A  86 48 CE 3D 03 01 07 03
42 00 04 D7 27 CB 1C 2A  57 A8 58 F1 09 E1 C7 1C
C5 43 4D C2 EC 72 96 F4  EC A5 31 B9 D0 66 38 C9
FC 0B B4 F3 F7 28 67 0C  52 EC B7 C0 E2 F3 8B 4B
D5 AE 9D E2 17 85 B1 86  5A FB 03 49 BD 2D 1F A2
31 2E 6F 30 0A 06 08 2A  86 48 CE 3D 04 03 02 03
49 00 30 46 02 21 00 FD  11 0B 51 3E BF 02 43 FC
3D 40 18 71 B3 BA B8 BE  86 55 7F 3E 94 04 29 35
94 55 E3 91 5B 3F 37 02  21 00 E7 1D 9E 5D 93 95
6D 49 49 48 64 20 8F 00  F9 BD A5 1A 2A 52 D1 E7
44 36 0E 61 5F 42 3D 4B  F9 26
```

BASE64 (テキストデータ) (RFC 4648)

```
MIIBBjCBrAIBAjAKBggqhkjOPQQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTAx
ODEyMzcwOVoXDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwwEbXRsczBZMBMGByqG
SM49AgEGCCqGSM49AwEHA0IABNcnyxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
/Au08/coZwxS7LfA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0EAwID
SQAwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUVeORWz83AiEA5x2eXZOV
bUlJSGQgjwD5vaUaKlLR50Q2DmFfQj1L+SY=
```

PEM (テキストデータ) (RFC 7468)

```
-----BEGIN CERTIFICATE-----
MIIBBjCBrAIBAjAKBggqhkjOPQQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTAx
ODEyMzcwOVoXDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwwEbXRsczBZMBMGByqG
SM49AgEGCCqGSM49AwEHA0IABNcnyxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
/Au08/coZwxS7LfA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0EAwID
SQAwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUVeORWz83AiEA5x2eXZOV
bUlJSGQgjwD5vaUaKlLR50Q2DmFfQj1L+SY=
-----END CERTIFICATE-----
```

主体者識別名（Subject Distinguished Name）

✓ 証明書の subject フィールドは、公開鍵に紐づく組織の識別名（Distinguished Name）

✓ 主体者識別名の Common Name という要素がホスト名を格納する場所として使われる。

例）Authlete 社の証明書の主体者（Subject）

```
30 19 31 17 30 15 06 03  55 04 03 0C 0E 2A 2E 61
75 74 68 6C 65 74 65 2E  63 6F 6D
```

DER フォーマットの証明書（バイナリデータ）から主体者（Subject）を表す部分を抜き出したもの

```
SEQUENCE (要素数=1)
 SET (要素数=1)
  SEQUENCE (要素数=2)
   OBJECT IDENTIFIER 2.5.4.3 (commonName)
   UTF8String *.authlete.com
```

表現されている ASN.1 データ構造。
DER/BER のデータ構造解析には
ASN.1 JavaScript decoder が便利。
→ https://lapo.it/asn1js/

主体者識別子

CN=*.authlete.com

※ 識別名の文字列表現について定めた仕様↓
RFC 4514 Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names

主体者別名（Subject Alternative Name）

- ✓ X.509 証明書のバージョン 3（RFC 5280）で拡張機能が実装された。
- ✓ 拡張の一つとして、主体者別名（Subject Alternative Name）拡張が定義された。

RFC 5280, 4.2.1.6. Subject Alternative Name

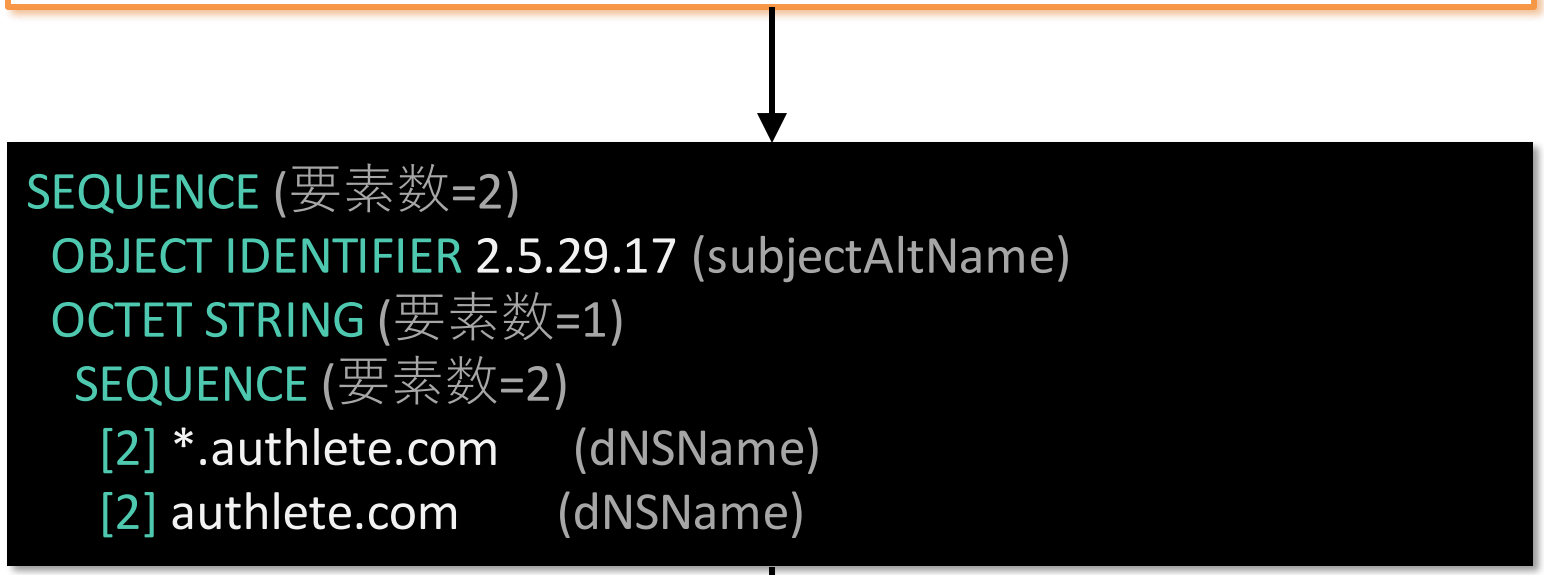
```
SubjectAltName ::= GeneralNames

GeneralNames ::= SEQUENCE SIZE (1..MAX) OF GeneralName

GeneralName ::= CHOICE {
    otherName                 [0] OtherName,
    rfc822Name                [1] IA5String,          ← メールアドレス
    dNSName                   [2] IA5String,          ← DNS 名
    x400Address               [3] ORAddress,
    directoryName             [4] Name,
    ediPartyName              [5] EDIPartyName,
    uniformResourceIdentifier [6] IA5String,          ← URI
    iPAddress                 [7] OCTET STRING,       ← IP アドレス
    registeredID              [8] OBJECT IDENTIFIER }
```

Subject を使う方法では、ホスト名しか指定できず、複数の主体者を指定する方法も規定されていない。

Subject Alternative Name を使えば、メールアドレス、URI、IP アドレス等も使え、複数の主体者を指定できる。

例）Authlete 社の証明書の Subject Alternative Name

```
                                30 27 06 03 55 1D 11 04
20 30 1E 82 0E 2A 2E 61  75 74 68 6C 65 74 65 2E
63 6F 6D 82 0C 61 75 74  68 6C 65 74 65 2E 63 6F
6D
```



`GeneralName` が二つ列挙されている。

タグの値はどちらも [2] で、これは `dNSName` を意味する。

主体者別名

1	DNS 名	`*.authlete.com`
2	DNS 名	`authlete.com`

自己署名証明書作成


OAuth 2.0
OpenID Connect
Authorization Focused • Reliable and Scalable • Developer Friendly
Faster Time to Market • Choice of Hosting Options • Broad Usage
Integrates with any Authentication methods
API Security
OpenID
AUTHLETE

✓ openssl コマンドを確認する

```
$ /usr/bin/openssl version -a
LibreSSL 2.6.5
built on: date not available
platform: information not available
options:  bn(64,64) rc4(16x,int) des(idx,cisc,16,int) blowfish(idx)
compiler: information not available
OPENSSLDIR: "/private/etc/ssl"
```

LibreSSL は 2014 年に OpenSSL から派生した。Mac では macOS High Sierra 以降、OpenSSL の代わりに LibreSSL を使っている。

```
$ brew install openssl
$ /usr/local/opt/openssl/bin/openssl version -a
OpenSSL 1.1.1g  21 Apr 2020
built on: Tue Apr 21 13:28:37 2020 UTC
platform: darwin64-x86_64-cc
options:  bn(64,64) rc4(16x,int) des(int) idea(int) blowfish(ptr)
compiler: clang -fPIC -arch x86_64 -O3 -Wall -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DVPAES_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -D_REENTRANT -DNDEBUG
OPENSSLDIR: "/usr/local/etc/openssl@1.1"
ENGINESDIR: "/usr/local/Cellar/openssl@1.1/1.1.1g/lib/engines-1.1"
Seeding source: os-specific
```

以降の例では OpenSSL の openssl コマンドを使う。LibreSSL 版は OpenSSL 版の新しいコマンドラインオプションに対応していないため。

✓ 秘密鍵を作成する

```
$ openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 > private_key.pem
```

- `genpkey` → 秘密鍵を作成する。RSA や DSA の秘密鍵を作成するための **genrsa** や **gendsa** があるが、現在は **genpkey** に取って代わられている。
- `-algorithm EC` → 楕円曲線（Elliptic Curve）アルゴリズムを使う。このオプションを使用するときは必ず **-pkeyopt** オプションより前に置かなければならないことに注意。
- `-pkeyopt ec_paramgen_curve:P-256` → P-256 曲線を使う。NIST による推奨値の一つ。

  Digital Signature Standard (DSS), D.2.3. Curve P-256
  https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

```
$ cat private_key.pem
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgwSmbZ2gTKFbyy+q6
jXhuGXFuz8mCqjUhRYvQ/EEgqqihRANCAAS4QckKxSuC89ZfQyTWP+M01QUcJRRS
HG/oYhFEU5FOoSJGgxZgJeUFUglE3HiT/NCsdj8COWDCTqP9Iz2oEDnz
-----END PRIVATE KEY-----
```

✓ 鍵の内容を確認する

```
$ openssl pkey -text -noout -in private_key.pem
Private-Key: (256 bit)
priv:
    c1:29:9b:67:68:13:28:56:f2:cb:ea:ba:8d:78:6e:
    19:71:6e:cf:c9:82:aa:35:21:45:8b:d0:fc:41:20:
    aa:a8
pub:
    04:b8:41:c9:0a:c5:2b:82:f3:d6:5f:43:24:d6:3f:
    e3:34:d5:05:1c:25:14:52:1c:6f:e8:62:11:44:53:
    91:4e:a1:22:46:83:16:60:25:e5:05:52:09:44:dc:
    78:93:fc:d0:ac:76:3f:02:39:60:c2:4e:a3:fd:23:
    3d:a8:10:39:f3
ASN1 OID: prime256v1
NIST CURVE: P-256
```

- pkey → 鍵に関する処理を行う。
- -text → プレインテキストで情報を表示
- -noout → エンコードされた情報は非表示
- -in private_key.pem → 入力ファイル指定

秘密鍵が公開鍵の情報を内包していることに注目。JWK 形式にすると分かりやすい。

```
$ npm install -g eckles
$ eckles private_key.pem
{
  "kty": "EC",
  "crv": "P-256",
  "d": "wSmbZ2gTKFbyy-q6jXhuGXFuz8mCqjUhRYvQ_EEgqqg",
  "x": "uEHJCsUrgvPWX0Mk1j_jNNUFHCUUUhxv6GIRRFORTqE",
  "y": "IkaDFmAl5QVSCUTceJP80Kx2PwI5YMJOo_0jPagQOfM"
}
```

d は秘密鍵にだけ含まれる。x と y は秘密鍵と公開鍵の両方に含まれる。
d を消すと公開鍵になる。

参考：公開鍵を抽出する

```
$ openssl pkey -pubout -in private_key.pem > public_key.pem
```

- pkey → 鍵に関する処理を行う。
- -pubout → 公開鍵を出力
- -in private_key.pem → 入力ファイル指定

```
$ cat public_key.pem
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEuEHJCsUrgvPWX0Mk1j/jNNUFHCUU
Uhxv6GIRRFORTqEiRoMWYCXlBVIJRNx4k/zQrHY/Ajlgwk6j/SM9qBA58w==
-----END PUBLIC KEY-----
```

```
$ eckles public_key.pem
{
  "kty": "EC",
  "crv": "P-256",
  "x": "uEHJCsUrgvPWX0Mk1j_jNNUFHCUUUhxv6GIRRFORTqE",
  "y": "IkaDFmAl5QVSCUTceJP80Kx2PwI5YMJOo_0jPagQOfM"
}
```

公開鍵に d は含まれない。

✓ 証明書を作成する

```
$ openssl req -X509 -key private_key.pem -subj /CN=client.example.com > certificate.pem
```

- `req -x509` → X.509 証明書を作成する。`req` は CSR（Certificate Signing Request）を生成するためのコマンドだが、`-x509` オプションをつけると CSR ではなく自己署名証明書を生成する。
- `-key private_key.pem` → 署名に使う秘密鍵・証明の対象となる公開鍵を指定する。
- `-subj /CN=client.example.com` → 主体者識別子を指定する。`-subj` オプションが指定されていない場合は、主体者識別子を入力するためのプロンプトが表示される。
- 有効期限を指定しない場合はデフォルトの 30 日となる。`-days` オプション参照のこと。

```
-----BEGIN CERTIFICATE-----
MIIBjzCCATWgAwIBAgIUdRbH+ElI8iLjnR9eJwCpIU8e8xYwCgYIKoZIzj0EAwIw
HTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMB4XDTIwMDYyNzExMzgwM1oX
DTIwMDcyNzExMzgwM1owHTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMFkw
EwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEuEHJCsUrgvPWX0Mk1j/jNNUFHCUUUhxv
6GIRRFORTqEiRoMWYCXlBVIJRNx4k/zQrHY/Ajlgwk6j/SM9qBA586NTMFEwHQYD
VR0OBBYEFJaMKA22eKiMXGvSojeoLGChcABcMB8GA1UdIwQYMBaAFJaMKA22eKiM
XGvSojeoLGChcABcMA8GA1UdEwEB/wQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIg
N+a6RbvOz+32qOOgKnbQB8sSVeD0gvRoRK13ZuducX4CIQDkgzc1BHoQJ6Pby3aO
byBmhjJS/LhhROgzecP/JMDxqA==
-----END CERTIFICATE-----
```

✓ 証明書の内容を確認する

```
$ openssl x509 -text -noout -in certificate.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            75:16:c7:f8:49:48:f2:22:e3:9d:1f:5e:27:00:a9:21:4f:1e:f3:16
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = client.example.com
        Validity
            Not Before: Jun 27 11:38:03 2020 GMT
            Not After : Jul 27 11:38:03 2020 GMT
        Subject: CN = client.example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:b8:41:c9:0a:c5:2b:82:f3:d6:5f:43:24:d6:3f:
                    e3:34:d5:05:1c:25:14:52:1c:6f:e8:62:11:44:53:
                    91:4e:a1:22:46:83:16:60:25:e5:05:52:09:44:dc:
                    78:93:fc:d0:ac:76:3f:02:39:60:c2:4e:a3:fd:23:
                    3d:a8:10:39:f3
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                96:8C:28:0D:B6:78:A8:8C:5C:6B:D2:A2:37:A8:2C:60:A1:70:00:5C
            X509v3 Authority Key Identifier:
                keyid:96:8C:28:0D:B6:78:A8:8C:5C:6B:D2:A2:37:A8:2C:60:A1:70:00:5C

            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:37:e6:ba:45:bb:ce:cf:ed:f6:a8:e3:a0:2a:76:
         d0:07:cb:12:55:e0:f4:82:f4:68:44:ad:77:66:e7:6e:71:7e:
         02:21:00:e4:83:37:35:04:7a:10:27:a3:db:cb:76:8e:6f:20:
         66:86:32:52:fc:b8:61:44:e8:33:79:c3:ff:24:c0:f1:a8
```

公開鍵の情報

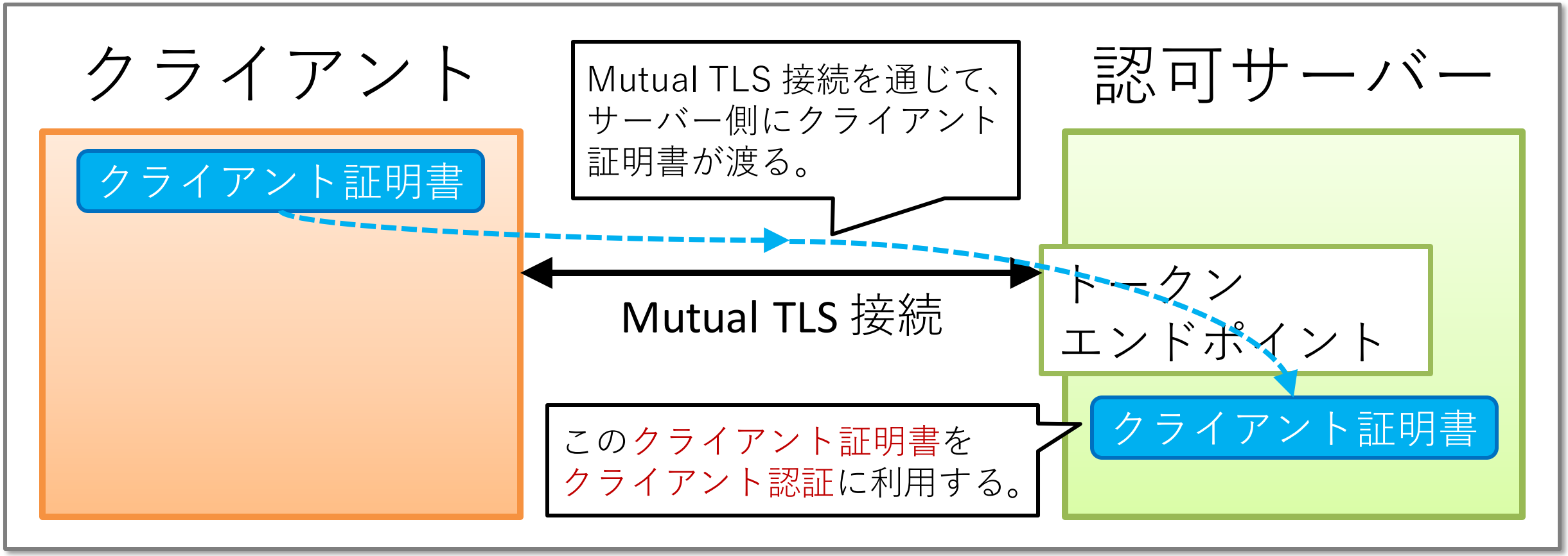
- `x509` → X.509 証明書に関する処理を行う。
- `-text` → プレインテキストで情報を表示
- `-noout` → エンコードされた情報は非表示
- `-in certificate.pem` → 入力ファイル指定

自己署名証明書なので、発行者（Issuer）と主体者（Subject）が同一であることに注目。

証明書系クライアント認証

クライアント証明書をクライアント認証に使う方法がある。

✓ RFC 8705 : OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens / 2. Mutual TLS for Client Authentication



クライアント証明書によるクライアント認証その１（PKI 証明書）

tls_client_auth

✓ クライアントは下記のメタデータのどれか一つを認可サーバーに登録しておく。

クライアントメタデータ	証明書フィールド
tls_client_auth_subject_dn	Subject Distinguished Name
tls_client_auth_san_dns	Subject Alternative Name, DNS
tls_client_auth_san_uri	Subject Alternative Name, URI
tls_client_auth_san_ip	Subject Alternative Name, IP address
tls_client_auth_san_email	Subject Alternative Name, Email

✓ 認可サーバーは、提示された証明書の該当項目が事前登録された値と一致することを確認する。

クライアント証明書によるクライアント認証その２（自己署名証明書）

self_signed_tls_client_auth

✓ クライアントは、X.509 証明書を含む JWK Set（jwks）、または、その JWK Set を指す URI（jwks_uri）を認可サーバーに登録しておく。

```
{
  "kty": "EC",
  "x":   "1yfLHCpXqFjxCeHHHMVDTcLscpb07KUxudBmOMn8C7Q",
  "y":   "8_coZwxS7LfA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8",
  "crv": "P-256",
  "x5c": [
    "MIIBBjCBrAIBAjAKBggqhkjOPQQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTA
     xODEyMzcwOVoXDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwwEbXRsczBZMBMGBy
     qGSM49AgEGCCqGSM49AwEHA0IABNcnyxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZ
     jjJ/Au08/coZwxS7LfA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0E
     AwIDSQAwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUVeORWz83AiEA5x2
     eXZOVbUlJSGQgjwD5vaUaKlLR50Q2DmFfQj1L+SY="
  ]
}
```

←証明書の公開鍵の JWK 表現

←証明書

RFC 8705, Appendix A. Example "cnf" Claim, Certificate, and JWK より抜粋（微調整含む）

RFC 7517 : JSON Web Key (JWK), 4.7. "x5c" (X.509 Certificate Chain Parameter)

The "x5c" (X.509 certificate chain) parameter contains a chain of one or more PKIX certificates [RFC5280]. The certificate chain is represented as a JSON array of certificate value strings. Each string in the array is a base64-encoded (Section 4 of [RFC4648] -- not base64url-encoded) DER [ITU.X690.1994] PKIX certificate value. The PKIX certificate containing the key value MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one. The key in the first certificate MUST match the public key represented by other members of the JWK. Use of this member is OPTIONAL.

- ✓ x5c は X.509 証明書チェーンを JSON 配列で表現したもの。
- ✓ 各要素は、X.509 証明書を DER (Distinguished Encoding Rules) で表現したものを、さらに BASE64 で表現したもの。
- ✓ 最初の要素は、当該 JWK が表す公開鍵を含む証明書でなければならない。
- ✓ 二番目以降の要素は、直前の証明書の署名を検証するための公開鍵を含む証明書。

メタデータ


OAuth 2.0
OpenID Connect
Authorization Focused • Reliable and Scalable • Developer Friendly
Faster Time to Market • Choice of Hosting Options • Broad Usage
Integrates with any Authentication methods
API Security
OpenID
AUTHLETE

OpenID Connect Discovery 1.0, 3. OpenID Provider Metadata

token_endpoint_auth_methods_supported

OPTIONAL. JSON array containing a list of Client Authentication methods supported by this Token Endpoint. The options are `client_secret_post`, `client_secret_basic`, `client_secret_jwt`, and `private_key_jwt`, as described in Section 9 of OpenID Connect Core 1.0 [OpenID.Core]. Other authentication methods MAY be defined by extensions. If omitted, the default is `client_secret_basic` -- the HTTP Basic Authentication Scheme specified in Section 2.3.1 of OAuth 2.0 [RFC6749].

token_endpoint_auth_signing_alg_values_supported

OPTIONAL. JSON array containing a list of the JWS signing algorithms (`alg` values) supported by the Token Endpoint for the signature on the JWT [JWT] used to authenticate the Client at the Token Endpoint for the `private_key_jwt` and `client_secret_jwt` authentication methods. Servers SHOULD support `RS256`. The value `none` MUST NOT be used.

- ✓ `token_endpoint_auth_methods_supported` は、トークンエンドポイントでサポートされるクライアント認証方式のリスト。
- ✓ `token_endpoint_auth_signing_alg_values_supported` は、トークンエンドポイントでサポートされるクライアントアサーションの署名アルゴリズムのリスト。JWT系クライアント認証（`private_key_jwt` と `client_secret_jwt`）のみを対象としたメタデータ。

RFC 8414 : OAuth 2.0 Authorization Server Metadata

✓ 認可サーバーがサポートするクライアント認証方式とクライアントアサーション署名アルゴリズムを、エンドポイントごとに定義している。

トークンエンドポイント (RFC 6749)
• token_endpoint_auth_methods_supported • token_endpoint_auth_signing_alg_values_supported
リボケーションエンドポイント (RFC 7009)
• revocation_endpoint_auth_methods_supported • revocation_endpoint_auth_signing_alg_values_supported
イントロスペクションエンドポイント (RFC 7662)
• introspection_endpoint_auth_methods_supported • introspection_endpoint_auth_signing_alg_values_supported

OpenID Connect Dynamic Client Registration 1.0, 2. Client Metadata

token_endpoint_auth_method

OPTIONAL. Requested Client Authentication method for the Token Endpoint. The options are `client_secret_post`, `client_secret_basic`, `client_secret_jwt`, `private_key_jwt`, and `none`, as described in Section 9 of OpenID Connect Core 1.0 [OpenID.Core]. Other authentication methods MAY be defined by extensions. If omitted, the default is `client_secret_basic` -- the HTTP Basic Authentication Scheme specified in Section 2.3.1 of OAuth 2.0 [RFC6749].

token_endpoint_auth_signing_alg

OPTIONAL. JWS [JWS] `alg` algorithm [JWA] that MUST be used for signing the JWT [JWT] used to authenticate the Client at the Token Endpoint for the `private_key_jwt` and `client_secret_jwt` authentication methods. All Token Requests using these authentication methods from this Client MUST be rejected, if the JWT is not signed with this algorithm. Servers SHOULD support `RS256`. The value `none` MUST NOT be used. The default, if omitted, is that any algorithm supported by the OP and the RP MAY be used.

✓ 当該クライアントが、トークンエンドポイントで使用するクライアント認証方式とクライアントアサーション署名アルゴリズム。

OpenID Connect Client Initiated Backchannel Authentication Flow – Core 1.0
7.1. Authentication Request

The Client MUST authenticate to the Backchannel Authentication Endpoint using the authentication method registered for its `client_id`, such as the authentication methods from Section 9 of [OpenID.Core] or authentication methods defined by extension in other specifications.

✓ **CIBA**（Client Initiated Backchannel Authentication）のバックチャネル認証エンドポイントではクライアント認証が必須。つまり CIBA を利用できるのはコンフィデンシャルクライアントのみ。

バックチャネル認証エンドポイントにおけるクライアント認証関連メタデータに関する議論

[Issue 102] CIBA: Metadata for client auth at backchannel endpoint
https://bitbucket.org/openid/mobile/issues/102

RFC 8414 に倣い、`backchannel_authentication_endpoint_auth_methods_supported` と `backchannel_authentication_endpoint_auth_signing_alg_values_supported` を定義してはどうかという提案だったが、CIBA の仕様に取り込まれることはなかった。なぜか？

Brian Campbell>
There's some historical weirdness to how client authentication has come to be represented (and named) in metadata. Client registration metadata has only a `token_endpoint_auth_method`, which despite the name has become the de facto place in the client data model to say how the client will authenticate to the AS when making any direct client -> AS call. The parameter name is a bit unfortunate but I believe it makes sense to have a single (backchannel) authentication method per client. RFC 8414 took a different direction and has `revocation_endpoint_auth_methods_supported` and `introspection_endpoint_auth_methods_supported` etc., which I believe was a mistake. I don't see a clear use case for needing or allowing a different set of auth methods for the different endpoints. And having a bunch of `_supported` parameters for different endpoints seems likely to clutter up the metadata document with a bunch of redundant info.

I'd propose that some text be added into CIBA core that says that, for the backchannel authentication endpoint, the AS supports the same client authentication methods as indicated with `token_endpoint_auth_methods_supported`. And state that, for a client, the `token_endpoint_auth_method` is the authentication method registered for its `client_id` regardless of the endpoint being called.

- ✓ エンドポイントの違いによって異なるクライアント認証方法を使うべきユースケースは特にない。RFC 8414 はエンドポイント毎にクライアント認証関連メタデータを定義したが、誤りだと思う。
- ✓ クライアント認証方法はクライアント毎に一つでよく、事実上 `token_endpoint_auth_method` メタデータがその用途で使われている。ただし、その名称は残念であるが。
- ✓ バックチャネル認証エンドポイントのクライアント認証関連メタデータはトークンエンドポイントのそれを流用する。

RFC 8705, 2.1.2. Client Registration Metadata

tls_client_auth_subject_dn
: A string representation -- as defined in [RFC4514] -- of the expected subject distinguished name of the certificate that the OAuth client will use in mutual-TLS authentication.

tls_client_auth_san_dns
: A string containing the value of an expected `dNSName` SAN entry in the certificate that the OAuth client will use in mutual-TLS authentication.

tls_client_auth_san_uri
: A string containing the value of an expected `uniformResourceIdentifier` SAN entry in the certificate that the OAuth client will use in mutual-TLS authentication.

tls_client_auth_san_ip
: A string representation of an IP address in either dotted decimal notation (for IPv4) or colon-delimited hexadecimal (for IPv6, as defined in [RFC5952]) that is expected to be present as an `iPAddress` SAN entry in the certificate that the OAuth client will use in mutual-TLS authentication. Per Section 8 of [RFC5952], the IP address comparison of the value in this parameter and the SAN entry in the certificate is to be done in binary format.

tls_client_auth_san_email
: A string containing the value of an expected `rfc822Name` SAN entry in the certificate that the OAuth client will use in mutual-TLS authentication.

Financial-grade API の要求事項

Financial-grade API – Part 1: Read-Only API Security Profile, 5.2.2. Authorization server, 4

shall authenticate the confidential client at the token endpoint 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];

Financial-grade API – Part 2: Read and Write API Security Profile, 5.2.2. Authorization server, 14

shall authenticate the confidential client at the token endpoint using one of the following methods (this overrides FAPI part 1 clause 5.2.2.4):

1. Mutual TLS for OAuth Client Authentication as specified in section 2 of [MTLS];
2. `private_key_jwt` as specified in section 9 of [OIDC];

クライアント認証方式	Part 1	Part 2
client_secret_basic	×	×
client_secret_post	×	×
client_secret_jwt	○	×
private_key_jwt	○	○
tls_client_auth	○	○
self_signed_tls_client_auth	○	○

- ✓ RFC 6749 で示されているクライアント認証方式（client_secret_basic / client_secret_post）は、FAPI では使用不可。
- ✓ JWT 系/証明書系のクライアント認証方式を使うことになるが、署名アルゴリズムが対称鍵系の client_secret_jwt は、FAPI Part 2 では使用不可。

Financial-grade API – Part 2: Read and Write API Security Profile, 8.6. JWS algorithm considerations

Both clients and authorisation servers:

1. shall use `PS256` or `ES256` algorithms;
2. should not use algorithms that use RSASSA-PKCS1-v1_5 (e.g. `RS256`);
3. shall not use `none`;

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

- ✓ FAPI Part 2 では、JWS の署名アルゴリズムとして `PS256` か `ES256` しか使えない。
- ✓ 特に、RFC 7518 (JSON Web Algorithms) で Recommended とされている `RS256` が使用不可であることに注意。
- ✓ リクエストオブジェクトや ID トークン等、JWS はいろいろな場所で使われているが、クライアントアサーションもその一つ。

FAPI Part 2 では、クライアントアサーションの署名アルゴリズムが `PS256` と `ES256` のどちらかに制限される。

Financial-grade API – Part 1: Read-Only API Security Profile, 5.2.2. Authorization server, 5 & 6

5. shall require a key of size 2048 bits or larger if RSA algorithms are used for the client authentication;
6. shall require a key of size 160 bits or larger if elliptic curve algorithms are used for the client authentication;

✓ クライアント認証で RSA アルゴリズムを使う場合、鍵長は 2048 ビット以上。

✓ クライアント認証で楕円曲線アルゴリズムを使う場合、鍵長は 160 ビット以上。

↓

「クライアントアサーションの署名で」と同義。

Authlete のソースコード（企業秘密）の一部。鍵長が FAPI の要求仕様を満たしているかチェックしている。

```
// The minimum size of the key to verify the signature of the client's
// assertion allowed by the service profiles of the service.
Integer allowedMinSize =
    getAllowedMinSizeOfKeyForClientAssertionSignatureVerification();

if (allowedMinSize == null)
{
    // This means that there is no requirement for the minimum size of
    // the key to verify the signature of the client's assertion by the
    // service profiles of the service.
    return;
}

// The key that was used to verify the signature of the client's assertion
// when the client was authenticated with 'private_key_jwt'.
JWK jwk = mBuilder.getJwkForClientAssertionSignatureVerification();

if (jwk.size() >= allowedMinSize)
{
    // OK. The key size is allowed.
    return;
}

// The length of the key to verify the signature of the client's assertion
// is shorter than the minimum size.
throw toException(invalid_request, A152303, allowedMinSize);
```

えぇーっ! FAPI の公式 Conformance Suite でさえ鍵長のテストをしてないのに、Authlete はそこまで実装してるの?! Conformance Suite 自身のテストに Authlete が使われるのも頷ける実装品質の高さだね! Authlete すごい! #ステマ

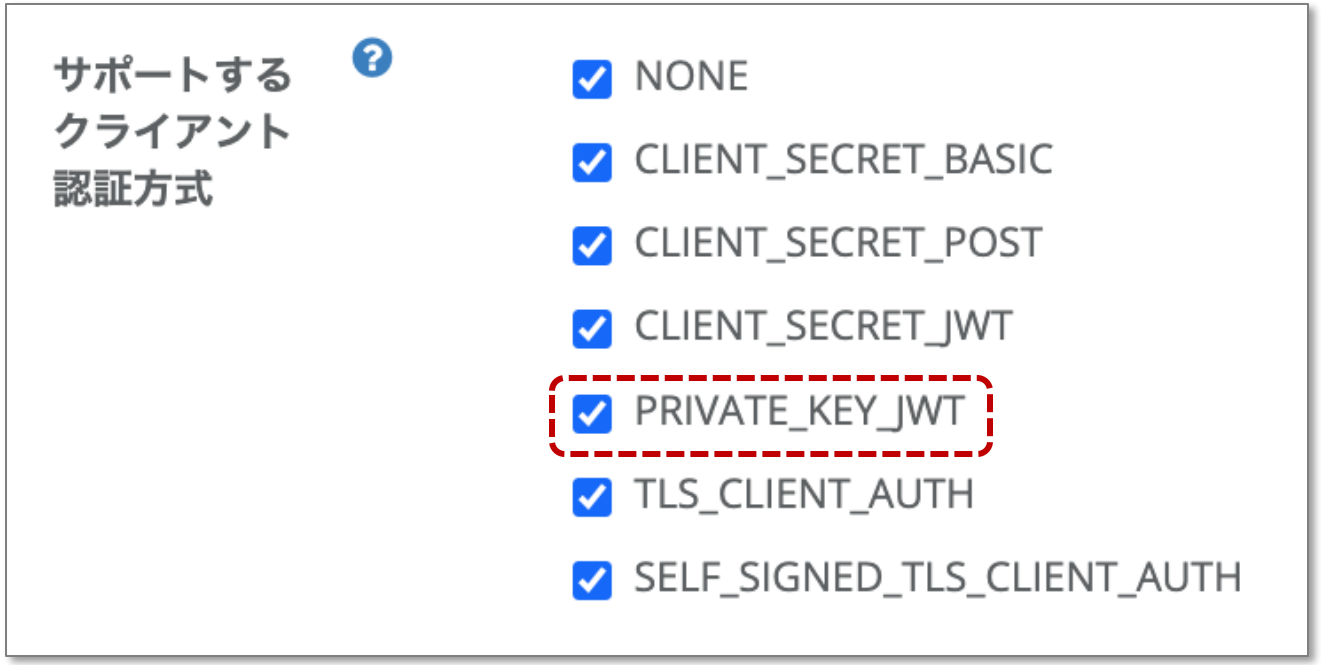
private_key_jwt の実際

サーバーの設定（1/2）

✓ `token_endpoint_auth_methods_supported`（サポートするクライアント認証方式）に `private_key_jwt` を含める。

Authlete の場合 →

『Authlete ナレッジベース』(https://kb.authlete.com) の『`private_key_jwt` によるクライアント認証』もご参照ください！



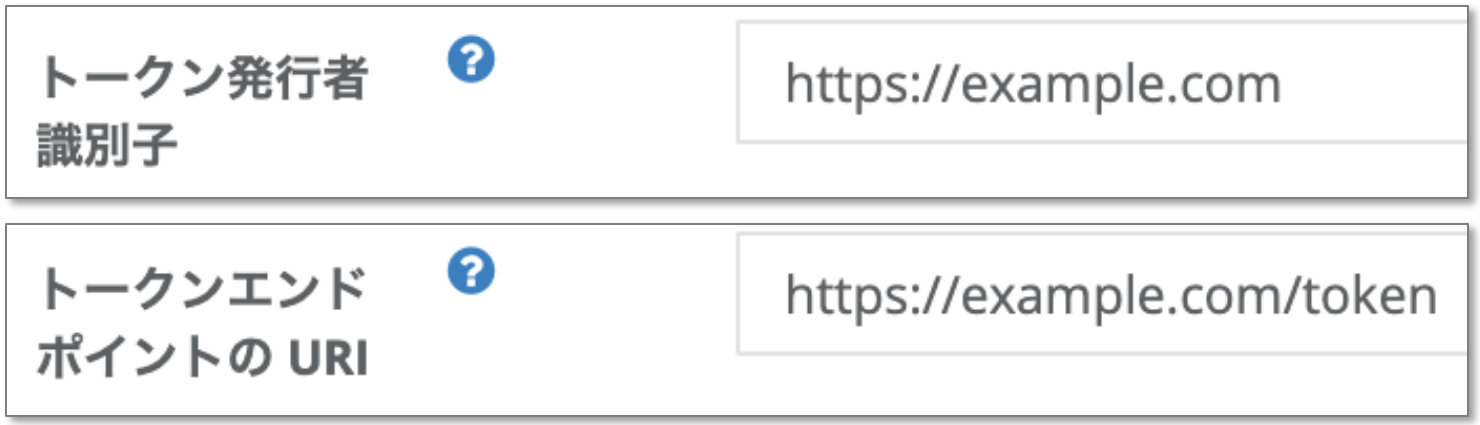
✓ `token_endpoint_auth_signing_alg_values_supported`（サポートするクライアントアサーション署名アルゴリズム）にサポートするアルゴリズムを含める。

Authlete の場合 → RFC 7518 (JWA) に挙げられている署名アルゴリズムを全てサポート。
特別な設定は不要。

サーバーの設定（2/2）

✓ クライアントアサーションの aud の値として使うため、issuer（トークン発行者識別子）または token_endpoint（トークンエンドポイントの URL）を設定する。

Authlete の場合 →



- issuer は OpenID Provider（IdP）の識別子。ID トークンなどの、IdP が発行する JWT の iss の値として用いられる。また、ディスカバリーエンドポイント（OIDC Discovery 1.0）のベース URL としても用いられる。

- aud の正しい値は厳密には実装依存。RFC 7523 や OIDC Core ならトークンエンドポイントの URL だが、CIBA なら発行者識別子となる。

  Authlete の場合 → トークンエンドポイントの URL と発行者識別子、どちらでもよい。

クライアントの設定（1/2）

✓ クライアントタイプをコンフィデンシャルにする。

Authlete の場合 →

✓ `token_endpoint_auth_method`（クライアント認証方式）で `private_key_jwt` を選択する。

Authlete の場合 →

✓ `token_endpoint_auth_signing_alg`（クライアントアサーション署名アルゴリズム）で非対称鍵系アルゴリズムを選択する。

Authlete の場合 →

クライアントの設定（2/2）

✓ 鍵を JWK Set 形式で生成する。

Bespoke Engineering 社と Authlete 社が提供する無料サービスの mkjwk を使えば JWK 形式の鍵を簡単に生成可能。mkjwk のソースコードは全て公開されている。

✓ 公開鍵を含む JWK Set を `jwks` として登録する。または JWK Set の場所を `jwks_uri` として登録する。

JWK セットの内容

```
{
  "keys": [
    {
      "kty": "EC",
      "crv": "P-256",
      "x": "eaob5vYd87rmFzFZAbDngxS0DbkcG5Nj8iAheCa90QA",
      "y": "r_lrETJsTP0fL47VuKFxvlxzrCpe-OtHE6Mo6FEREhg",
      "alg": "ES256"
    }
  ]
}
```

公開鍵

`https://mkjwk.org/`

クライアントアサーション作成

✓ クライアントアサーションのペイロード部を用意する。

項目	要否
iss	必須
sub	必須
aud	必須
exp	必須
nbf	任意
iat	任意
jti	任意

ペイロード部の例

```
{
  "iss":"4326385670",   ←クライアントID
  "sub":"4326385670",   ←クライアントID
  "aud":"https://example.com/token",←トークン
                                    エンドポイント
  "exp":1593684000,←有効期限終了日時
  "jti":"e7e1ea3a-92c6-441a-91d8-8a24b9cda16f"
}
```

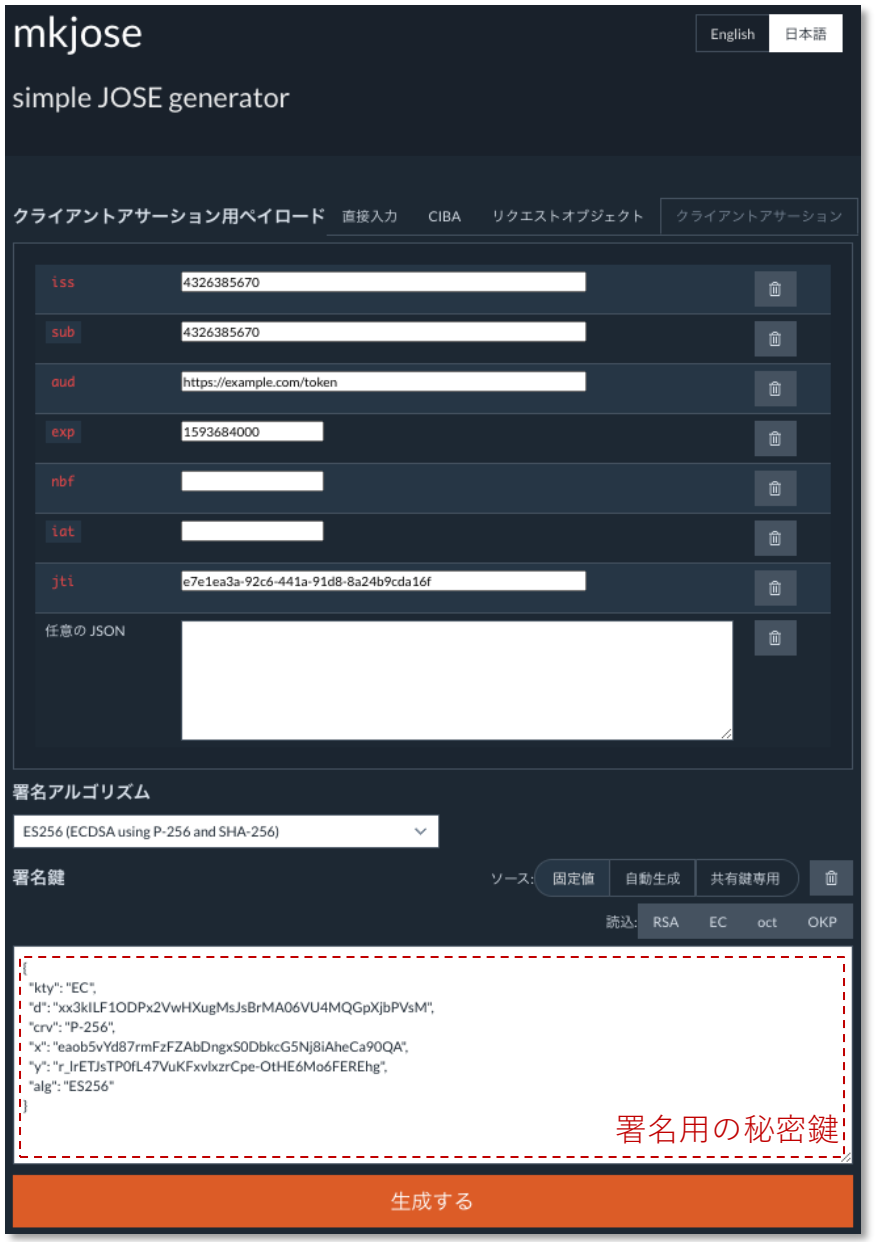
RFC 7523 では jti は任意とされているが、OIDC では必須。
Authlete の現実装は常に jti を要求する。

✓ 秘密鍵で署名し、JWT 化する。

Bespoke Engineering 社と Authlete 社が提供する無料サービスの mkjose を使えば JOSE (JSON Object Signing and Encryption) を簡単に生成可能 (JWT は JOSE の一種)。mkjose のソースコードは全て公開されている。

- mkjose は https://github.com/authlete/authlete-jose の一部である jose-generator を利用している。コマンドラインで JOSE を生成したい場合は、jose-generator を直接使うとよい。

https://mkjose.org/



トークンリクエスト（private_key_jwt でクライアント認証）

```
$ curl -k -v https://example.com/token
   -d client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer
   -d client_assertion=eyJhbGciOiJFUzI1NiJ9.ewogICAgImlzcyI6ICI0MzI2Mzg1NjcwIiwKICAgICJzdWIiOiAiNDMyNjM4NTY3MCIsCiAgICAiYXVkIjogImh0dHBzOi8vZXhhbXBsZS5jb20vdG9rZW4iLAogICAgImV4cCI6IDE1OTM2ODQwMDAsCiAgICAianRpIjogImU3ZTFlYTNhLTkyYzYtNDQxYS05MWQ4LThhMjRiOWNkYTE2ZiIKfQ.IOSZca1mDB_2J9PJiMsqRuEYyQnBrihk70JqoB7F85Gflre2smNMAFDpys6uqkQSy9TnzbLdhSZDp5duaSoI3A
   -d grant_type=client_credentials
   -d client_id=4326385670
```

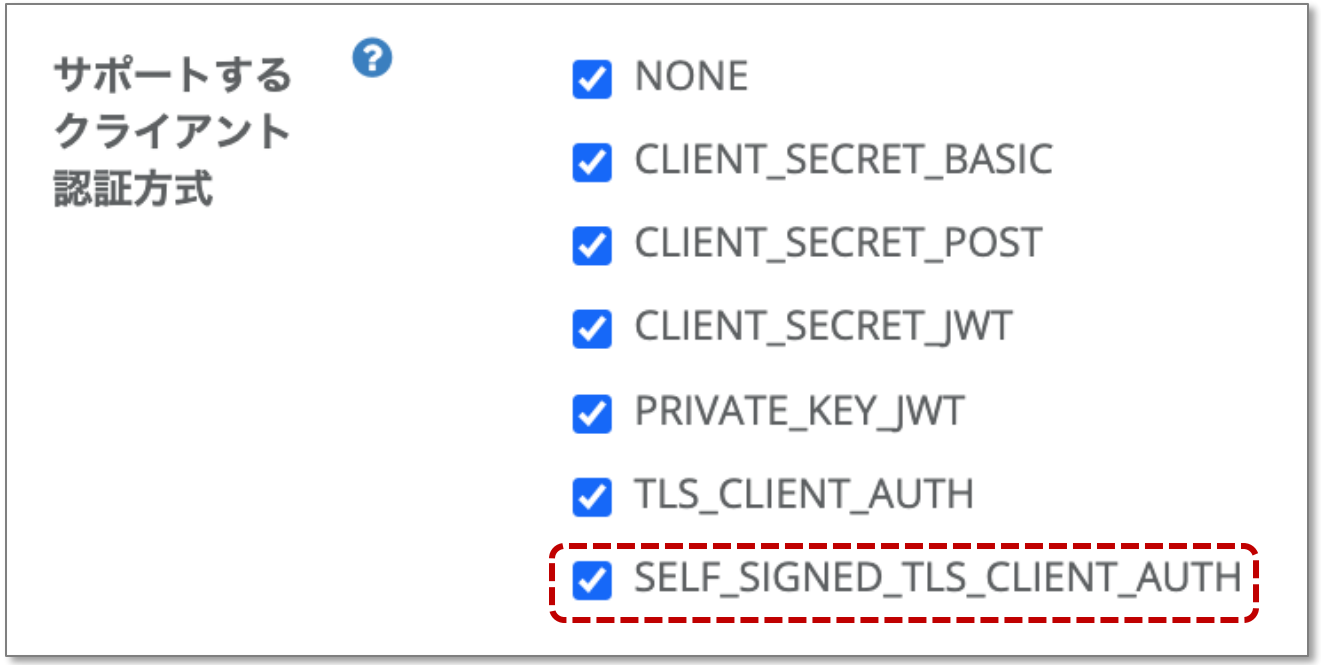
```
POST /token HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer&
client_assertion=eyJhbGciOiJFUzI1NiJ9.ewogICAgImlzcyI6ICI0MzI2Mzg1NjcwIiwKICAgICJzdWIiOiAiNDMyNjM4NTY3MCIsCiAgICAiYXVkIjogImh0dHBzOi8vZXhhbXBsZS5jb20vdG9rZW4iLAogICAgImV4cCI6IDE1OTM2ODQwMDAsCiAgICAianRpIjogImU3ZTFlYTNhLTkyYzYtNDQxYS05MWQ4LThhMjRiOWNkYTE2ZiIKfQ.IOSZca1mDB_2J9PJiMsqRuEYyQnBrihk70JqoB7F85Gflre2smNMAFDpys6uqkQSy9TnzbLdhSZDp5duaSoI3A&
grant_type=client_credentials&
client_id=4326385670
```

self_signed_tls_client_auth の実際

サーバーの設定

✓ `token_endpoint_auth_methods_supported`（サポートするクライアント認証方式）に `self_signed_tls_client_auth` を含める。

Authlete の場合 →



クライアントの設定（1/3）

✓ クライアントタイプをコンフィデンシャルにする。

Authlete の場合 →

✓ `token_endpoint_auth_method`（クライアント認証方式）で `self_signed_tls_client_auth` を選択する。

Authlete の場合 →

クライアントの設定（2/3）

✓ 自己署名証明書を作成する。

↓秘密鍵 (PEM)

```
$ openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 > private_key.pem
$ openssl req -x509 -key private_key.pem -subj /CN=client.example.com > certificate.pem
```

↑証明書 (PEM)

✓ 公開鍵と自己署名証明書を含む JWK を作成する。

```
$ openssl pkey -pubout -in private_key.pem > public_key.pem ←公開鍵 (PEM)
$ eckles public_key.pem > public_key.jwk ←公開鍵 (JWK)
$ (sed -e '$d' -e '/"$/s/$/,/' public_key.jwk; printf '  "x5c": [\n    "'; sed -e '/^-/d' certificate.pem | tr -d '\n'; printf '"\n  ]\n}\n') > certificate.jwk
```

↓証明書 (JWK)

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "uEHJCsUrgvPWX0Mk1j_jNNUFHCUUUhxv6GIRRFORTqE",
  "y": "IkaDFmAl5QVSCUTceJP80Kx2PwI5YMJOo_0jPagQOfM",
  "x5c": [
    "MIIBjzCCATWgAwIBAgIUdRbH+ElI8iLjnR9eJwCpIU8e8xYwCgYIKoZIzj0EAwIwHTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMB4XDTIwMDYyNzExMzgwM1oXDTIwMDcyNzExMzgwM1owHTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEuEHJCsUrgvPWX0Mk1j/jNNUFHCUUUhxv6GIRRFORTqEiRoMWYCXlBVIJRNx4k/zQrHY/Ajlgwk6j/SM9qBA586NTMFEwHQYDVR0OBBYEFJaMKA22eKiMXGvSojeoLGChcABcMB8GA1UdIwQYMBaAFJaMKA22eKiMXGvSojeoLGChcABcMA8GA1UdEwEB/wQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIgN+a6RbvOz+32qOOgKnbQB8sSVeD0gvRoRK13ZuducX4CIQDkgzc1BHoQJ6Pby3aObyBmhjJS/LhhROgzecP/JMDxqA=="
  ]
}
```

←公開鍵（public_key.jwk から抜き出した部分）

↑証明書（certificate.pem から抜き出した部分）

クライアントの設定（3/3）

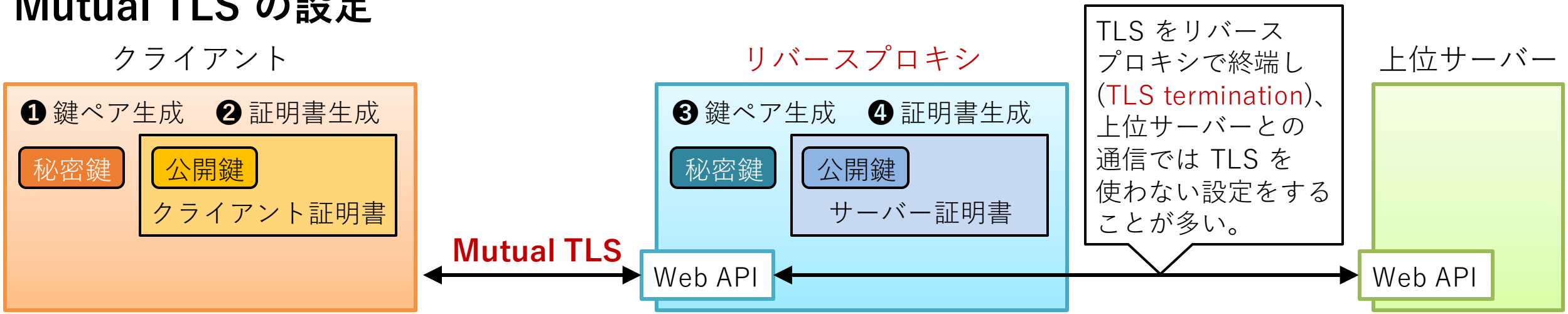
✓ 自己署名証明書をJWK Set を `jwks` として登録する。または JWK Set の場所を `jwks_uri` として登録する。

Authlete の場合 →

JWK セットの内容


```
{
  "keys": [
    {
      "kty": "EC",
      "crv": "P-256",
      "x": "uEHJCsUrgvPWX0Mk1j_jNNUFHCUUUhxv6GIRRFORTqE",
      "y": "IkaDFmAl5QVSCUTceJP80Kx2PwI5YMJOo_0jPagQOfM",
      "x5c": [
"MIIBjzCCATWgAwIBAgIUdRbH+EII8iLjnR9eJwCpIU8e8xYwCgYIKoZIzj0EAwIwHTEbMBkG
A1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMB4XDTIwMDYyNzExMzgwM1oXDTIwMDc
yNzExMzgwM1owHTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMFkwEwYHKo
ZIzj0CAQYIKoZIzj0DAQcDQgAEuEHJCsUrgvPWX0Mk1j/jNNUFHCUUUhxv6GIRRFORTqEiR
oMWYCXlBVIJRNx4k/zQrHY/AjIgwk6j/SM9qBA586NTMFEwHQYDVR0OBBYEFJaMKA22eK
iMXGvSojeoLGChcABcMB8GA1UdIwQYMBaAFJaMKA22eKiMXGvSojeoLGChcABcMA8GA
1UdEwEB/wQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIgN+a6RbvOz+32qOOgKnbQB8sS
VeD0gvRoRK13ZuducX4CIQDkgzc1BHoQJ6Pby3aObyBmhjJS/LhhROgzecP/JMDxqA=="
    ]
  }
 ]
}
```

Mutual TLS の設定




```
events {}
http {
  server {
    listen 8443 ssl;   ← TLS 通信をポート番号 8443 で受け付ける。
    ssl_certificate      /etc/nginx/server_certificate.pem;  ← サーバー証明書
    ssl_certificate_key  /etc/nginx/server_private_key.pem;  ← サーバー秘密鍵
    ssl_verify_client    optional_no_ca;                     ← クライアント証明書を要求する。
    proxy_set_header X-Ssl-Cert $ssl_client_escaped_cert;    ← クライアント証明書の内容を上位サーバーに伝える。

    location = /token {                                      ← リバースプロキシが /token で受け取ったリクエストを、
      proxy_pass http://localhost:8080/api/token;            ← 上位サーバー(http://localhost:8080) の /api/token に転送する。
    }
  }
}
```

サーバーが要求しない限り、クライアントはクライアント証明書を送ってこないことに注意。ここでは、証明書を要求するが検証処理を省くよう、optional_no_ca を設定。

リバースプロキシで TLS 終端する場合、上位サーバーのスキームは https ではなく http になる。

nginx をリバースプロキシとして使う際の設定例（実験用）

Module ngx_http_ssl_module, Embedded Variables

http://nginx.org/en/docs/http/ngx_http_ssl_module.html#variables

$ssl_client_escaped_cert

returns the client certificate in the PEM format (urlencoded) for an established SSL connection (1.13.5);

$ssl_client_cert

returns the client certificate in the PEM format for an established SSL connection, with each line except the first prepended with the tab character; this is intended for the use in the `proxy_set_header` directive;

The variable is deprecated, the `$ssl_client_escaped_cert` variable should be used instead.

`proxy_set_header X-Ssl-Cert $ssl_client_escaped_cert`

```
X-Ssl-Cert: -----BEGIN%20CERTIFICATE-----%0AMIIBjzCCATWgAwIBAgIUdRb
H%2BElI8iLjnR9eJwCpIU8e8xYwCgYIKoZIzj0EAwIw%0AHTEbMBkGA1UEAwwSY2xpZ
W50LmV4YW1wbGUuY29tMB4XDTIwMDYyNzExMzgwM1oX%0ADTIwMDcyNzExMzgwM1owH
TEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMFkw%0AEwYHKoZIzj0CAQYIKoZIz
j0DAQcDQgAEuEHJCsUrgvPWX0Mk1j%2FjNNUFHCUUUhxv%0A6GIRRFORTqEiRoMWYCX
lBVIJRNx4k%2FzQrHY%2FAjlgwk6j%2FSM9qBA586NTMFEwHQYD%0AVR0OBBYEFJaMK
A22eKiMXGvSojeoLGChcABcMB8GA1UdIwQYMBaAFJaMKA22eKiM%0AXGvSojeoLGChc
ABcMA8GA1UdEwEB%2FwQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIg%0AN%2Ba6RbvOz
%2B32qOOgKnbQB8sSVeD0gvRoRK13ZuducX4CIQDkgzc1BHoQJ6Pby3aO%0AbyBmhjJ
S%2FLhhROgzecP%2FJMDxqA%3D%3D%0A-----END%20CERTIFICATE-----%0A
```

`proxy_set_header X-Ssl-Cert $ssl_client_cert`

```
X-Ssl-Cert: -----BEGIN CERTIFICATE-----
 MIIBjzCCATWgAwIBAgIUdRbH+ElI8iLjnR9eJwCpIU8e8xYwCgYIKoZIzj0EAwIw
 HTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMB4XDTIwMDYyNzExMzgwM1oX
 DTIwMDcyNzExMzgwM1owHTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMFkw
 EwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEuEHJCsUrgvPWX0Mk1j/jNNUFHCUUUhxv
 6GIRRFORTqEiRoMWYCXlBVIJRNx4k/zQrHY/Ajlgwk6j/SM9qBA586NTMFEwHQYD
 VR0OBBYEFJaMKA22eKiMXGvSojeoLGChcABcMB8GA1UdIwQYMBaAFJaMKA22eKiM
 XGvSojeoLGChcABcMA8GA1UdEwEB/wQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIg
 N+a6RbvOz+32qOOgKnbQB8sSVeD0gvRoRK13ZuducX4CIQDkgzc1BHoQJ6Pby3aO
 byBmhjJS/LhhROgzecP/JMDxqA==
 -----END CERTIFICATE-----
```

- ✓ `$ssl_client_escaped_cert` では、クライアント証明書の内容を URL エンコードし、一行で送る。
- ✓ `$ssl_client_cert` では、RFC 2616 (旧 HTTP/1.1) の規則を利用し、クライアント証明書の内容を複数行に分けて送る。
- ✓ RFC 7230 (新 HTTP/1.1) が Line Folding を廃止したため、上位サーバーが RFC 7230 に厳密に従う Web サーバーの場合、`$ssl_client_cert` ではエラーが発生してしまう。

リバースプロキシ

✓ サーバー用の秘密鍵とサーバー証明書を用意する。ここでは適当に自己署名証明書を作成する。

```
$ openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256    > server_private_key.pem #秘密鍵
$ openssl req -x509 -key server_private_key.pem -subj /CN=localhost > server_certificate.pem #証明書
```

✓ リバースプロキシを起動する。ここでは Nginx を利用する。

```
$ vi nginx.conf
$ nginx -c nginx.conf          # 止めるときは nginx -s stop
```

認可サーバー（上位サーバー）

✓ 認可サーバーを起動する。ここでは https://github.com/authlete/java-oauth-server を使う。

```
$ git clone https://github.com/authlete/java-oauth-server
$ cd java-oauth-server
$ vi authlete.properties
$ mvn jetty:run &              # 止めるときは mvn jetty:stop
```

認可サーバーの実装は、リバースプロキシとクライアントとの間の TLS 通信で使用されたクライアント証明書を、リバースプロキシが指定する方法で受け取れなければならない。java-oauth-server は、HTTP ヘッダー `X-Ssl-Cert` に URL エンコードされたクライアント証明書 (PEM) があれば、認識する。

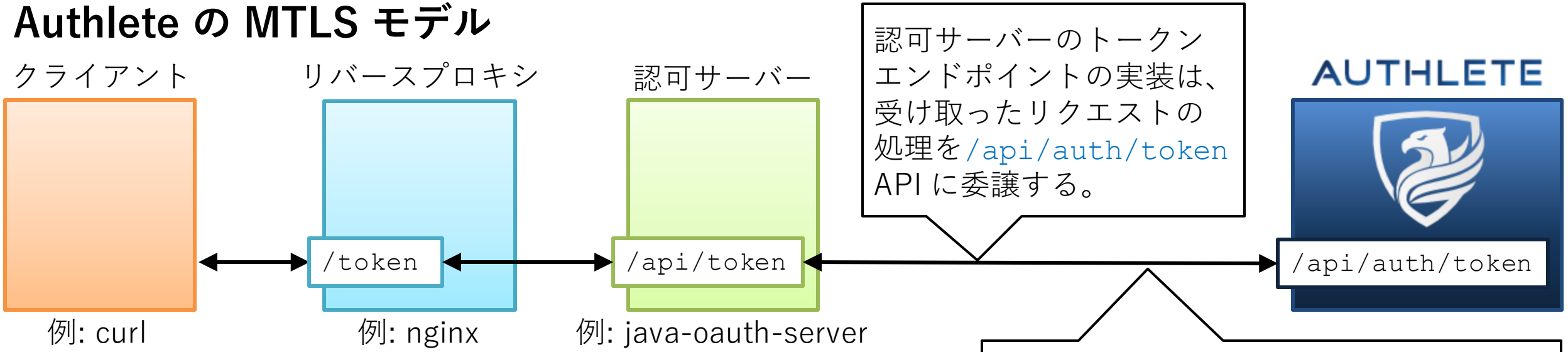
トークンリクエスト（self_signed_tls_client_auth でクライアント認証）

```
$ curl
    -k                                  # サーバー証明書の検証を行わない。サーバー証明書は実験用の自己署名証明書なので。
    --key  /tmp/private_key.pem         # クライアント秘密鍵
    --cert /tmp/certificate.pem         # クライアント証明書
    https://localhost:8443/token        # リバースプロキシ(nginx)が提供するトークンエンドポイント
    -d client_id=4326385670             # トークンリクエストパラメーター client_id
    -d grant_type=client_credentials    # トークンリクエストパラメーター grant_type
```

上位サーバーを直接たたく場合

```
$ curl                                  # クライアント証明書の内容をURLエンコードし、X-Ssl-Certヘッダーの値とする。
    -H 'X-Ssl-Cert:'`php -r "echo rawurlencode(file_get_contents('certificate.pem'));"`
    http://localhost:8080/api/token     # 上位サーバー(java-oauth-server)が提供するトークンエンドポイント
    -d client_id=4326385670             # トークンリクエストパラメーター client_id
    -d grant_type=client_credentials    # トークンリクエストパラメーター grant_type
```

Authlete の MTLS モデル



- ✓ Authlete のアーキテクチャは、クライアント証明書の用途を (1) TLS 接続と (2) RFC 8705 (クライアント認証及び証明書バインディング) に分け、(2) のみをサポートしている。
- ✓ このアーキテクチャにより、どのようなリバースプロキシや API 管理ソリューションを使っているとしても、それがクライアント証明書を取り出す機能さえ持っていれば、RFC 8705 をサポートする認可サーバーを実装することができる。

Justin Richer による解説動画（英語字幕・日本語訳付き）が面白いので、是非見てください！
https://www.authlete.com/ja/resources/videos/20180724/authlete-fapi-enhancements/

*"**Authlete FAPI Enhancements Part 1**" by Justin Richer in July 2018* (8:38~)

https://www.youtube.com/watch?v=hYhHan5FzlA&t=518

But remember in the Authlete model. We're talking to an API. So what do we do for mutual TLS in the Authlete model?

The client is going to do mutual TLS authentication to the authorization server. Then the authorization server is basically just going to say, "Hey, here's the certificate that I was presented as part of that incoming thing. I'm gonna hand that over to the API."

And then the API is effectively gonna be acting as my certificate authority. It's going to figure out does this certificate makes sense for this client for this context.

Now here's what I find particularly interesting about this model.

Like I said, TLS assumes that it's one computer talking to another. In today's environment, with cloud services and microservices and Docker containers and stuff like that. What does computer even mean anymore?

We've got one system talking to another system. Chances are you're going through a TLS terminator, reverse proxies, API gateways and a whole stack of other things.

This notion of we've got point-to-point encryption and it's fully authenticated and fully validated in both directions. It doesn't match the deployment model of computers that we have today.

So the kinds of things that we're doing here, out of the box from a TLS perspective, it looks kind of weird. But I would argue that's because TLS is all slow at keeping up with how computers are being deployed these days.

いま一度、Authlete のモデルを思い出してください。認可サーバーは Authlete API と通信することになります。Authlete モデルでは、双方向 TLS をどのように取り扱っているのでしょう?

クライアントは認可サーバーに対して双方向 TLS 認証を実行します。すると認可サーバーは Authlete API に対して、「クライアントからの通信の中に証明書があったんだけど。これを渡すから、あとはよろしく」と言うことになります。

そして Authlete API は、認可サーバーにとっての認証局として機能します。Authlete は「このコンテキストにおいて、このクライアントがこの証明書を用いることは、適切かどうか」を確かめることになります。

ここからが、このモデルの面白いところです。

先にお話しした通り、TLS は 1 台のコンピューターが別のコンピューターと通信することを前提としています。 今日の環境では、クラウドサービス、マイクロサービス、Docker コンテナーなどがあります。そのような中、”コンピューター” とはどういう意味でしょうか？

あるシステムが別のシステムと通信するとしましょう。その過程では、TLS ターミネーター、リバースプロキシー、API ゲートウェイ、その他いろいろなスタックを通過している可能性があります。

TLS が当初意図していた、ポイント・ツー・ポイントの暗号化、双方向での完全な認証と検証は、今日のコンピューターのデプロイメントモデルとは実際のところ一致しません。

Authlete API が行っていることとは、従来の TLS の観点にはないもので、一見すると奇妙に思えます。しかしわたしには、これは最近のコンピューターのデプロイ手法の進展に TLS が追いついていないのだと思えます。

https://www.authlete.com/ja/resources/videos/20180724/authlete-fapi-enhancements/

FAPI Basics Supplement: Integration with Reference Implementations

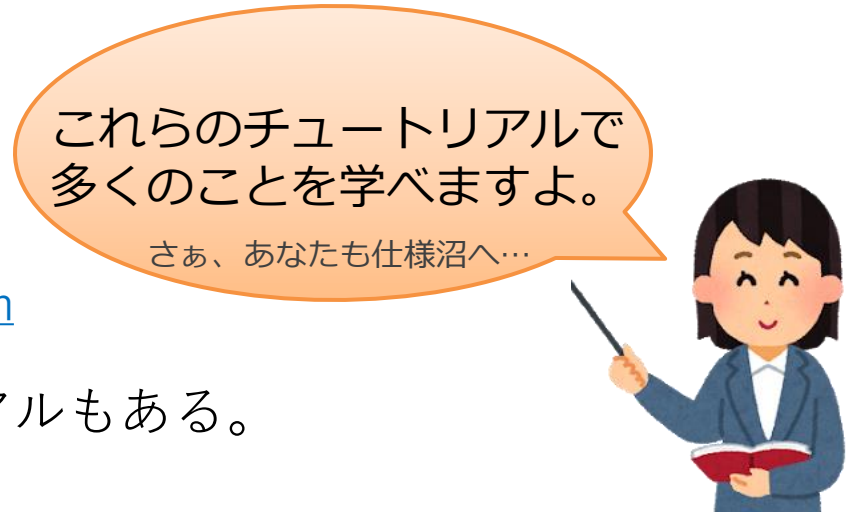
https://www.authlete.com/ja/developers/tutorial/fapi_with_ref_impl/

- ✓ Financial-grade API (FAPI) 準拠の認可フローと API リクエストを試すチュートリアル
- ✓ 認可コードと ID トークンの発行を伴うハイブリッドフローで、`tls_client_auth` によるクライアント認証のほか、証明書に紐付くアクセストークン（certificate-bound access token）の生成とその利用方法についても解説している。

証明書に紐付くアクセストークンについての動画による解説↓

OAuth & OIDC 勉強会【アクセストークン編】5. Proof of Possesion
`https://www.youtube.com/watch?v=ampSEBW6vKM&list=PLxDcFnLrbxvafUBu2GtX35G-iiktmZhWa&index=6`
`https://www.authlete.com/ja/resources/videos/20200422/` （資料ダウンロード）

- ✓ リバースプロキシとして Apache を使う方法を紹介している。



Authlete ナレッジベース: クライアント認証

https://kb.authlete.com/ja/s/oauth-and-openid-connect/t/client-auth

- ✓ `client_secret_jwt` や `tls_client_auth` のチュートリアルもある。

まとめ


OAuth 2.0
OpenID Connect
Authorization Focused • Reliable and Scalable • Developer Friendly
Faster Time to Market • Choice of Hosting Options • Broad Usage
Integrates with any Authentication methods
API Security
OpenID
AUTHLETE

クライアント認証方式	仕様	正当性証明方法	情報形式	情報提示方法
client_secret_post	RFC 6749	クライアントシークレットを提示	直値	client_secret
client_secret_basic	RFC 6749	クライアントシークレットを提示	直値	BASIC 認証
client_secret_jwt	RFC 7523	クライアントシークレットで署名	JWT	client_assertion
private_key_jwt	RFC 7523	秘密鍵で署名	JWT	client_assertion
tls_client_auth	RFC 8705	PKI 証明書を提示	証明書	Mutual TLS
self_signed_tls_client_auth	RFC 8705	自己署名証明書を提示	証明書	Mutual TLS

Financial-grade API (FAPI)

クライアント認証方式	Part 1	Part 2
client_secret_basic	×	×
client_secret_post	×	×
client_secret_jwt	○	×
private_key_jwt	○	○
tls_client_auth	○	○
self_signed_tls_client_auth	○	○

- ✓ クライアントアサーションの署名アルゴリズムに RSA を使う場合、鍵長は 2048 ビット以上。
- ✓ クライアントアサーションの署名アルゴリズムに楕円曲線を使う場合、鍵長は 160 ビット以上。
- ✓ FAPI Part 2 では、クライアントアサーションの署名アルゴリズムは PS256 または ES256 のみ。

Thank You

お問い合わせ

https://www.authlete.com/ja/contact/

一般	info@authlete.com
営業	sales@authlete.com
広報	pr@authlete.com
技術	support@authlete.com

@authlete_jp