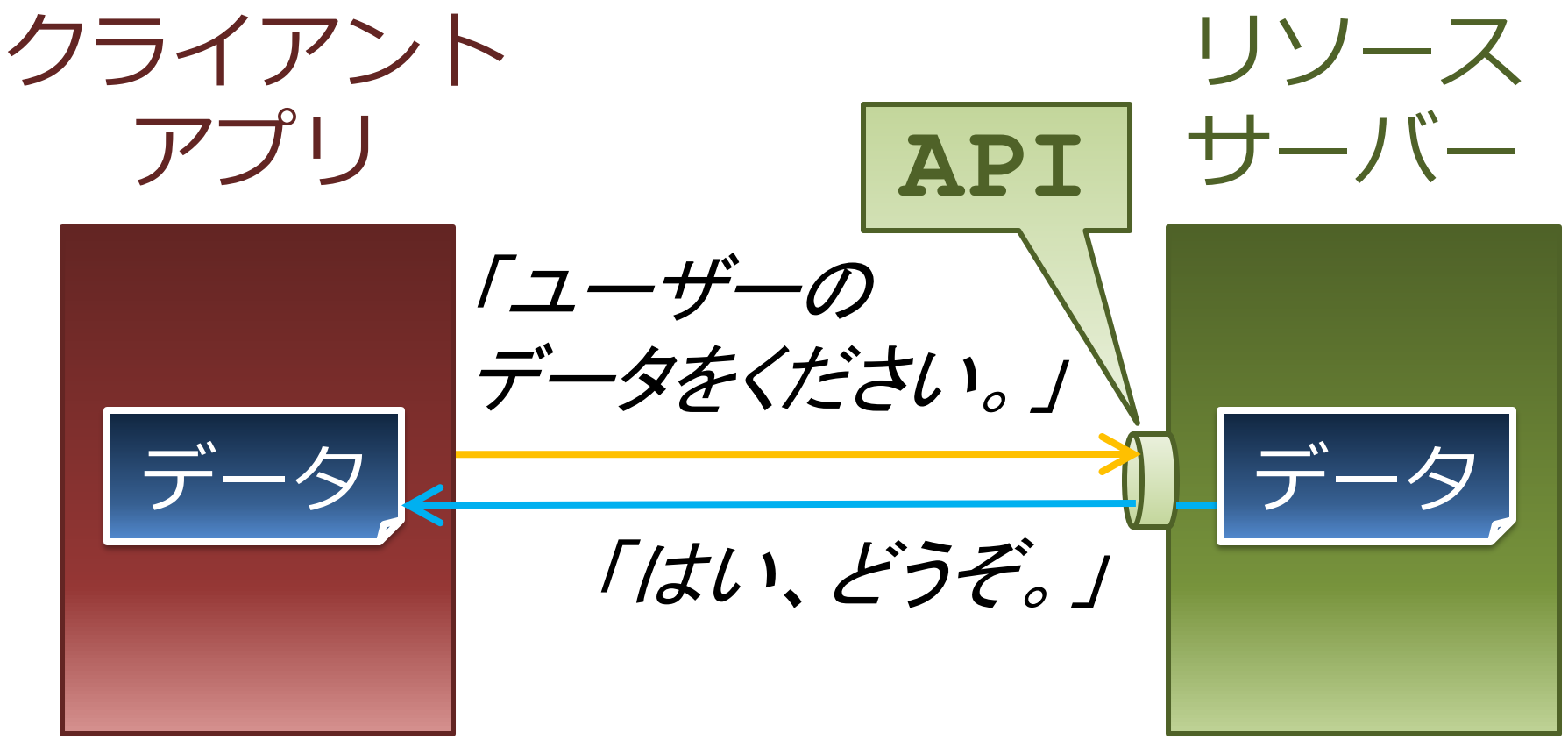
OAuth & OIDC 入門編

Online Session on March 17, 2020

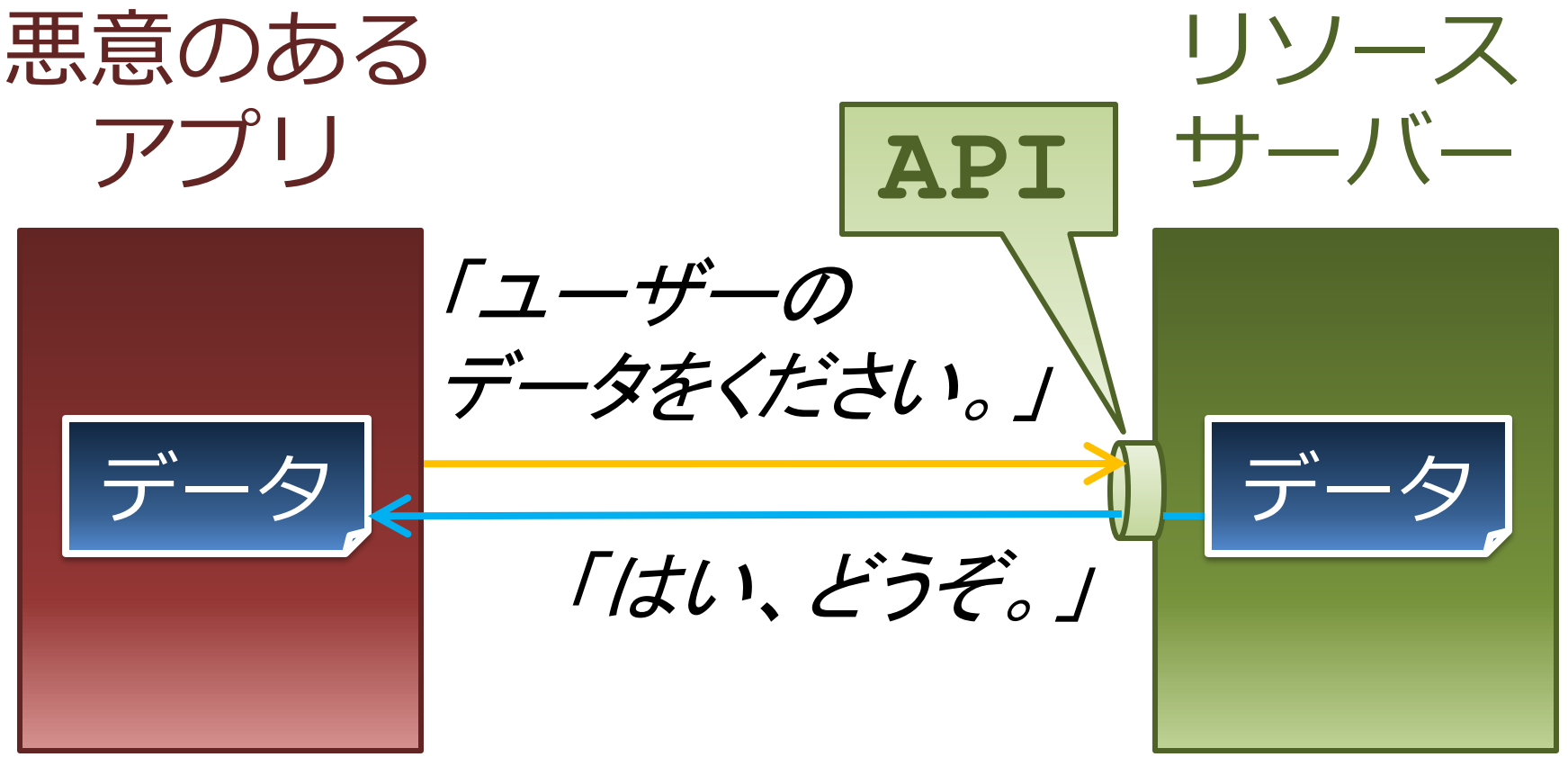
Co-founder, Authlete, Inc.
Takahiko Kawasaki <taka@authlete.com>

OAuth 2.0 概要


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

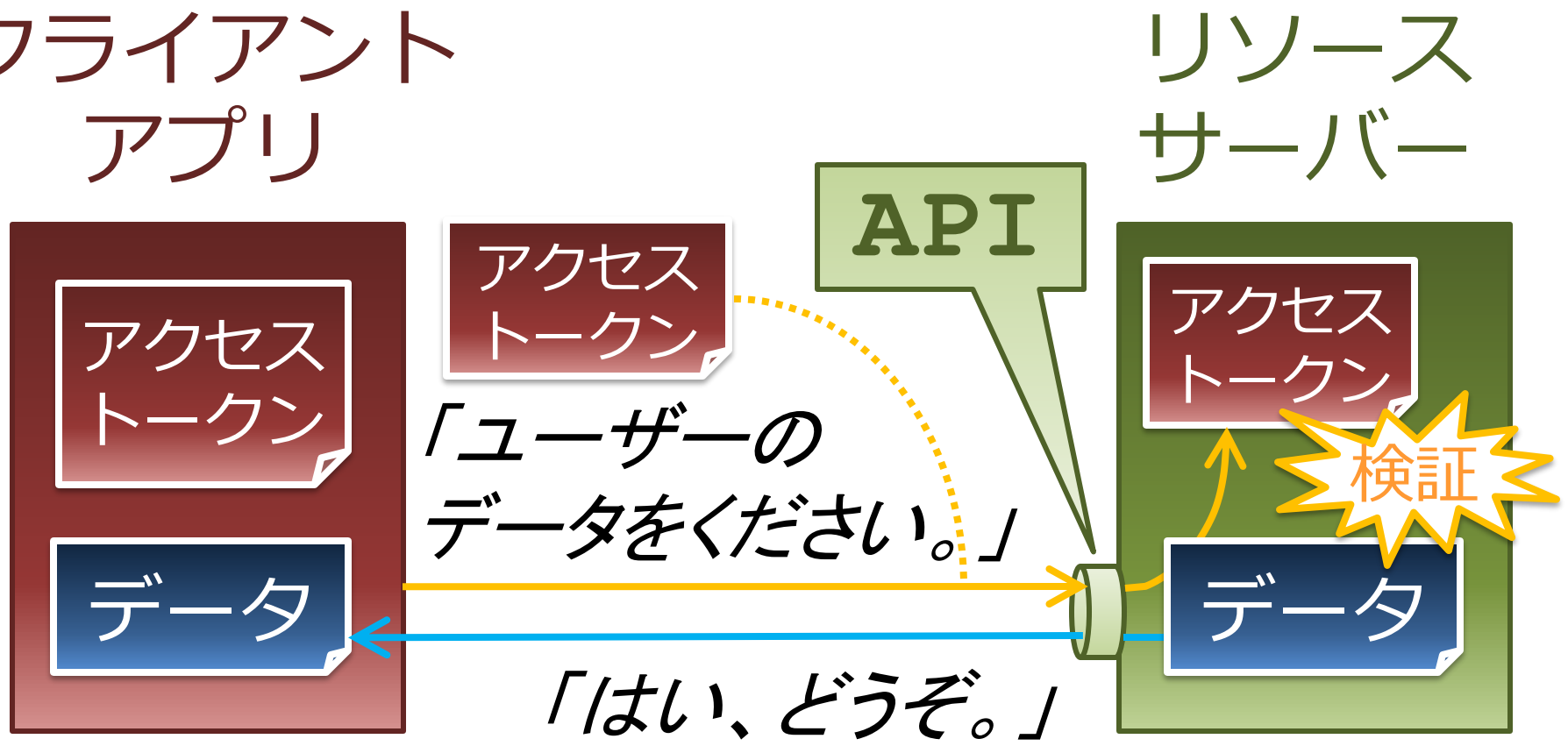

クライアント
アプリ
リソース
サーバー
API
「ユーザーの
データをください。」
データ
データ
「はい、どうぞ。」

ＡＰＩを守る仕組みが必要！



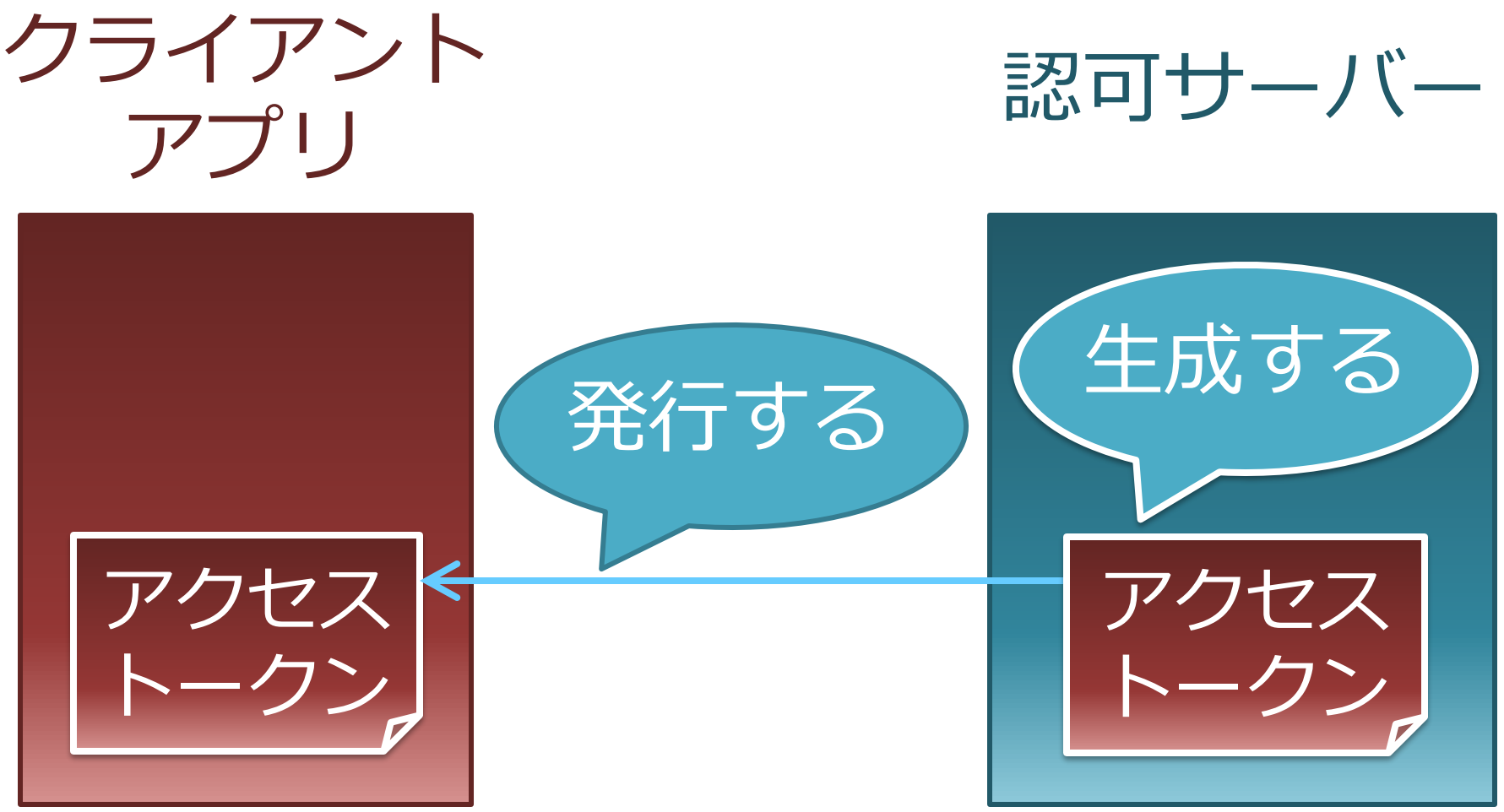
悪意のあるアプリにも
データが渡ってしまう！

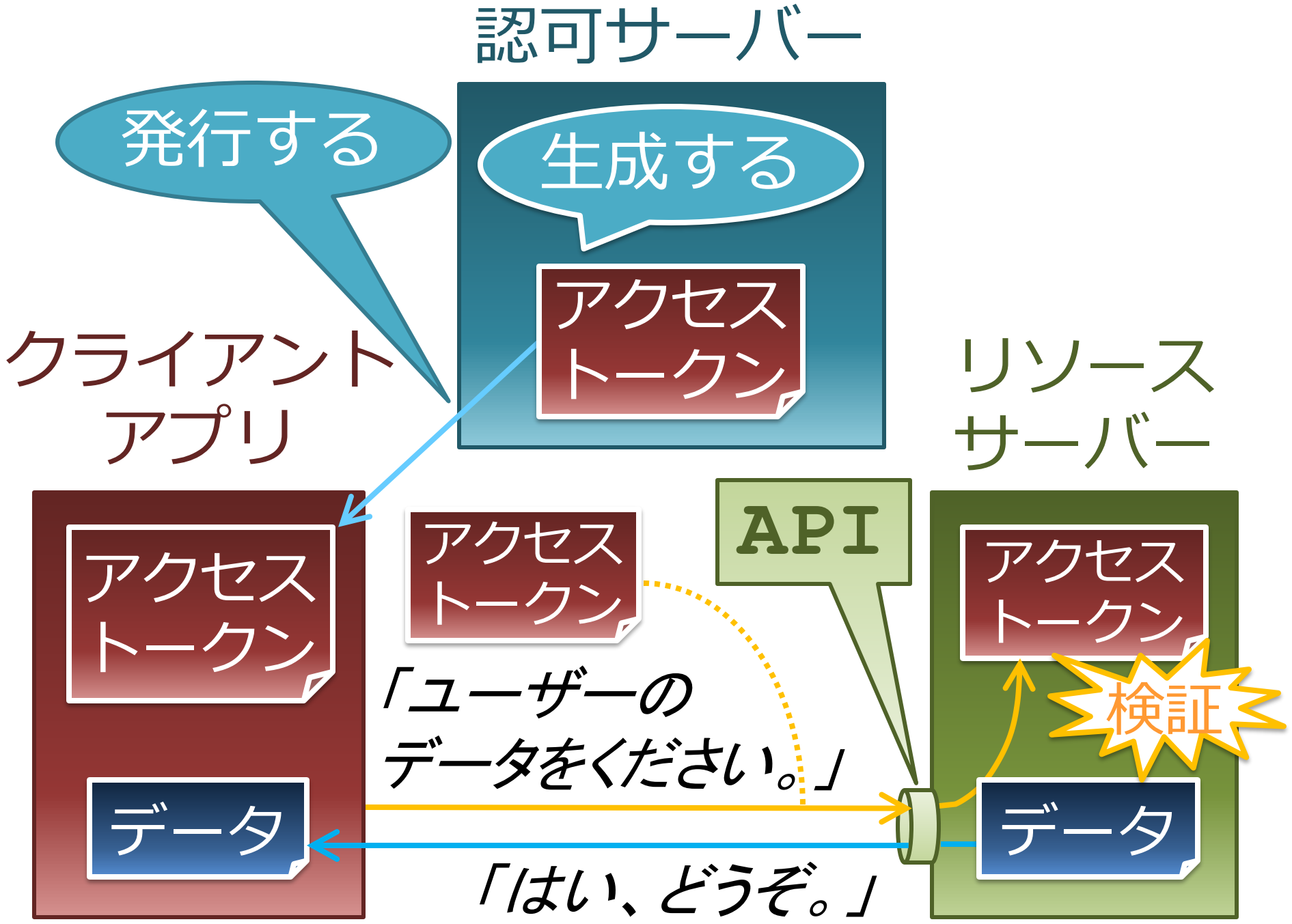
アクセストークンを発行する係が必要！

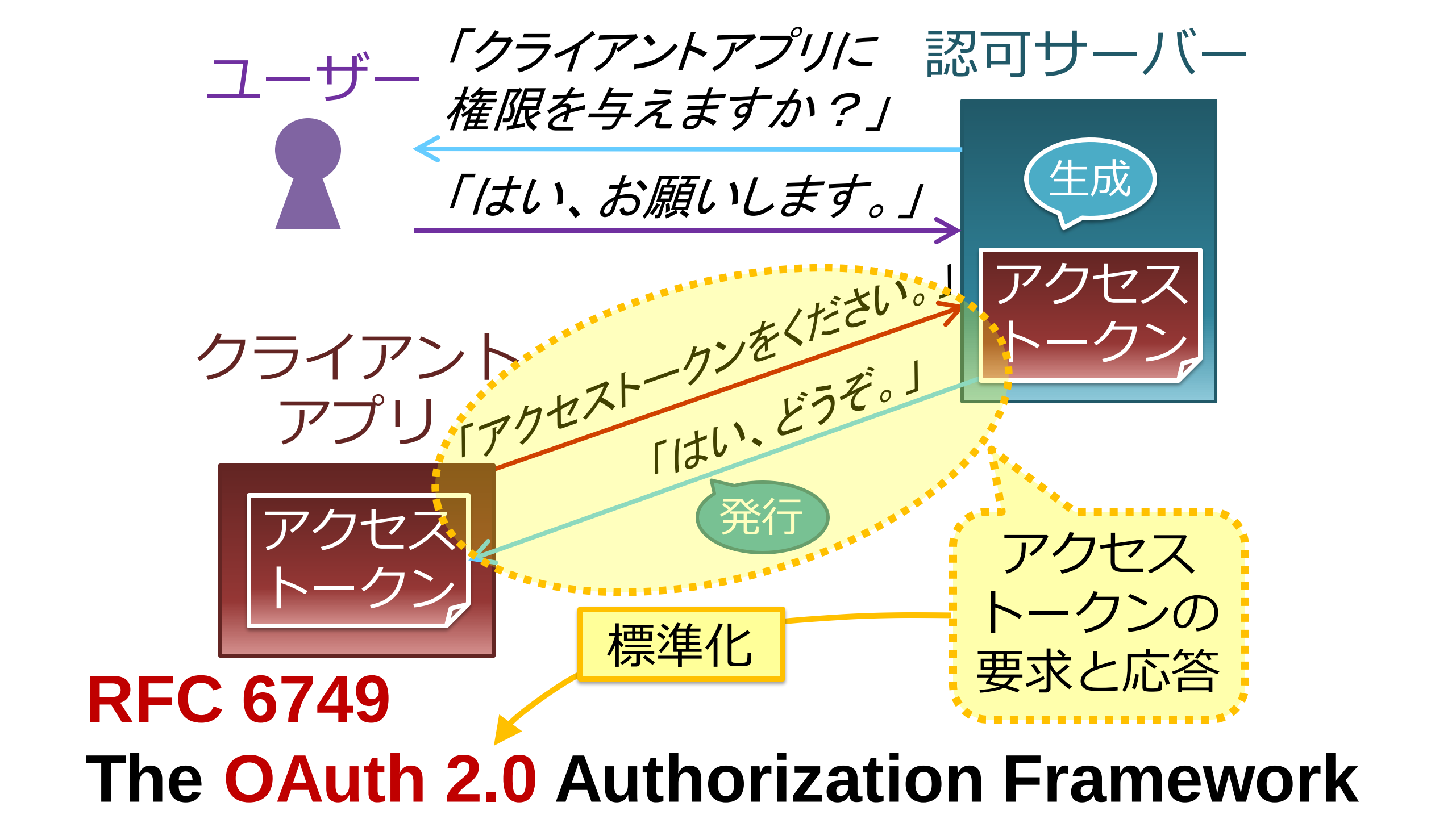


予めクライアントにアクセス
トークンを渡しておく必要あり

アクセストークンを
発行する係
||
認可サーバー

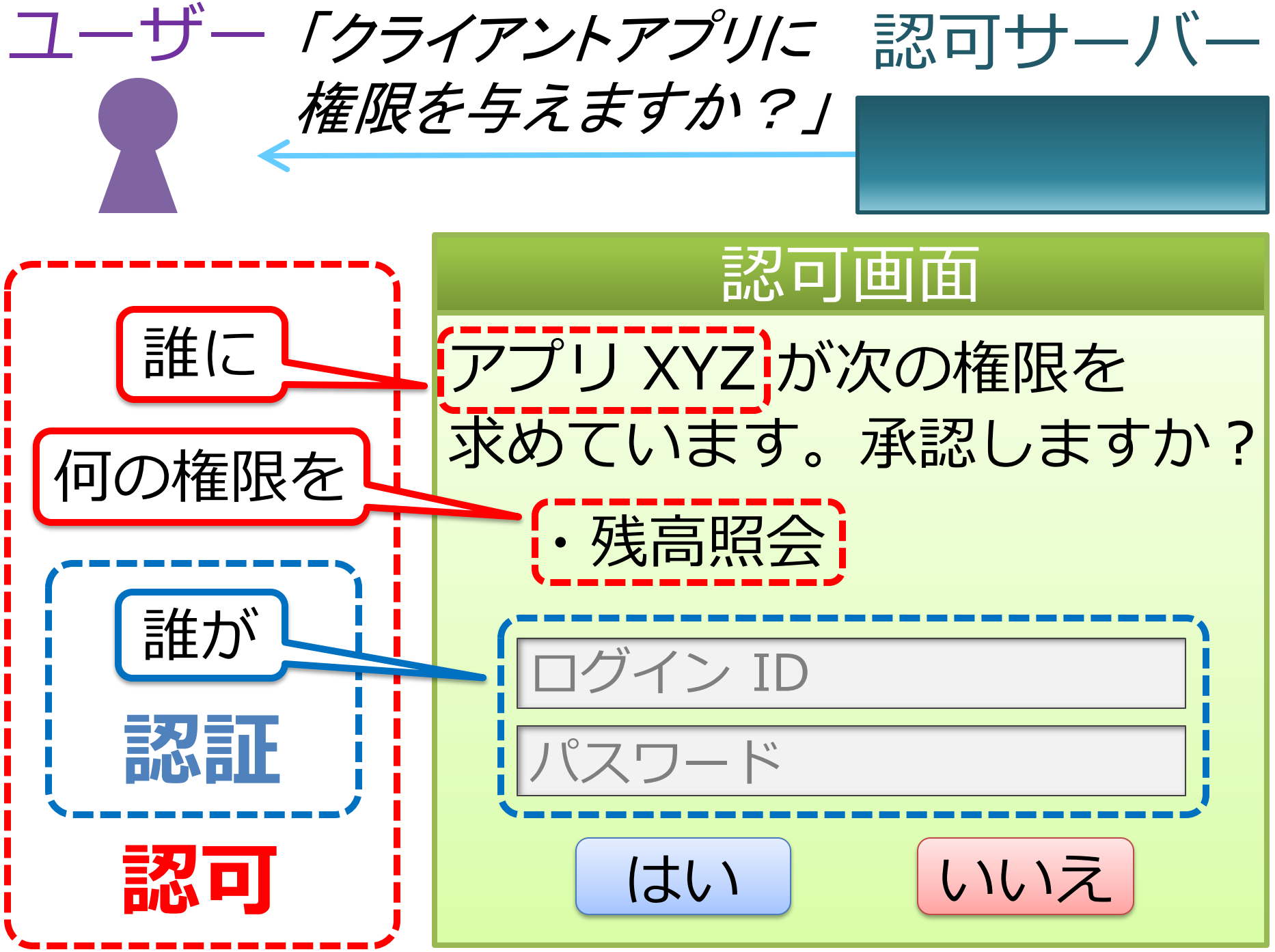

クライアント
アプリ
認可サーバー
発行する
生成する
アクセス
トークン
アクセス
トークン


認可サーバー
発行する
生成する
アクセス
トークン
クライアント
アプリ
リソース
サーバー
アクセス
トークン
アクセス
トークン
API
アクセス
トークン
検証
「ユーザーの
データをください。」
データ
データ
「はい、どうぞ。」



RFC 6749

The OAuth 2.0 Authorization Framework

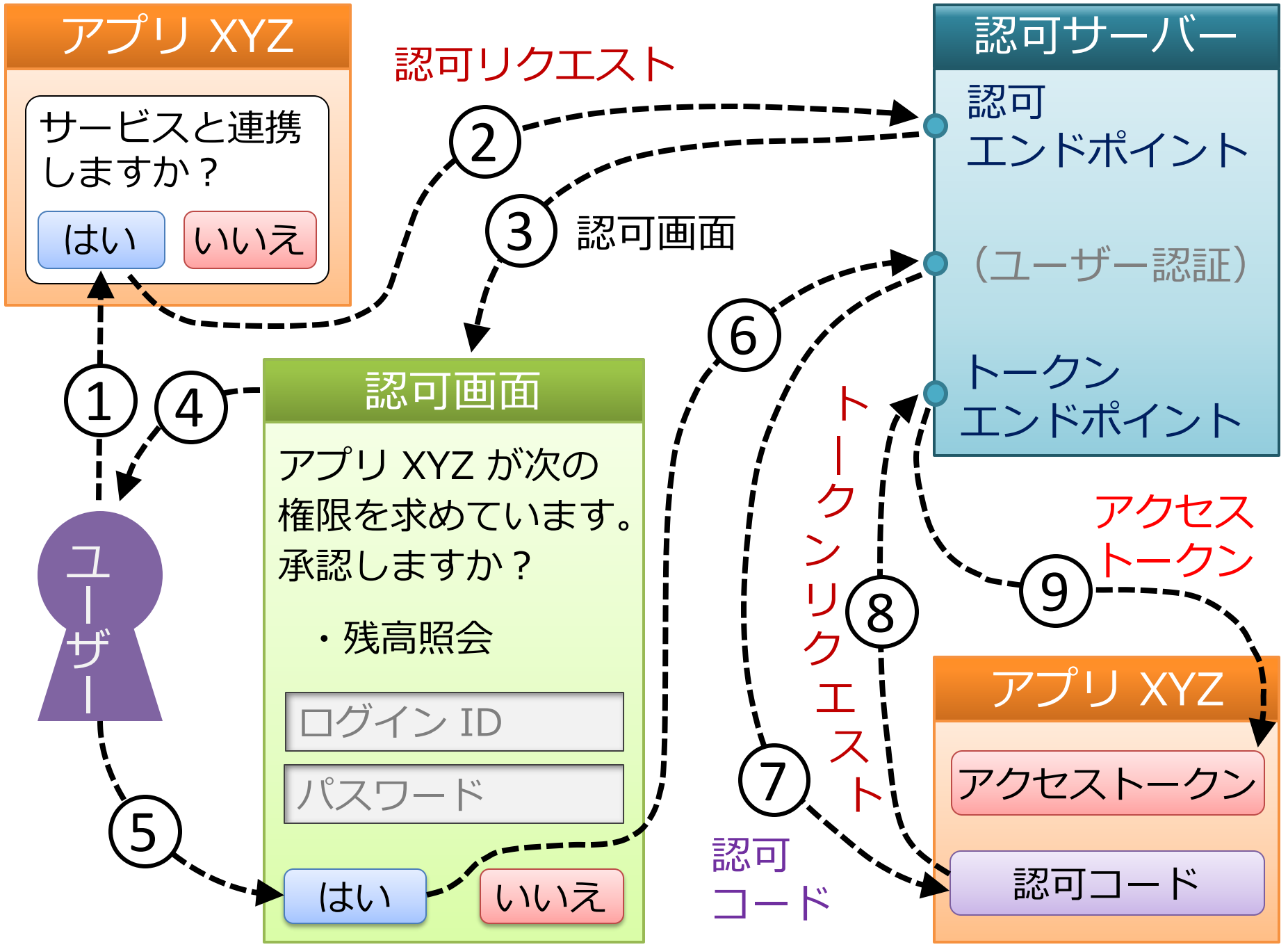

ユーザー
「クライアントアプリに
権限を与えますか？」
認可サーバー
認可画面
誰に
何の権限を
誰が
認証
認可
アプリ XYZ が次の権限を
求めています。承認しますか？
・残高照会
ログイン ID
パスワード
はい
いいえ

RFC 6749 は四つの認可フローを定義しています。

→ アクセストークン発行手順

	認可フロー名	特徴
1	認可コード	一時的に発行される認可コードを アクセストークンと交換する。
2	インプリシット	認可エンドポイントからアクセス トークンが直接発行される。
3	リソースオーナー・パス ワード・クレデンシャルズ	ユーザーのIDとパスワードを クライアントアプリに渡す。
4	クライアント・クレデン シャルズ	ユーザー認証無し。クライアント アプリの認証のみ必要。

アクセストークン再発行のための「リフレッシュトークンフロー」も定義されている。

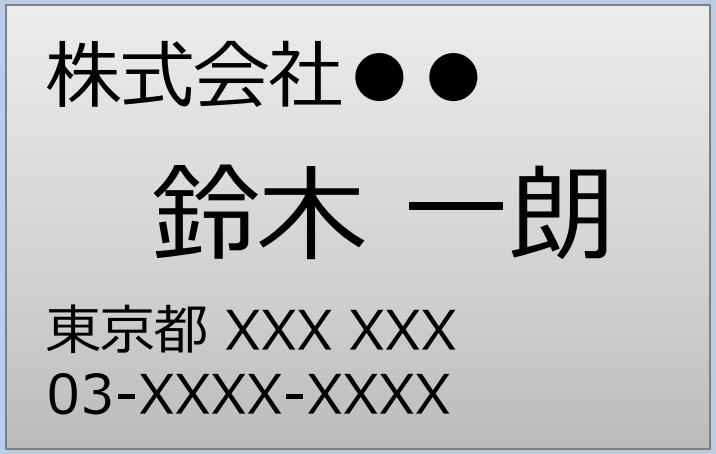

アプリ XYZ
サービスと連携しますか？
はい
いいえ
認可リクエスト
2
3
認可画面
認可サーバー
認可
エンドポイント
（ユーザー認証）
トークン
エンドポイント
1
4
認可画面
アプリ XYZ が次の権限を求めています。承認しますか？
・残高照会
ログイン ID
パスワード
はい
いいえ
ユーザー
5
6
トークンリクエスト
8
9
アクセストークン
7
認可コード
アプリ XYZ
アクセストークン
認可コード

OpenID Connect 概要

こんにちは！ 鈴木一朗です！

本当ですか？ 証明してください。

はい！ これが私の名刺です！



それでは証明になりません。
名刺は誰でも偽造できてしまうので。

こんにちは！ 鈴木一朗です！ 会社の『**署名**』入りの名刺を持ってきました！

株式会社●●

鈴木 一朗

東京都 XXX XXX
03-XXXX-XXXX

株式会社
●●

確認しますのでお待ちください。

こんにちは！ 鈴木一朗です！ 会社の『**署名**』入りの名刺を持ってきました！

株式会社●●

鈴木 一朗

東京都 XXX XXX 株式会社●●
03-XXXX-XXXX

確認しますのでお待ちください。

株式会社●●さん。

はい。
何かご用でしょうか？

御社が発行した名刺についている『**署名**』が本物かどうか確認したいので、『**公開鍵**』をください。

はい、どうぞ。

こんにちは！ 鈴木一朗です！ 会社の『**署名**』入りの名刺を持ってきました！

株式会社●●

鈴木 一朗

東京都 XXX XXX
03-XXXX-XXXX

株式会社●●

確認しますのでお待ちください。

株式会社●●様が発行された名刺であることを確認できました。

株式会社●●さん。

はい。
何かご用でしょうか？

御社が発行した名刺についている『**署名**』が本物かどうか確認したいので、『**公開鍵**』をください。

はい、どうぞ。

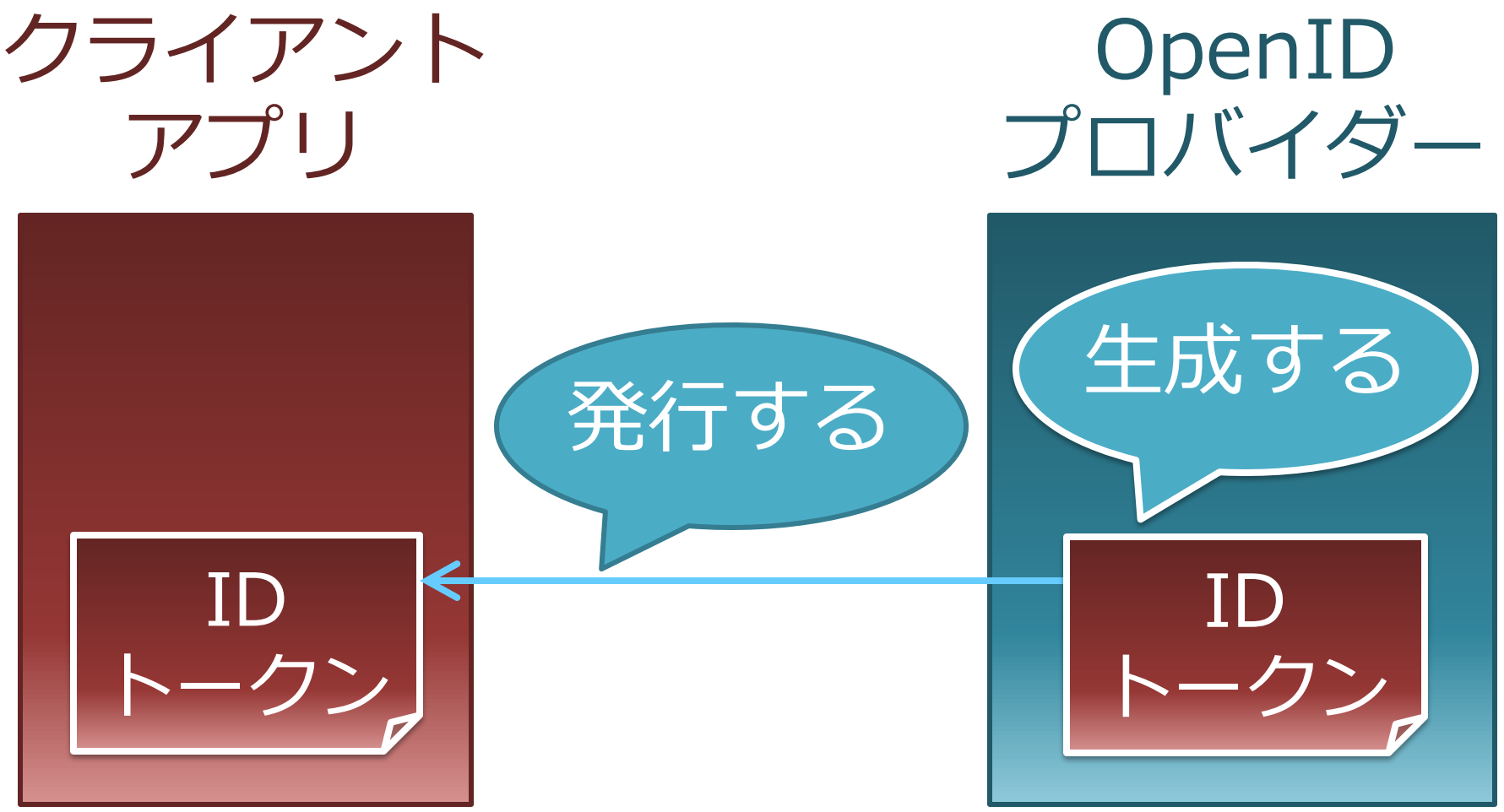
『発行者の署名付き名刺』

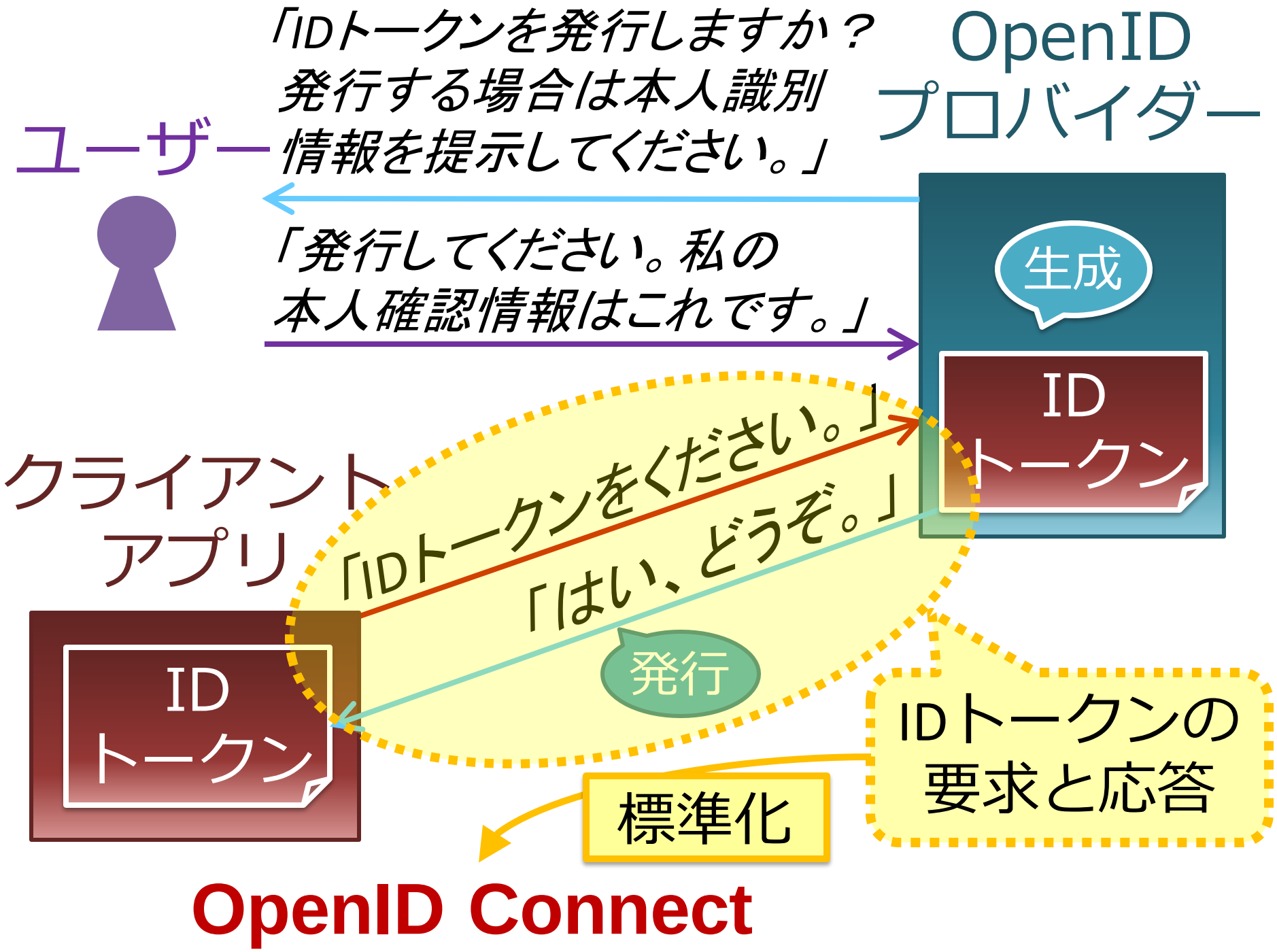
この概念に相当するもの

↓

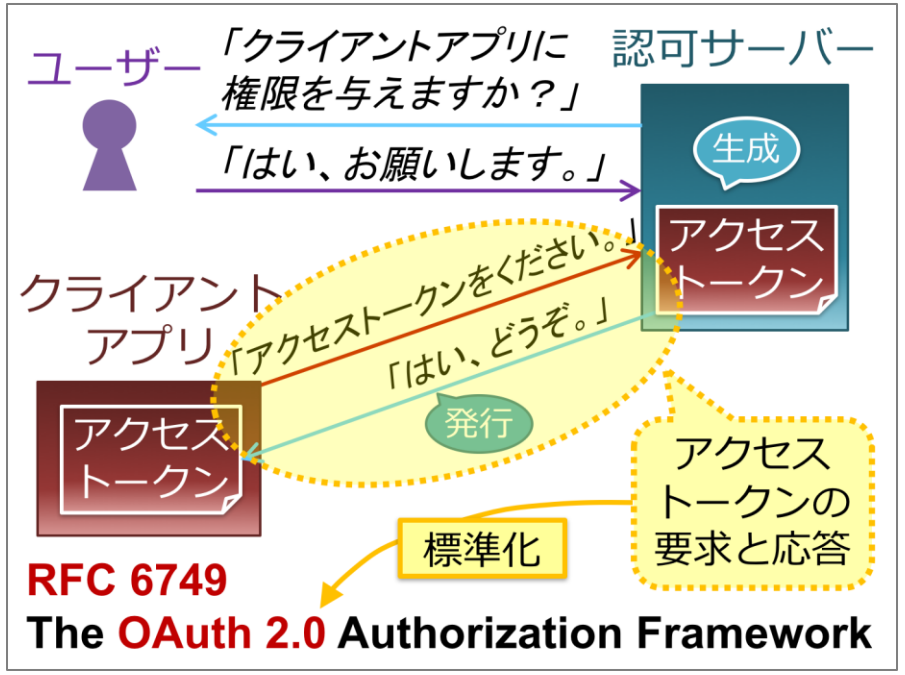
IDトークン

ＩＤトークンを
発行する係
||
OpenID
プロバイダー

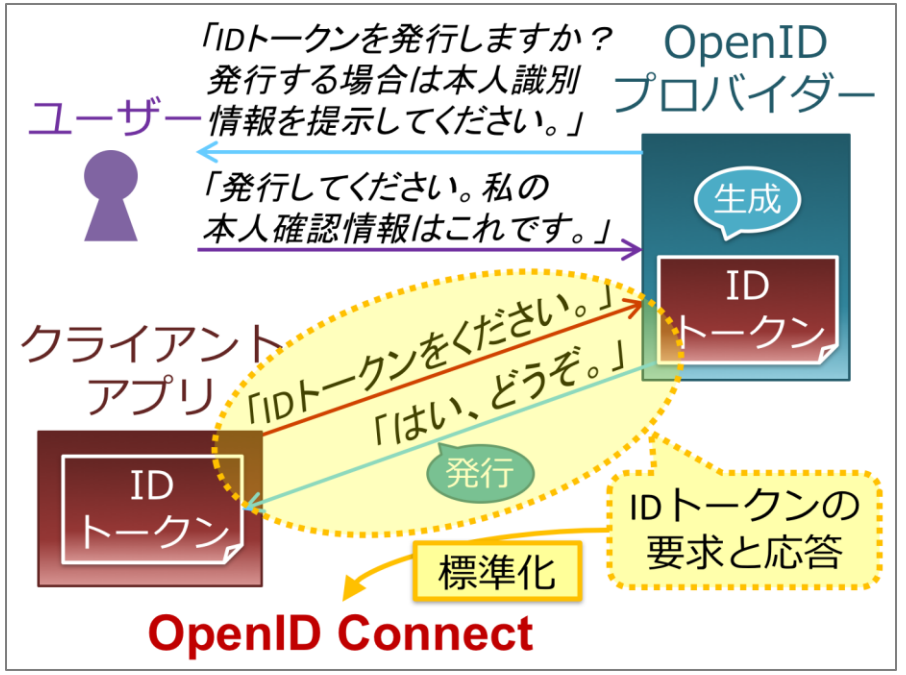

クライアント
アプリ
OpenID
プロバイダー
発行する
生成する
ID
トークン
ID
トークン


「IDトークンを発行しますか？
発行する場合は本人識別
情報を提示してください。」
OpenID
プロバイダー
ユーザー
「発行してください。私の
本人確認情報はこれです。」
生成
ID
トークン
クライアント
アプリ
「IDトークンをください。」
「はい、どうぞ。」
発行
ID
トークン
IDトークンの
要求と応答
標準化
OpenID Connect

OAuth 2.0



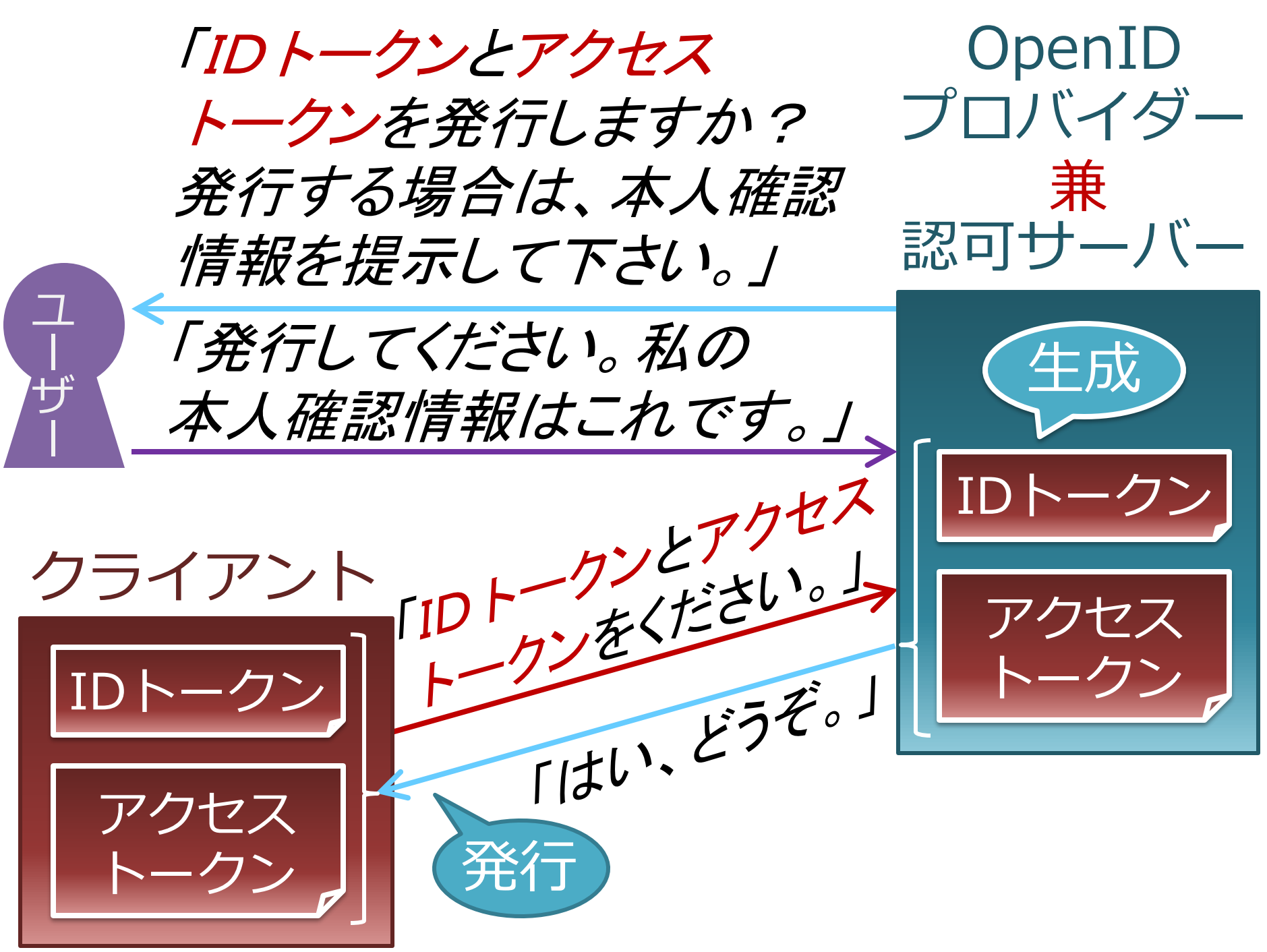
OpenID Connect



似てる・・・

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol

http://openid.net/connect/


「IDトークンとアクセストークンを発行しますか？発行する場合は、本人確認情報を提示して下さい。」
OpenID
プロバイダー
兼
認可サーバー
ユーザー
「発行してください。私の本人確認情報はこれです。」
生成
IDトークン
アクセストークン
クライアント
「IDトークンとアクセストークンをください。」
「はい、どうぞ。」
IDトークン
アクセストークン
発行

JWS, JWE, JWT

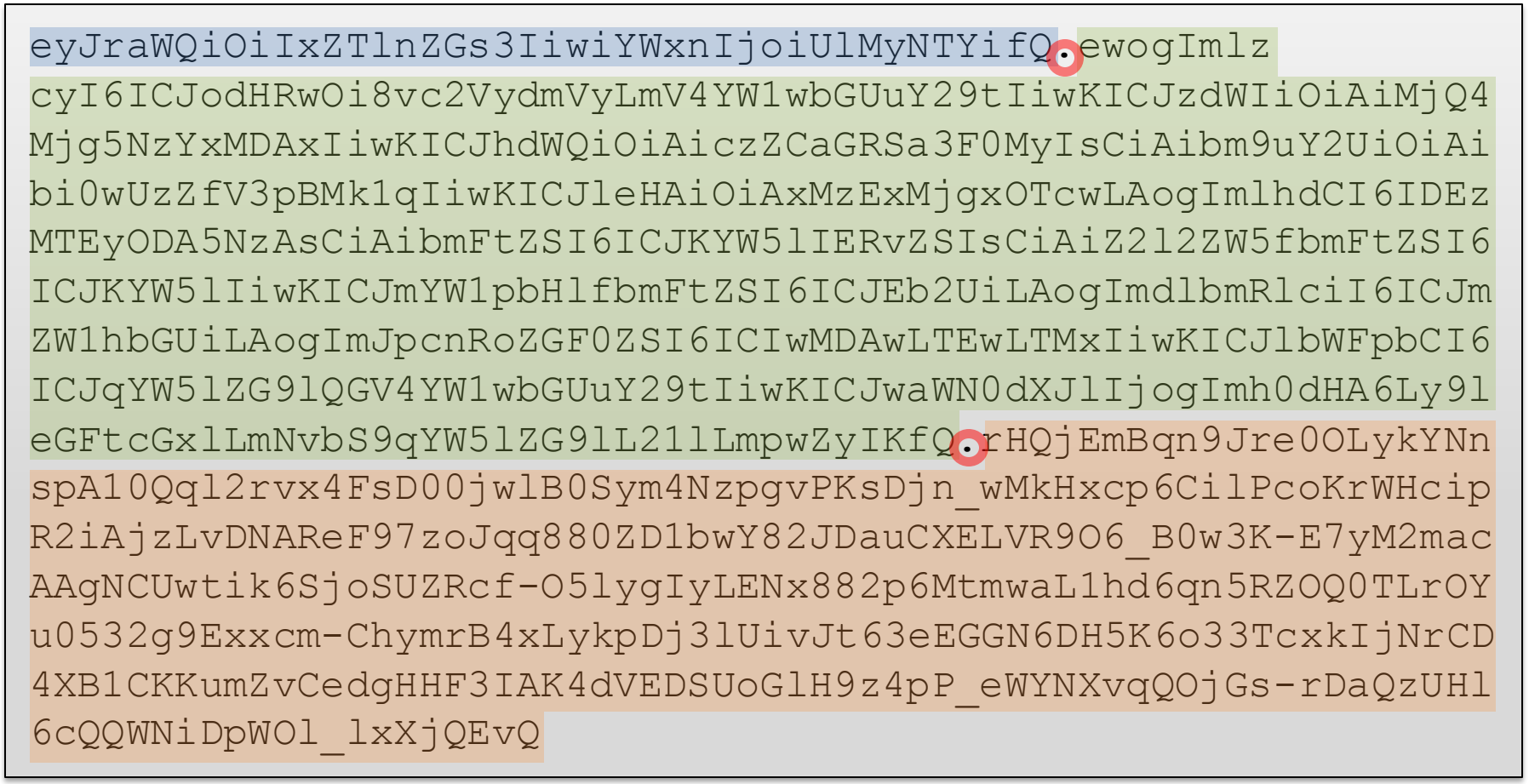

OAuth 2.0
OpenID Connect
Authorization Focused • Reliable and Scalable • Developer Friendly
Faster Time to Market • Choice of Hosting Options • Broad Usage
Integrates with any Authentication methods
API Security
OpenID
AUTHLETE

仕様書	省略形	名称
RFC 7515	**JWS**	JSON Web Signature
RFC 7516	**JWE**	JSON Web Encryption
RFC 7517	**JWK**	JSON Web Key
RFC 7518	**JWA**	JSON Web Algorithms
RFC 7519	**JWT**	JSON Web Token

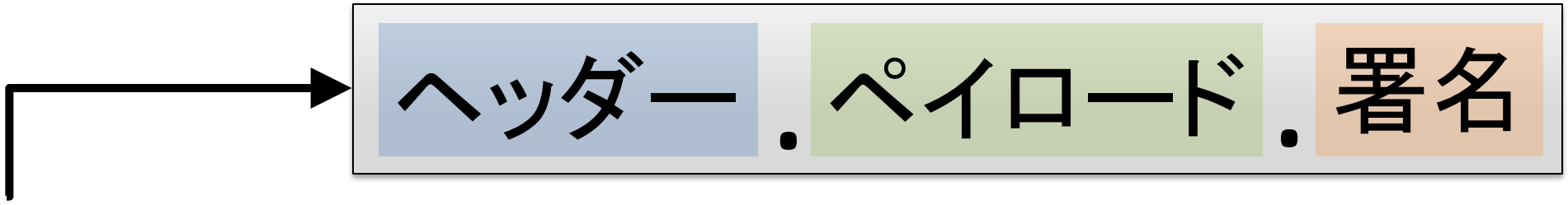
2014 年、

『Special European Identity Award for Best Innovation for Security in the API Economy』
（European Identity Award - API エコノミーと最高のセキュリティ・イノベーション特別賞）

受賞



(*“OpenID Connect Core 1.0, A.2. Example using response_type=id_token” より抜粋*)



“RFC 7515 (JSON Web Signature (JWS)), 7.1. JWS Compact Serialization”

```
BASE64URL(UTF8(JWS Protected Header)) || '.' ||
BASE64URL(JWS Payload) || '.' ||
BASE64URL(JWS Signature)
```

ヘッダー
ペイロード
署名

各部分は base64url で
エンコードされている。

JWS形式の
ID トークン

base64url でデコード

base64url でデコード

```
eyJraWQiOiIxZTlnZGs3IiwiYWxnIjoiUlMyNTYifQ.ewogImlz
cyI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4
Mjg5NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAi
bi0wUzZfV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEz
MTEyODA5NzAsCiAibmFtZSI6ICJKYW5lIERvZSIsCiAiZ2l2ZW5fbmFtZSI6
ICJKYW5lIiwKICJmYW1pbHlfbmFtZSI6ICJEb2UiLAogImdlbmRlciI6ICJm
ZW1hbGUiLAogImJpcnRoZGF0ZSI6ICIwMDAwLTEwLTMxIiwKICJlbWFpbCI6
ICJqYW5lZG9lQGV4YW1wbGUuY29tIiwKICJwaWN0dXJlIjogImh0dHA6Ly9l
eGFtcGxlLmNvbS9qYW5lZG9lL21lLmpwZyIKfQ.rHQjEmBqn9Jre0OLykYNn
spA10Qql2rvx4FsD00jwlB0Sym4NzpgvPKsDjn_wMkHxcp6CilPcoKrWHcip
R2iAjzLvDNAReF97zoJqq880ZD1bwY82JDauCXELVR9O6_B0w3K-E7yM2mac
AAgNCUwtik6SjoSUZRcf-O5lygIyLENx882p6MtmwaL1hd6qn5RZOQ0TLrOY
u0532g9Exxcm-ChymrB4xLykpDj3lUivJt63eEGGN6DH5K6o33TcxkIjNrCD
4XB1CKKumZvCedgHHF3IAK4dVEDSUoGlH9z4pP_eWYNXvqQOjGs-rDaQzUHl
6cQQWNiDpWOl_lxXjQEvQ
```

“OpenID Connect Core 1.0, A.2. Example using response_type=id_token”

base64url でデコード

```
{"kid":"1e9gdk7","alg":"RS256"}
```

```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "gender": "female",
 "birthdate": "0000-10-31",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```

（バイナリデータ）

ヘッダー `{"kid":"1e9gdk7","alg":"RS256"}`

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

JWS のヘッダー部分は JSON です。

alg は 署名アルゴリズムを表す必須のパラメーター。

alg を含む全てのパラメーターは RFC 7515 (JWS) に列挙されています。

```
1.alg   7.x5t
2.jku   8.x5t#S256
3.jwk   9.typ
4.kid   10.cty
5.x5u   11.crit
6.x5c
```

RFC 7518 (JSON Web Algorithms (JWA)), 3.1. “alg” (Algorithm) Header Parameter Values for JWS

ペイロード（JWS）

任意のバイト列

ペイロード（IDトークン）


```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "gender": "female",
 "birthdate": "0000-10-31",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```


JWS

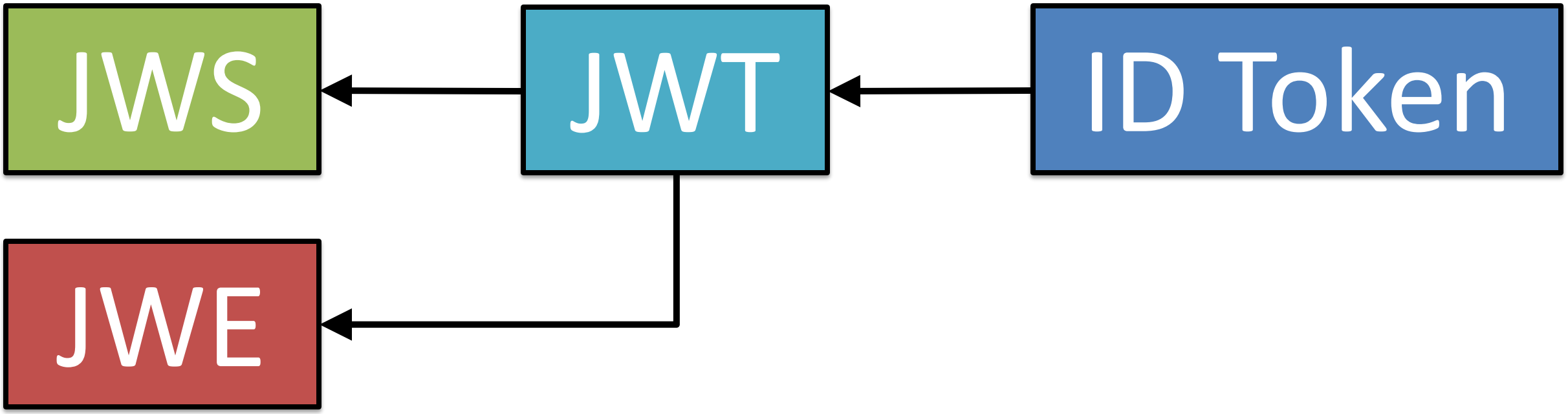
RFC 7515 (JWS) 自体はペイロードが JSON であることを要求しません。

JWT

JSON を要求し、いくつかのパラメーターを定義しているのは、RFC 7519 (JSON Web Token (JWT)) です。

ID Token

OpenID Connect は JWT を拡張し、IDトークンを定義しました。



JWS	JSON Web Signature	署名
JWT	JSON Web Token	トークン
JWE	JSON Web Encryption	暗号

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
6UklfCpIMfIjf7iGdXKHzg.
48V1_ALb6US04U3b.
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
SdiwkIr3ajwQzaBtQD_A.
XFBoMYUZodetZdvTiFvSkQ
```

RFC 7516 (JSON Web Encryption (JWE)), 7.1. JWE Compact Serialization

```
BASE64URL(UTF8(JWE Protected Header)) || '.' ||
BASE64URL(JWE Encrypted Key) || '.' ||
BASE64URL(JWE Initialization Vector) || '.' ||
BASE64URL(JWE Ciphertext) || '.' ||
BASE64URL(JWE Authentication Tag)
```

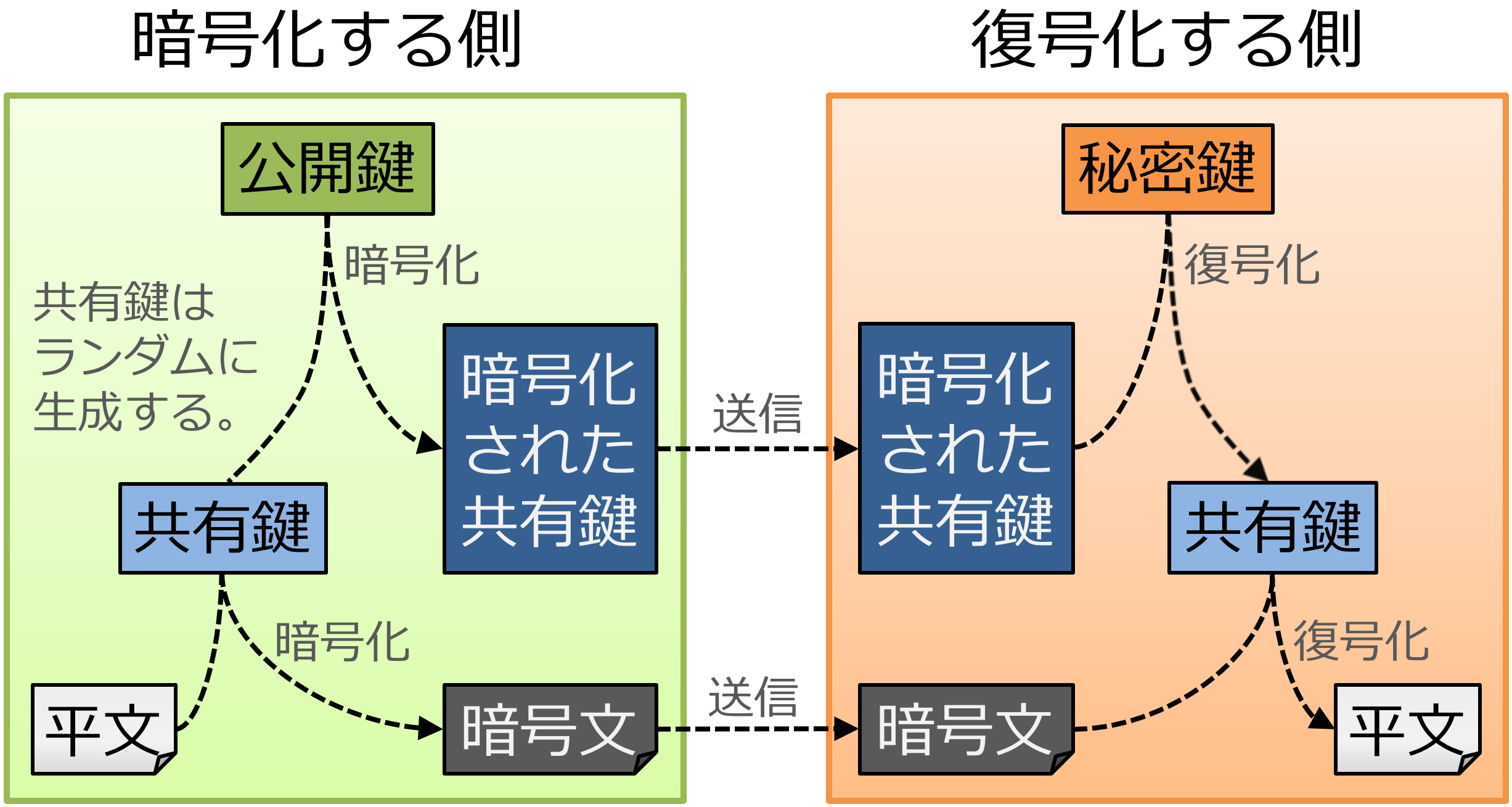
- ヘッダー
- 暗号化されたキー
- 初期ベクター
- 暗号文
- 認証タグ

```
BASE64URL(UTF8(JWE Protected Header)) || '.' ||
BASE64URL(JWE Encrypted Key) || '.' ||
BASE64URL(JWE Initialization Vector) || '.' ||
BASE64URL(JWE Ciphertext) || '.' ||
BASE64URL(JWE Authentication Tag)
```

『暗号化されたキー』（Encrypted Key）？

『暗号キー』（Encryption Key）ではなく？

「共有鍵を非対称鍵で暗号化する」という
二段階の暗号処理をしているため。


暗号化する側
復号化する側
公開鍵
暗号化
共有鍵は
ランダムに
生成する。
暗号化
された
共有鍵
送信
秘密鍵
復号化
共有鍵
暗号化
された
共有鍵
共有鍵
暗号化
復号化
平文
暗号文
送信
暗号文
平文


eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
6UklfCpIMfIjf7iGdXKHzg.
48V1_ALb6US04U3b.
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
SdiwkIr3ajwQzaBtQD_A.
XFBoMYUZodetZdvTiFvSkQ
{"alg":"RSA-OAEP","enc":"A256GCM"}
base64urlでデコード
平文の暗号アルゴリズム
共有鍵の暗号アルゴリズム

共有鍵暗号化のアルゴリズム（RFC 7518, 4.1.）

識別子 (alg)	アルゴリズム
RSA1_5	RSAES-PKCS1-v1_5
RSA-OAEP	RSAES OAEP using default parameters
RSA-OAEP-256	RSAES OAEP using SHA-256 and MGF1 with SHA-256
A128KW	AES Key Wrap with default initial value using 128-bit key
A192KW	AES Key Wrap with default initial value using 192-bit key
A256KW	AES Key Wrap with default initial value using 256-bit key
dir	Direct use of a shared symmetric key as the CEK (Content Encryption Key)
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement using Concat KDF
ECDH-ES+A128KW	ECDH-ES using Concat KDF and CEK wrapped with “A128KW”
ECDH-ES+A192KW	ECDH-ES using Concat KDF and CEK wrapped with “A192KW”
ECDH-ES+A256KW	ECDH-ES using Concat KDF and CEK wrapped with “A256KW”
A128GCMKW	Key wrapping with AES GCM using 128-bit key
A192GCMKW	Key wrapping with AES GCM using 192-bit key
A256GCMKW	Key wrapping with AES GCM using 256-bit key
PBES2-HS256+A128KW	PBES2 with HMAC SHA-256 and “A128KW” wrapping
PBES2-HS384+A192KW	PBES2 with HMAC SHA-384 and “A192KW” wrapping
PBES2-HS512+A256KW	PBES2 with HMAC SHA-512 and “A256KW” wrapping

共有鍵自体を直接本文暗号化に使う。特殊なので後述。

平文暗号化のアルゴリズム（RFC 7518, 5.1.）

識別子 (enc)	アルゴリズム
A128CBC-HS256	AES using 128-bit IV and SHA-256
A192CBC-HS384	AES using 192-bit IV and SHA-384
A256CBC-HS512	AES using 256-bit IV and SHA-512
A128GCM	AES GCM using 128-bit key
A192GCM	AES GCM using 192-bit key
A256GCM	AES GCM using 256-bit key

- 共有鍵暗号アルゴリズムの一つ『dir』

Content Encryption Key

dir	Direct use of a shared symmetric key as the CEK

二段階暗号処理ではなく、直接共有鍵暗号処理をおこなう。

- RFC 7518, “4.5. Direct Encryption with a Shared Symmetric Key”

> *This section defines the specifics of directly performing symmetric key encryption without performing a key wrapping step.*

ただし、RFC 7518 では、共有鍵の決め方のルールは定めていない。

- OpenID Connect Core 1.0, “10.2. Encryption”

> *The symmetric encryption key is derived from the `client_secret` value by using a left truncated SHA-2 hash of the octets of the UTF-8 representation of the `client_secret`.*

「クライアントシークレットのハッシュをとって切り詰めたもの」

JWT JSON Web Token

読み方

*The suggested pronunciation of JWT is the same as the English word "**jot**".* → ジョット

RFC 7519, Introduction

JWTとは？

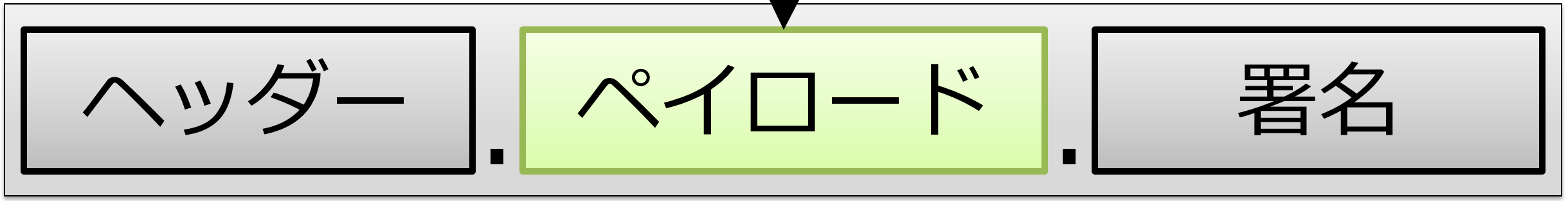
JSON 形式で表現されたクレーム (claim) の集合を JWS もしくは JWE に埋め込んだもの

JWS 形式の JWT

JSON 形式で表現されたクレームの集合

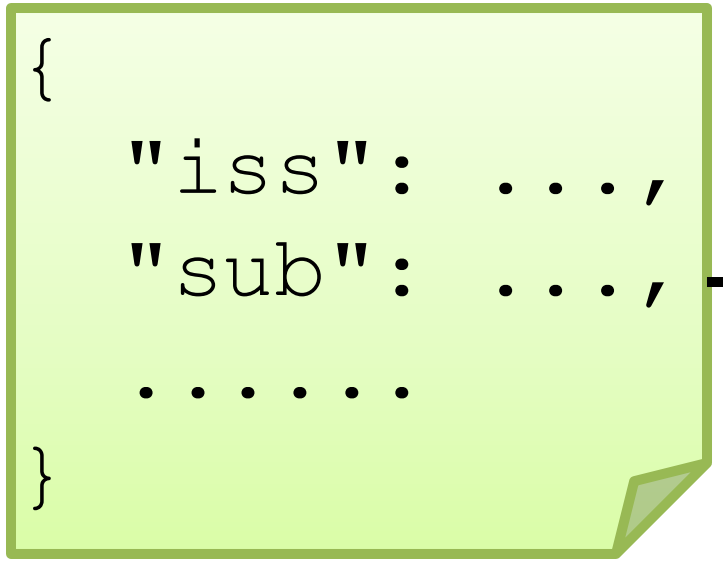
```
{
   "iss": ...,
   "sub": ...,
   ......
}
```

BASE64URL エンコード

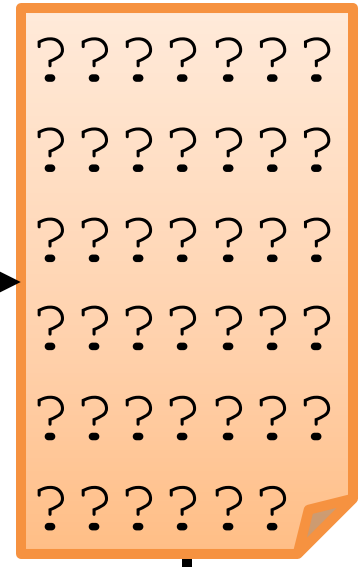


JWE 形式の JWT

JSON 形式で表現されたクレームの集合



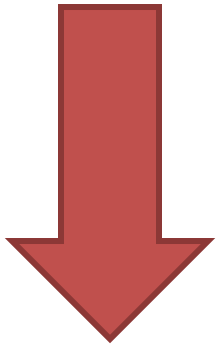
暗号化



BASE64URL エンコード

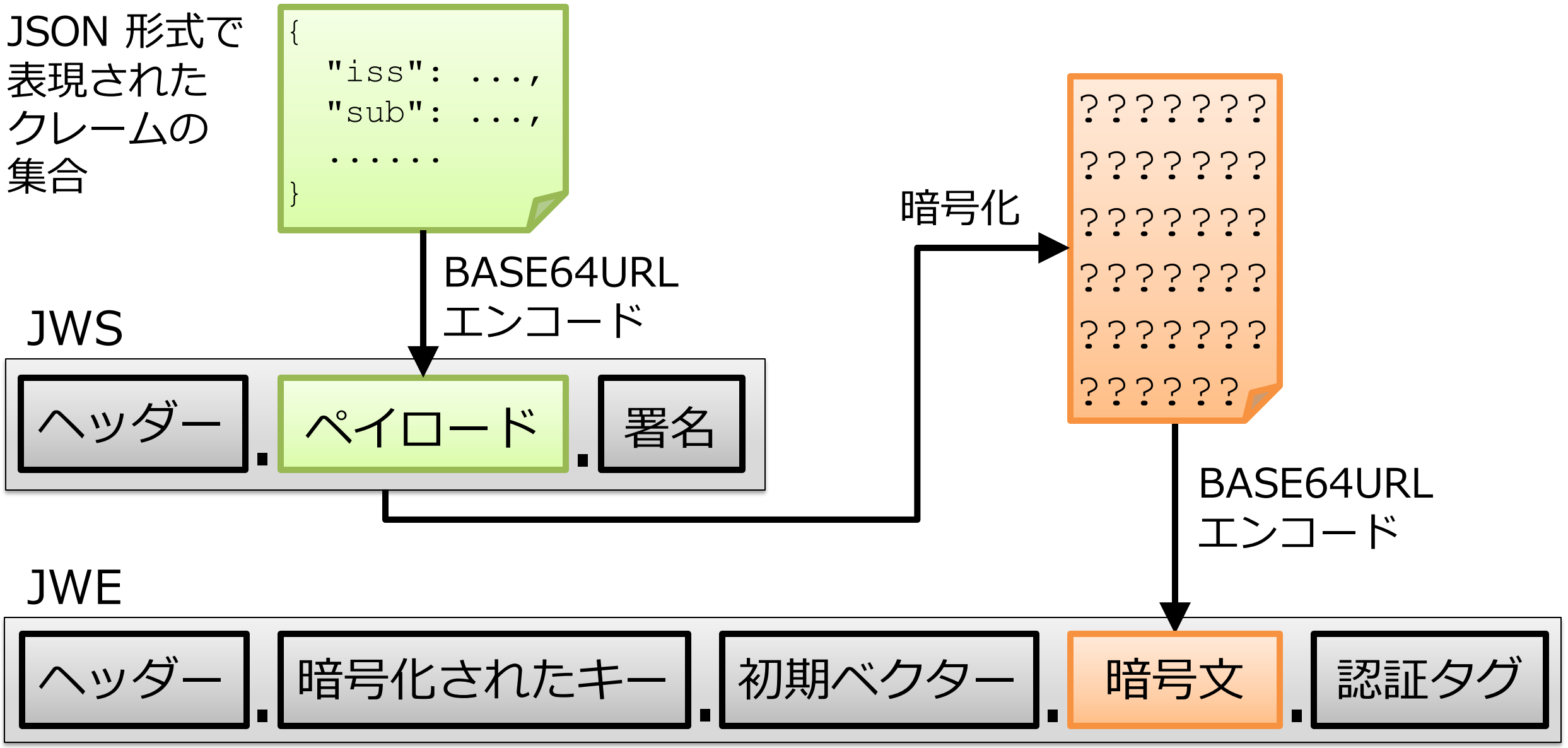
署名もしたいし、暗号化もしたい。

→ JWS を JWE でくるむ、もしくは
JWE を JWS でくるむ



Nested JWT

Nested JWT（JWS が JWE に含まれるパターン）



OpenID Connect Core 1.0, *“2. ID Token”*

> *ID Tokens MUST be signed using JWS and optionally both signed and then encrypted using JWS and JWE respectively, thereby providing authentication, integrity, non-repudiation, and optionally, confidentiality, per Section 16.14. If the ID Token is encrypted, it MUST be signed then encrypted, with the result being a Nested JWT, as defined in JWT.*

ID トークンには RFC 7519（JWT）より強い制限がかかる

署名	必須
暗号化	任意。但し暗号化の際は「署名後に暗号化」という順番が必須。

JWT クレーム

```
{
 "クレーム名": クレーム値,
 "クレーム名": クレーム値,
 ……
}
```

RFC 7519, *“3.1. Example JWT”*

```
{"iss":"joe",
 "exp":1300819380,
 "http://example.com/is_root":true}
```

RFC 7519, *“4.1. Registered Claim Names”*

クレーム名	意味	説明
iss	issuer	発行者
sub	subject	主体識別子
aud	audience	利用者
exp	expiration time	有効期限終了時刻
nbf	not before	有効期限開始時刻
iat	issued at	発行時刻
jti	JWT ID	JWT ID

ID Token


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

- ID トークンは JWT の一種
- ID トークンに含まれるクレーム

1. エンドユーザー認証に関するもの
2. エンドユーザー属性に関するもの
3. その他

OpenID Connect Core 1.0, *“2. ID Token”*

クレーム名	説明
iss	ID トークン発行者（OpenID プロバイダー識別子）
sub	認証されたユーザーの一意識別子
aud	想定する ID トークン利用者（クライアント ID）
exp	有効期限終了時刻
iat	ID トークン発行時刻
auth_time	ユーザーが認証された時刻
nonce	リプレイアタックを防ぐためのランダム文字列
acr	ユーザー認証が満した認証コンテキストクラス
amr	ユーザー認証方法
azp	承認された ID トークン発行対象

OpenID Connect Core 1.0, *“5.1. Standard Claims”* (1/2)

クレーム名	説明
sub	ユーザーの一意識別子
name	フルネーム
given_name	名
family_name	姓
middle_name	ミドルネーム
nickname	ニックネーム
preferred_username	好みのユーザー名
profile	プロフィールページの URL
picture	プロフィール画像の URL
website	Web サイトやブログの URL

OpenID Connect Core 1.0, *“5.1. Standard Claims”* (2/2)

クレーム名	説明
email	メールアドレス
email_verified	メールアドレスが検証済みか否か
gender	性別
birthdate	誕生日
zoneinfo	タイムゾーン
locale	ロケール
phone_number	電話番号
phone_number_verified	電話番号が検証済みか否か
address	住所
updated_at	情報の最終更新日

クレームによっては多言語対応が可能なものがある。

例）**family_name** → Kawasaki, 川崎, カワサキ

クレーム名末尾に「#言語タグ」をつけて区別する。

```
{"family_name":"Kawasaki",
 "family_name#ja-Hani-JP":"川崎",
 "family_name#ja-Kana-JP":"カワサキ"}
```

言語タグ → RFC 5646 (Tags for Identifying Languages)

OpenID Connect Core 1.0, 特殊なクレーム

クレーム名	説明
at_hash	ID トークンと同時に発行される アクセストークンのハッシュ値。
c_hash	ID トークンと同時に発行される 認可コードのハッシュ値。

Financial API, 特殊なクレーム

クレーム名	説明
s_hash	認可リクエストに含まれていた state パラメーターの値のハッシュ値。

OAuth 2.0 Flows

認可サーバーは２つのエンドポイントを提供する。

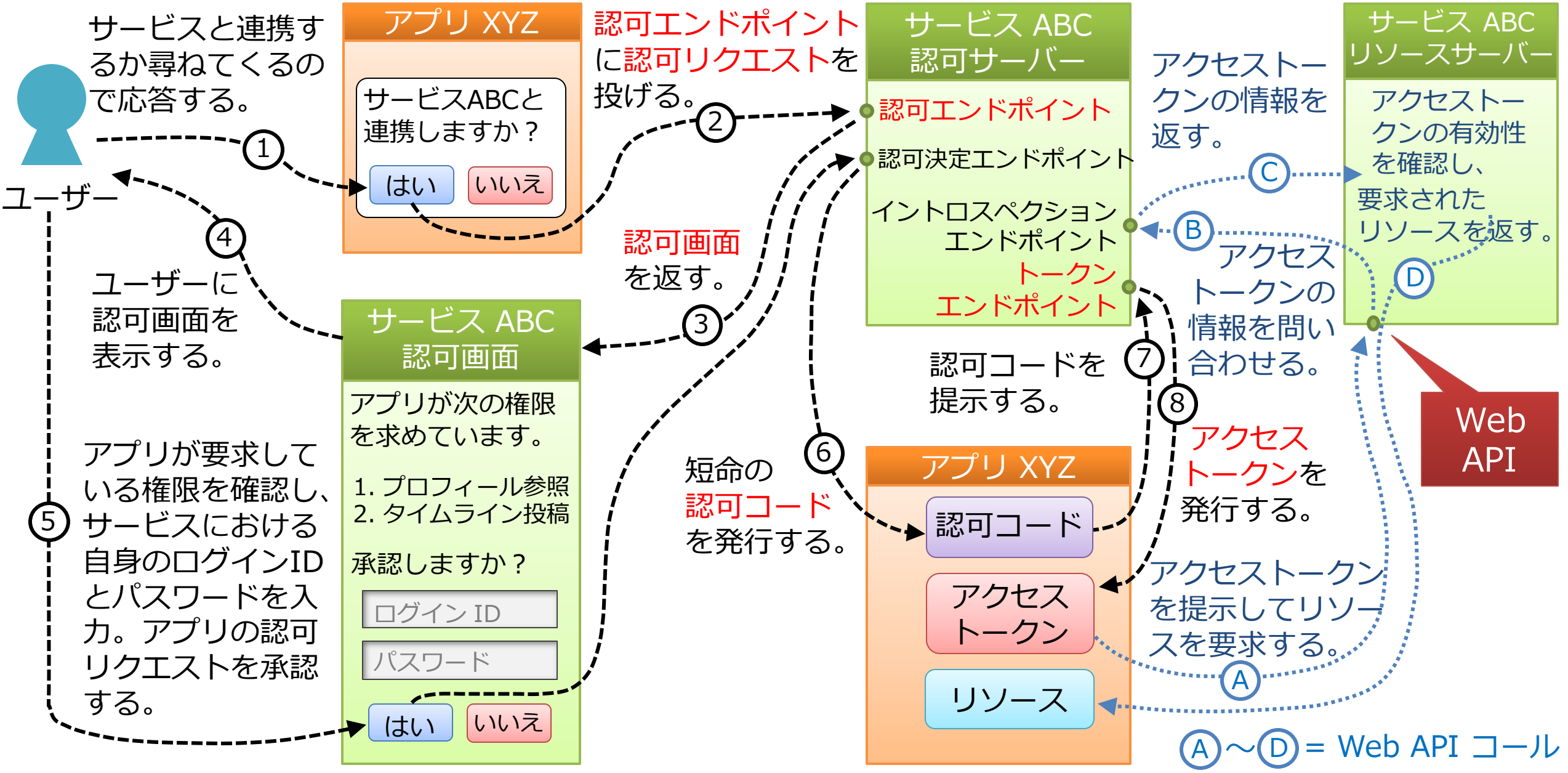
1	認可エンドポイント
2	トークンエンドポイント

RFC 6749（OAuth 2.0）は、これらのエンドポイントの動作を定義している。

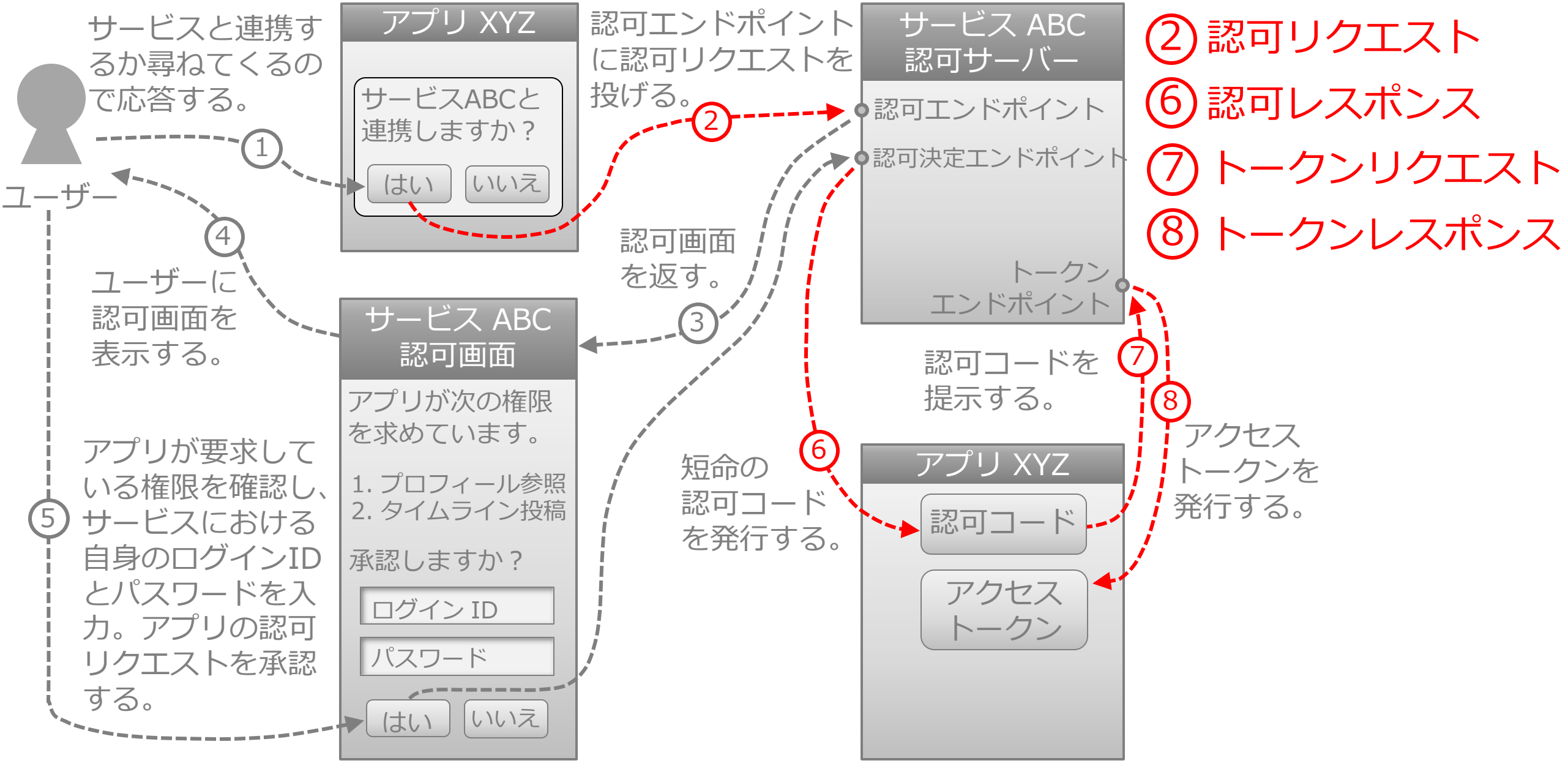
クライアントアプリケーションは、どちらか一方、もしくは両方のエンドポイントとやりとりして、アクセストークンを取得する。

フロー	認可 エンドポイント	トークン エンドポイント
認可コード	使う	使う
インプリシット	使う	使わない
リソースオーナー パスワード クレデンシャルズ	使わない	使う
クライアント クレデンシャルズ	使わない	使う
リフレッシュ トークン	使わない	使う

認可コードフロー (RFC 6749, 4.1)



認可コードフロー (RFC 6749, 4.1)



認可コードフロー (RFC 6749, 4.1)

認可リクエスト

```
GET {認可エンドポイント}
   ?response_type=code
   &client_id={クライアントID}
   &redirect_uri={リダイレクトURI}
   &scope={スコープ群}
   &state={任意文字列}
   HTTP/1.1
HOST: {認可サーバー}
```

認可レスポンス

```
HTTP/1.1 302 Found
Location: {リダイレクトURI}
   ?code={認可コード}
   &state={任意文字列}
```

- response_type=code で認可コードフローを指定。
- code の値が発行された認可コード。

認可レスポンスパラメーター群の受け取り方（認可コードフロー）

『OAuth 2.0 の認可レスポンスとリダイレクトに関する説明』より抜粋
https://qiita.com/TakahikoKawasaki/items/8567c80528da43c7e844

認可コードフロー (RFC 6749, 4.1)

トークンリクエスト

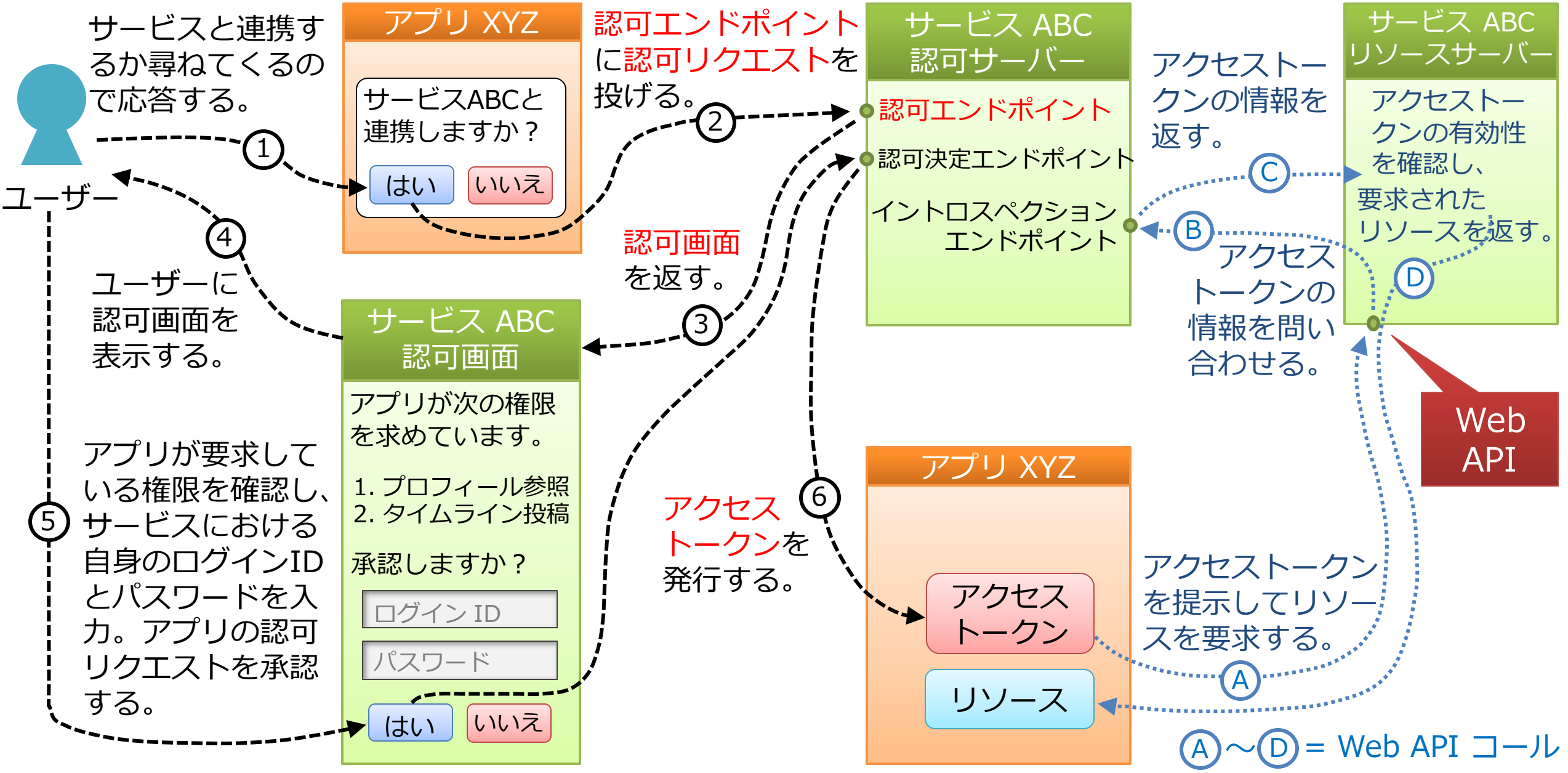
```
POST {トークンエンドポイント} HTTP/1.1
HOST: {認可サーバー}
Content-Type:
  application/x-www-form-urlencoded

grant_type=authorization_code&
code={認可コード}&
client_id={クライアントID}&
redirect_uri={リダイレクトURI}
```

トークンレスポンス

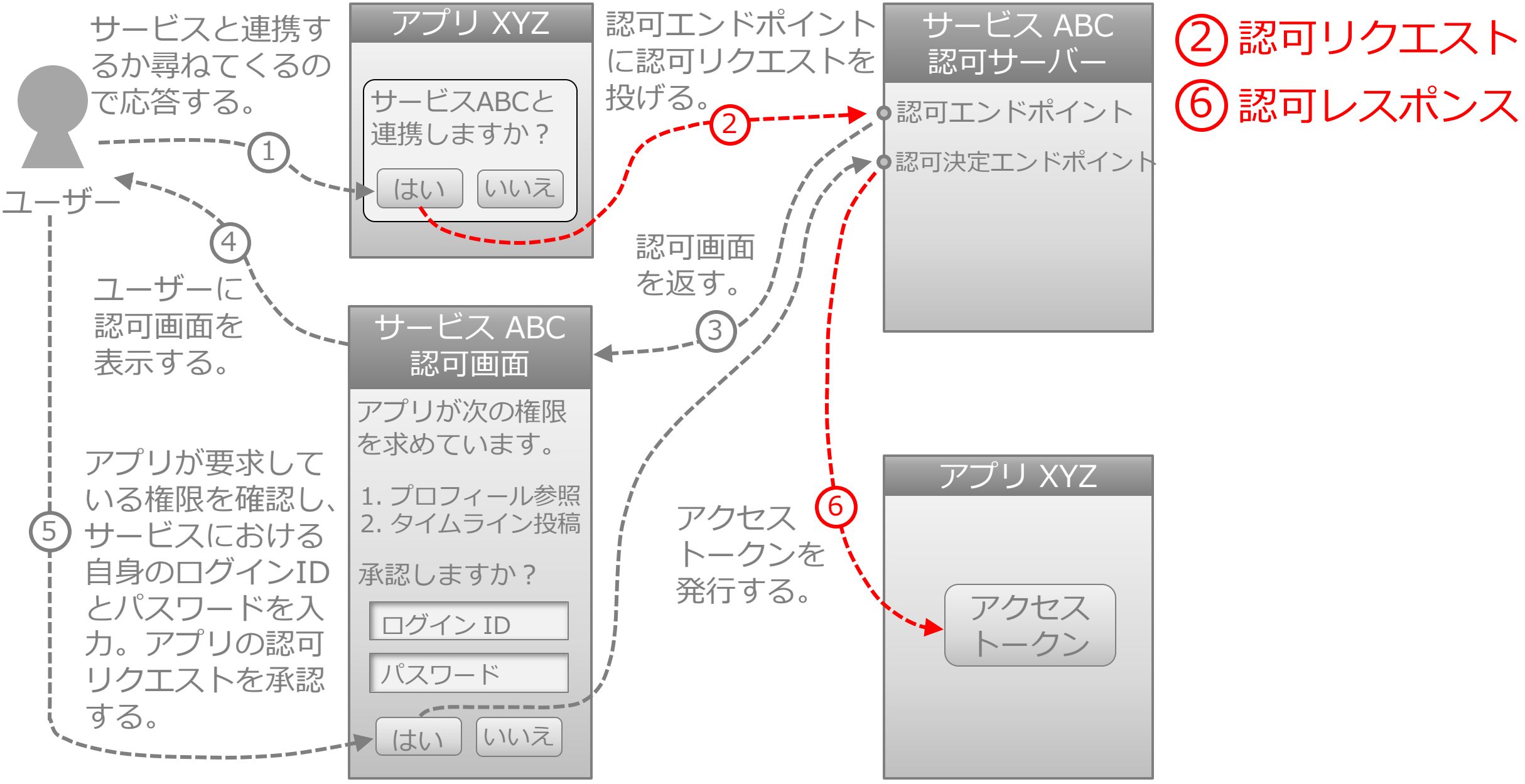
```
HTTP/1.1 200 OK
Content-Type:
  application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"{アクセストークン}",
  "token_type":"{トークンタイプ}",
  "expires_in":{有効秒数},
  "refresh_token":"{リフレッシュトークン}",
  "scope":"{スコープ群}"
}
```

- grant_type=authorization_code で認可コードフローを指定。
- 認可エンドポイントで発行された認可コードを code で指定。
- access_token の値が発行されたアクセストークン。

インプリシットフロー (RFC 6749, 4.2)



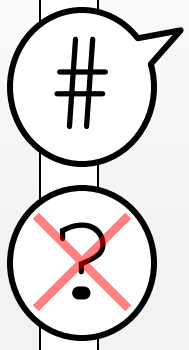
インプリシットフロー (RFC 6749, 4.2)



インプリシットフロー (RFC 6749, 4.2)

認可リクエスト

```
GET {認可エンドポイント}
   ?response_type=token
   &client_id={クライアントID}
   &redirect_uri={リダイレクトURI}
   &scope={スコープ群}
   &state={任意文字列}
   HTTP/1.1
HOST: {認可サーバー}
```



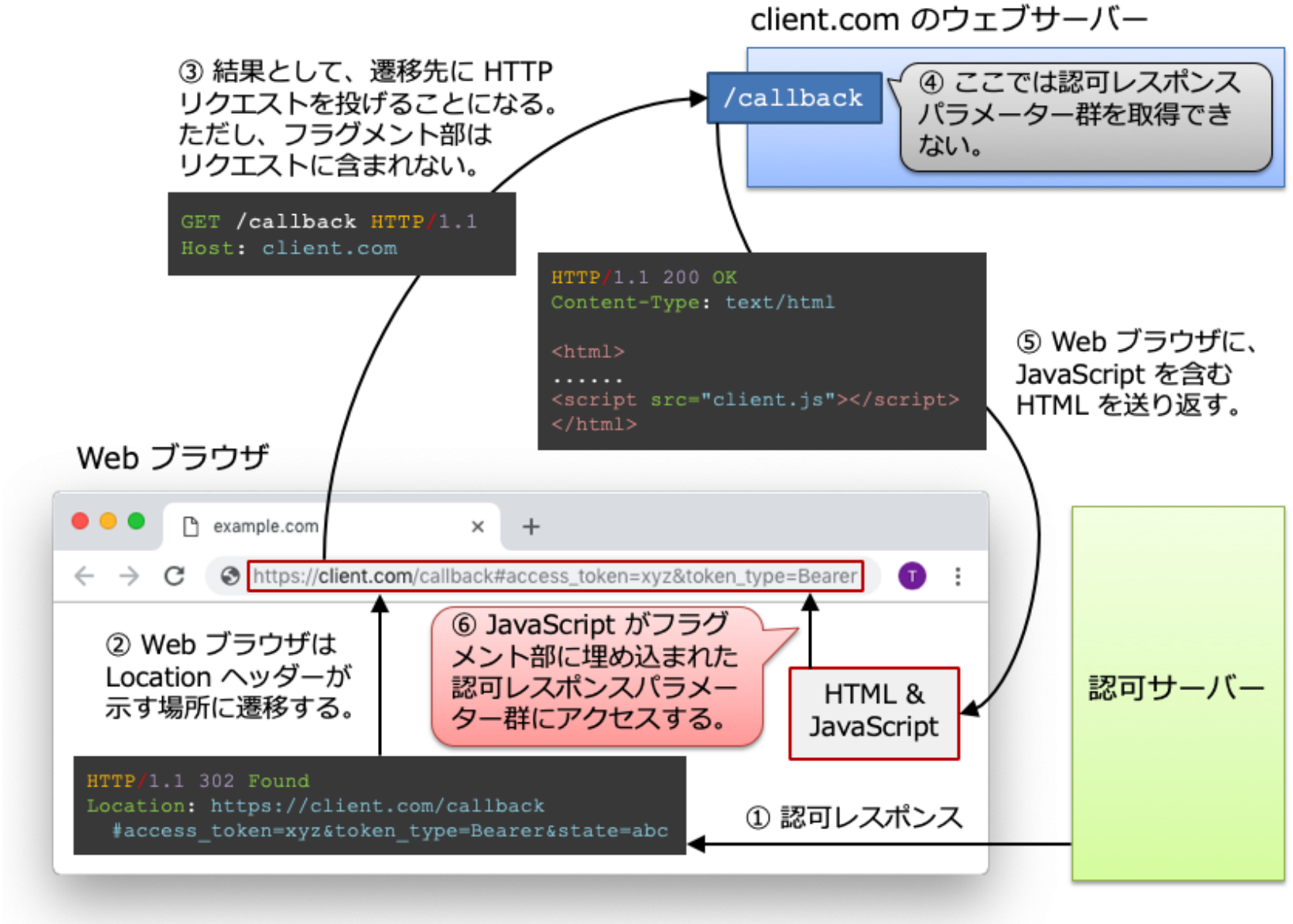
認可レスポンス

```
HTTP/1.1 302 Found
Location: {リダイレクトURI}
   #access_token={アクセストークン}
   &token_type={トークンタイプ}
   &expires_in={有効秒数}
   &scope={スコープ群}
   &state={任意文字列}
```

- response_type=token でインプリシットフローを指定。
- access_token の値が発行されたアクセストークン。
- レスポンスパラメーター群はフラグメント部に置かれる。
- フラグメント部のパラメーター群はサーバーに送られない。それらを参照できるのは、Web ブラウザやカスタムスキームハンドラーだけ。

クエリー部ではない。

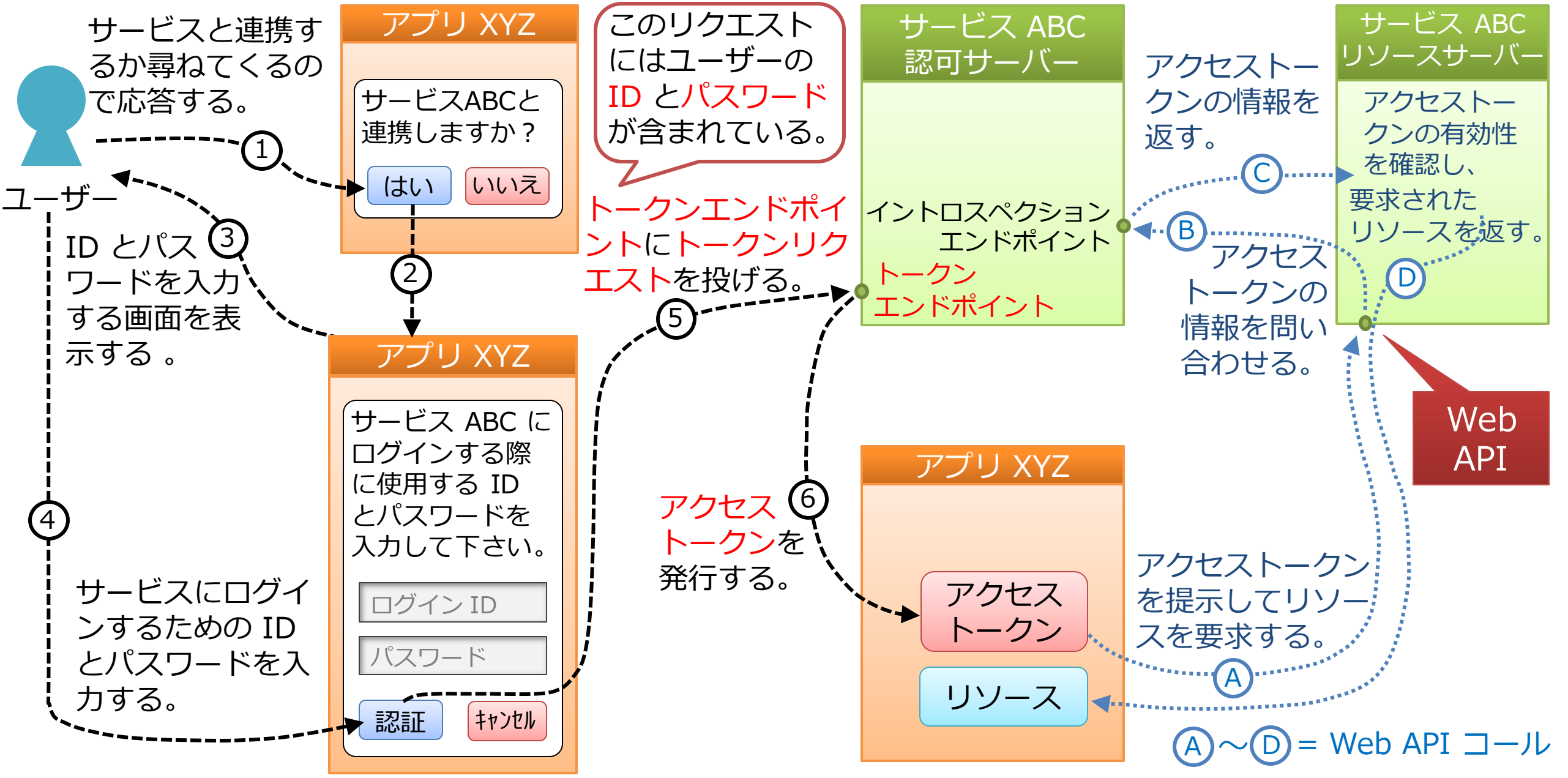
認可レスポンスパラメーター群の受け取り方（インプリシットフロー）



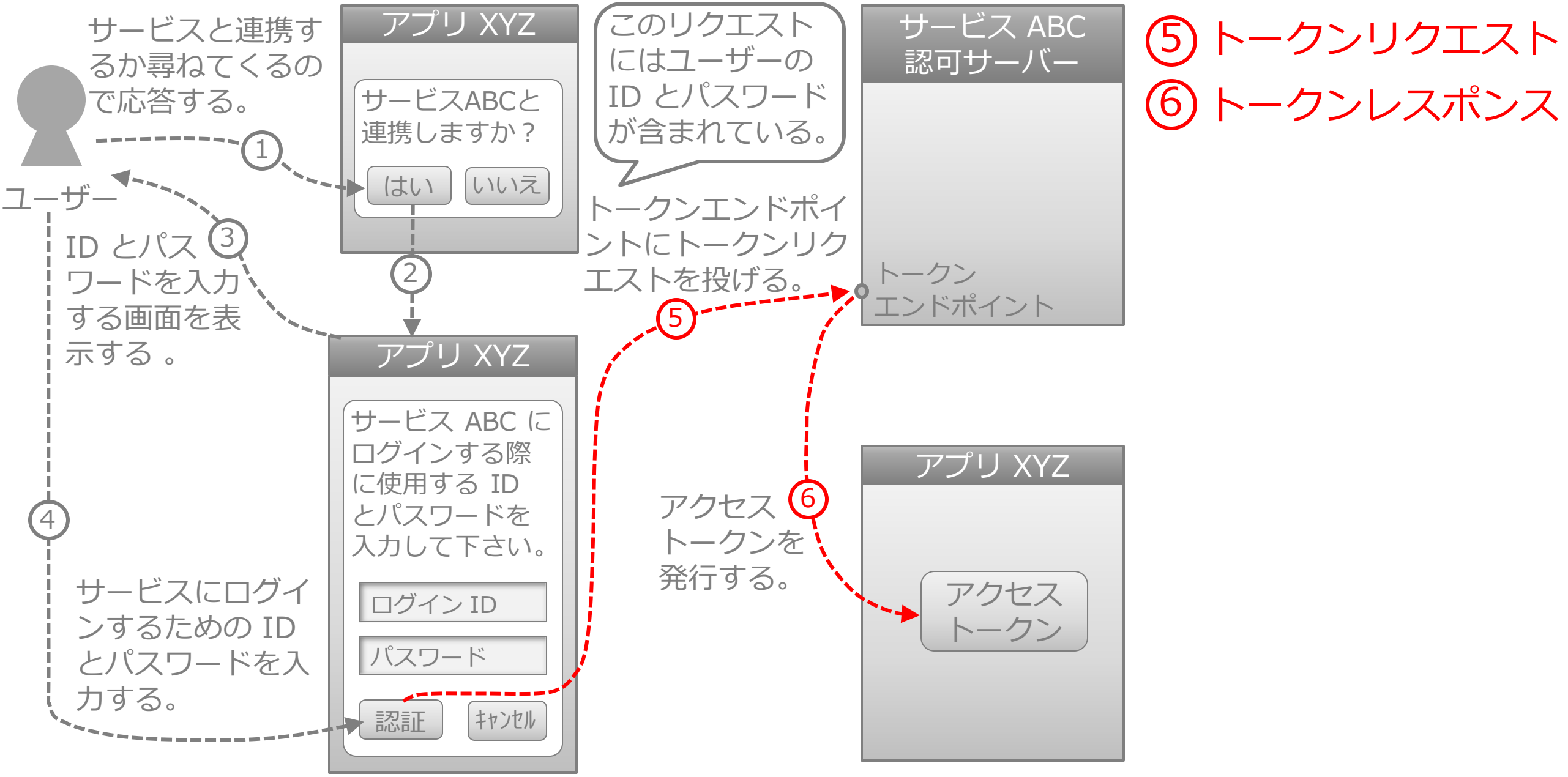
『OAuth 2.0 の認可レスポンスとリダイレクトに関する説明』より抜粋

https://qiita.com/TakahikoKawasaki/items/8567c80528da43c7e844

リソースオーナー・パスワード・クレデンシャルズフロー (RFC 6749, 4.3)



リソースオーナー・パスワード・クレデンシャルズフロー (RFC 6749, 4.3)



リソースオーナー・パスワード・クレデンシャルズフロー (RFC 6749, 4.3)

トークンリクエスト

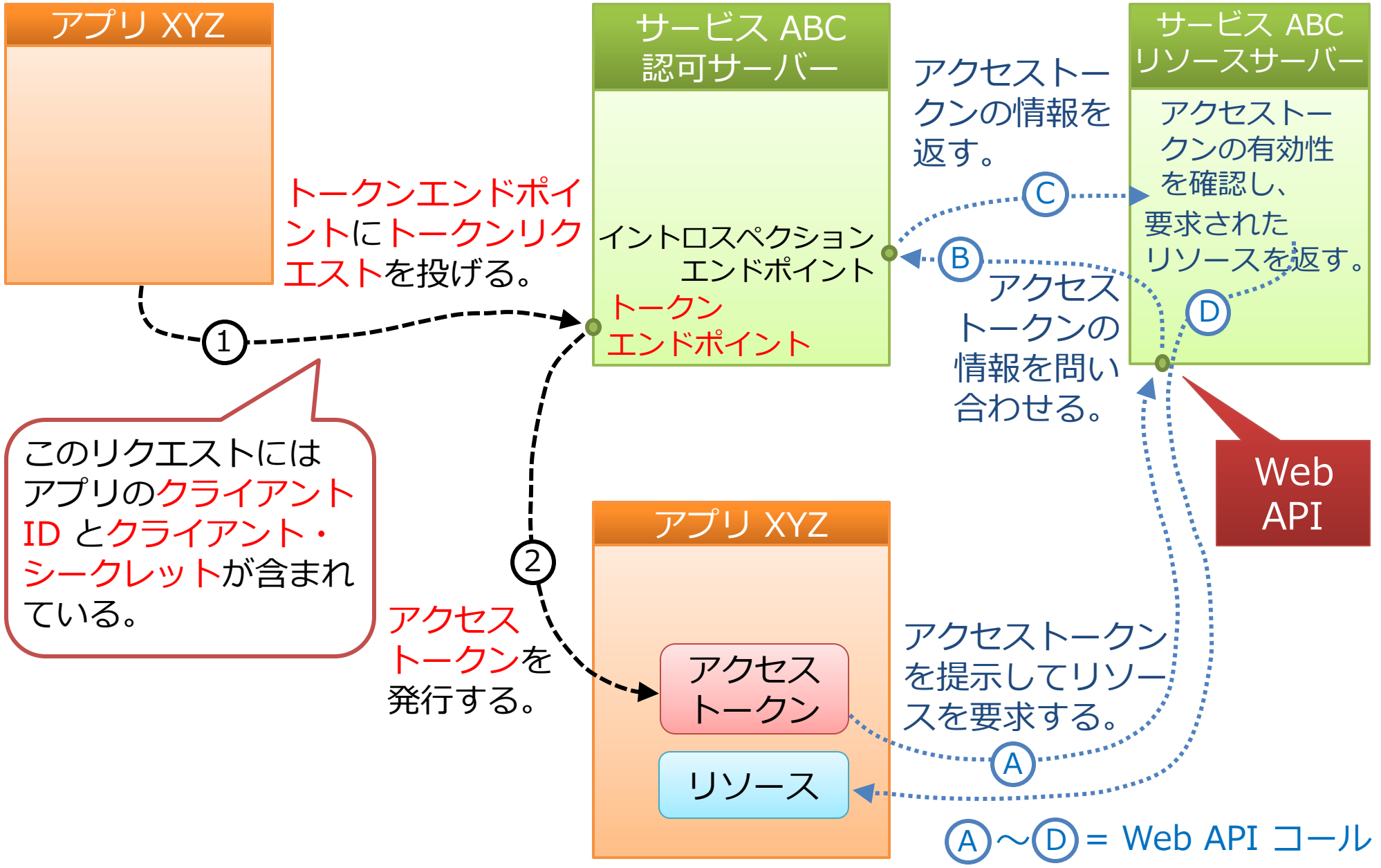
```
POST {トークンエンドポイント} HTTP/1.1
HOST: {認可サーバー}
Content-Type:
  application/x-www-form-urlencoded

grant_type=password&
username={ユーザーID}&
password={パスワード}&
client_id={クライアントID}&
scope={スコープ群}
```

トークンレスポンス

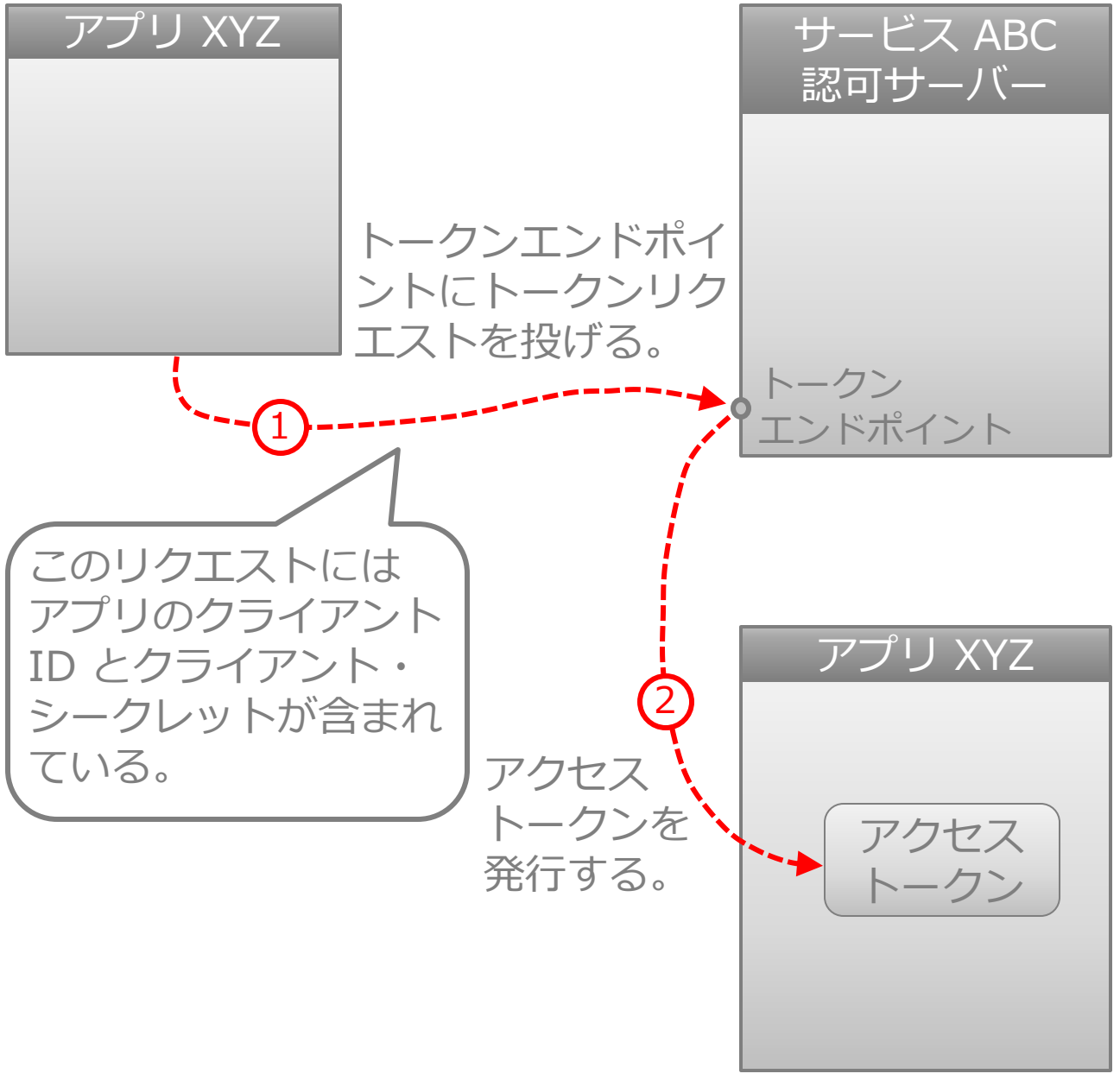
```
HTTP/1.1 200 OK
Content-Type:
  application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"{アクセストークン}",
  "token_type":"{トークンタイプ}",
  "expires_in":{有効秒数},
  "refresh_token":"{リフレッシュトークン}",
  "scope":"{スコープ群}"
}
```

- grant_type=password で
  リソースオーナー・パスワード・クレデンシャルズフローを指定。
- username と password でユーザーの ID とパスワードを提示。
- access_token の値が発行されたアクセストークン。

クライアント・クレデンシャルズフロー (RFC 6749, 4.4)



クライアント・クレデンシャルズフロー (RFC 6749, 4.4)



① トークンリクエスト

② トークンレスポンス

クライアント・クレデンシャルズフロー (RFC 6749, 4.4)

トークンリクエスト

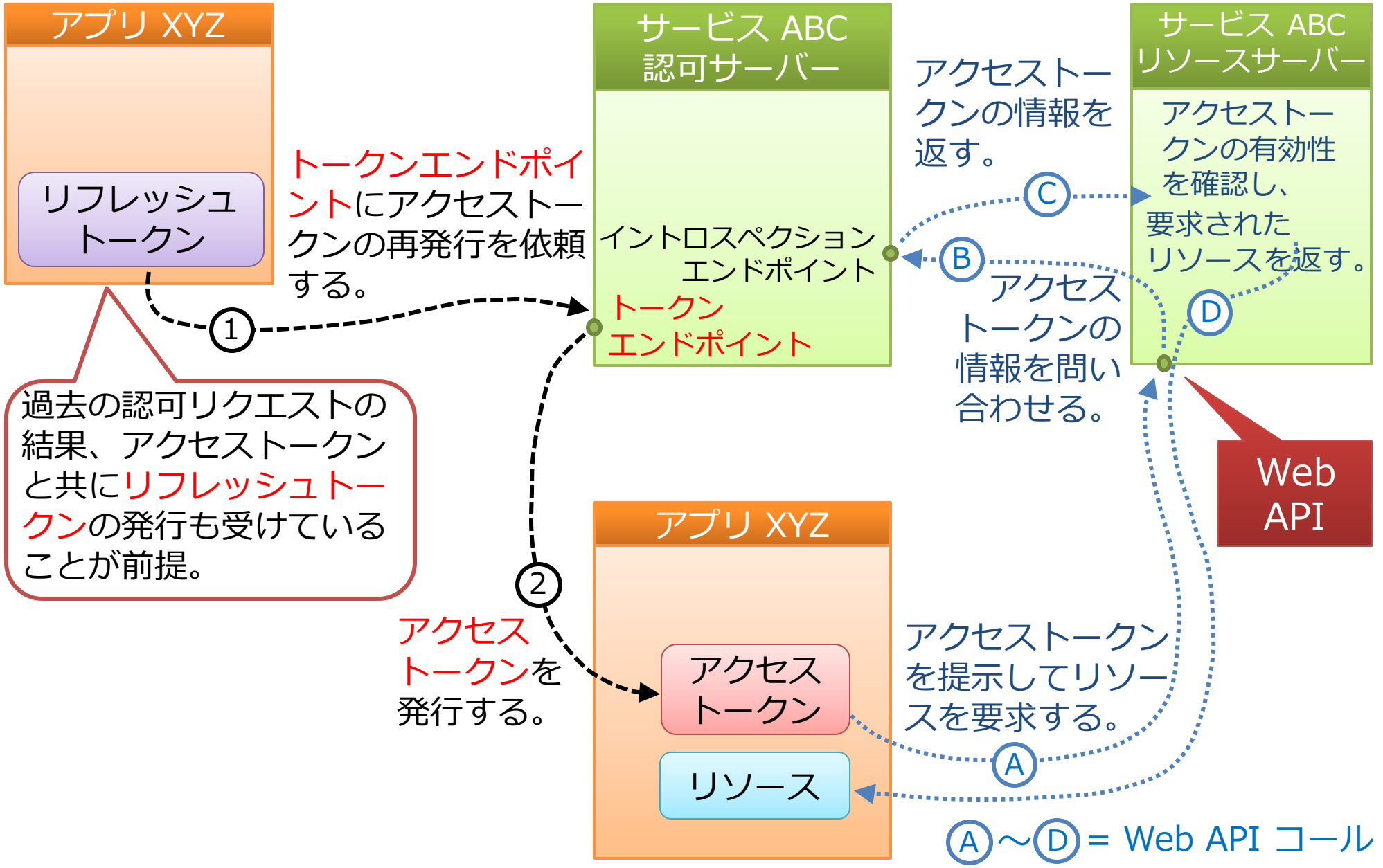
```
POST {トークンエンドポイント} HTTP/1.1
HOST: {認可サーバー}
Authorization:
  Basic {クライアントクレデンシャルズ}
Content-Type:
  application/x-www-form-urlencoded

grant_type=client_credentials&
scope={スコープ群}
```

トークンレスポンス

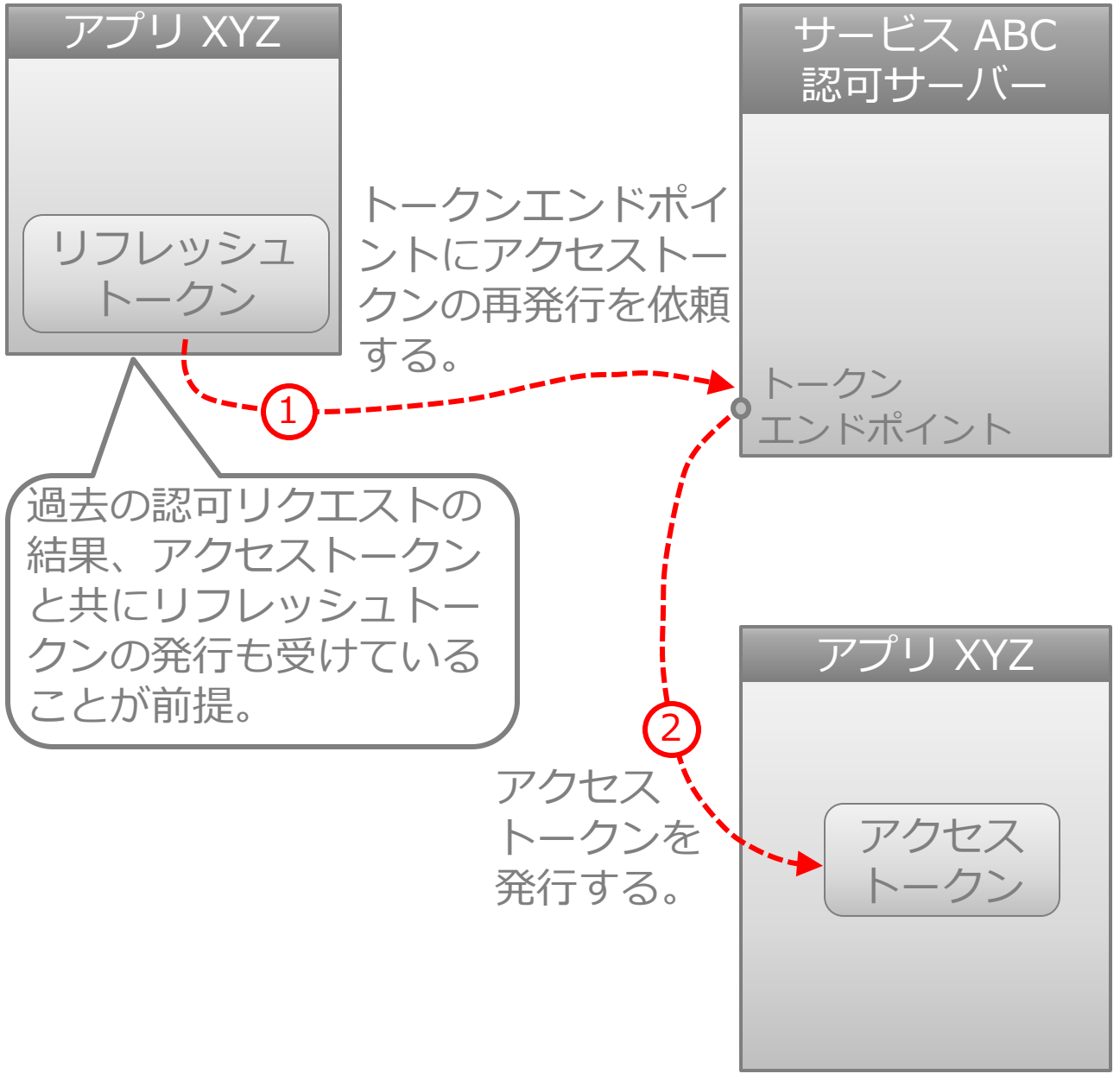
```
HTTP/1.1 200 OK
Content-Type:
  application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"{アクセストークン}",
  "token_type":"{トークンタイプ}",
  "expires_in":{有効秒数},
  "scope":"{スコープ群}"
}
```

- grant_type=client_credentials で クライアント・クレデンシャルズフローを指定。
- リクエストにクライアントのクレデンシャルズを含める。
- access_token の値が発行されたアクセストークン。

リフレッシュ・トークンフロー (RFC 6749, 6)



リフレッシュ・トークンフロー (RFC 6749, 6)



① トークンリクエスト

② トークンレスポンス

リフレッシュトークンフロー (RFC 6749, 6)

トークンリクエスト

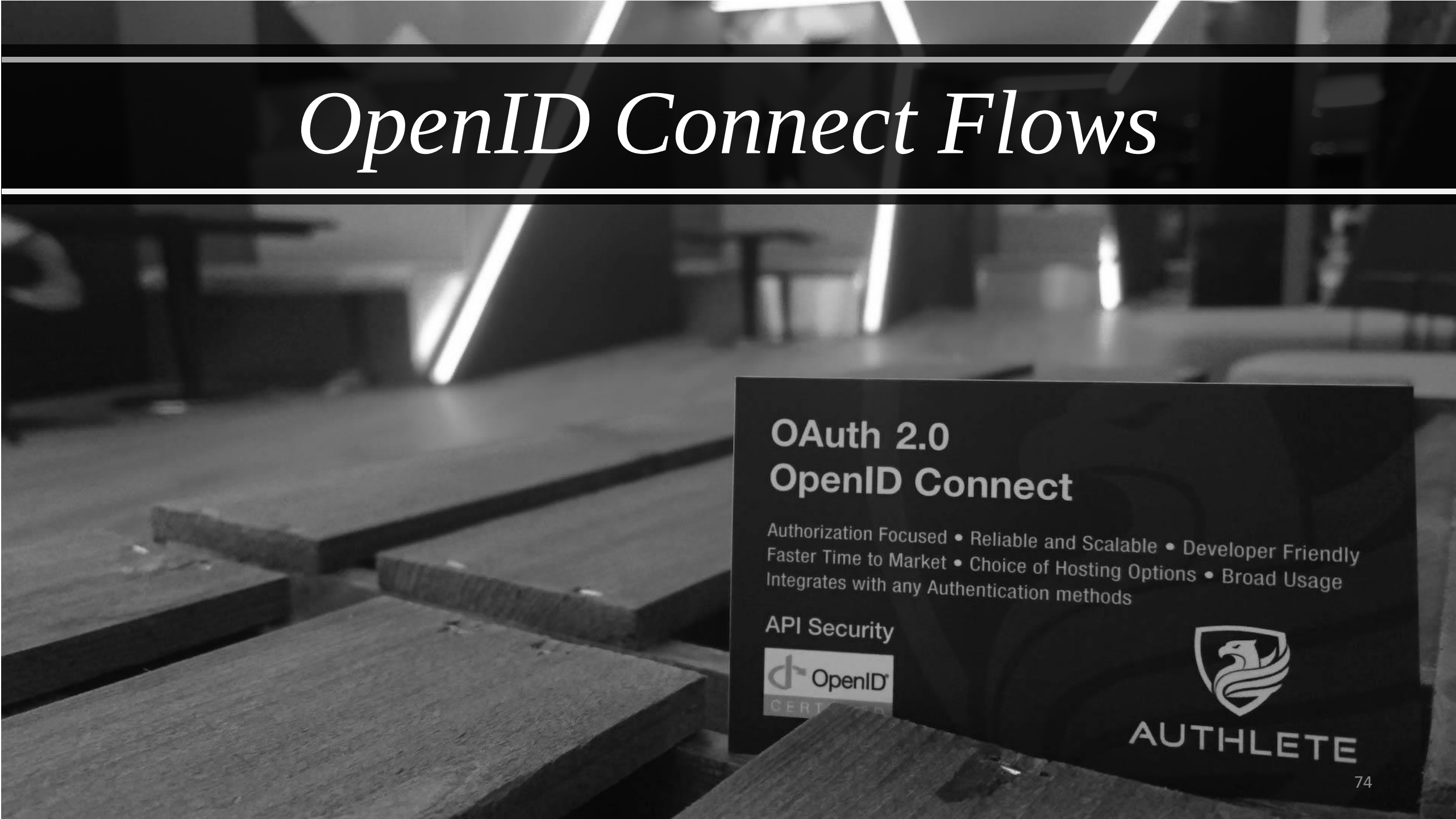
```
POST {トークンエンドポイント} HTTP/1.1
HOST: {認可サーバー}
Content-Type:
  application/x-www-form-urlencoded

grant_type=refresh_token&
refresh_token={リフレッシュトークン}&
client_id={クライアントID}&
scope={スコープ群}
```

トークンレスポンス

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

{
  "access_token":"{アクセストークン}",
  "token_type":"{トークンタイプ}",
  "expires_in":{有効秒数},
  "refresh_token":"{リフレッシュトークン}",
  "scope":"{スコープ群}"
}
```

- grant_type=refresh_token でリフレッシュトークンフローを指定。
- リフレッシュトークンを refresh_token で指定。
- access_token の値が発行されたアクセストークン。

OpenID Connect Flows



OAuth 2.0 では、認可エンドポイントを使うフローは次の二つ。

認可コードフロー

の認可リクエスト

```
GET {認可エンドポイント}
  ?response_type=code
  &client_id={クライアントID}
  &redirect_uri={リダイレクトURI}
  &scope={スコープ群}
  &state={任意文字列}
  HTTP/1.1
HOST: {認可サーバー}
```

インプリシットフロー

の認可リクエスト

```
GET {認可エンドポイント}
  ?response_type=token
  &client_id={クライアントID}
  &redirect_uri={リダイレクトURI}
  &scope={スコープ群}
  &state={任意文字列}
  HTTP/1.1
HOST: {認可サーバー}
```

response_type の値が code ならば認可コードフロー、token ならばインプリシットフロー。

OpenID Connect は response_type の仕様を拡張。

1	id_token を追加
2	none を追加
3	code, token, id_token の任意の組合せを許可

加えて、scope パラメーターが含みうる値として openid を定義。

response_type の値と、scope に openid を含むかどうかにより、フロー及び、認可サーバー（OpenID プロバイダー）から返されるトークンの種類が決まる。

OpenID Connect Core 1.0, 3.1.2.1. Authentication Request

> *scope*
>
> *REQUIRED. OpenID Connect requests MUST contain the openid scope value. If the openid scope value is not present, the behavior is entirely unspecified.*

"the behavior is entirely unspecified" の私の解釈は「正しく動かない」

しかし世の中には「正しく動くケースも含む」と考える人もいる。

→ https://www.ietf.org/mail-archive/web/oauth/current/msg17364.html

しかし、正しく動いてよいなら、「MUST contain」の意味がなくなる。

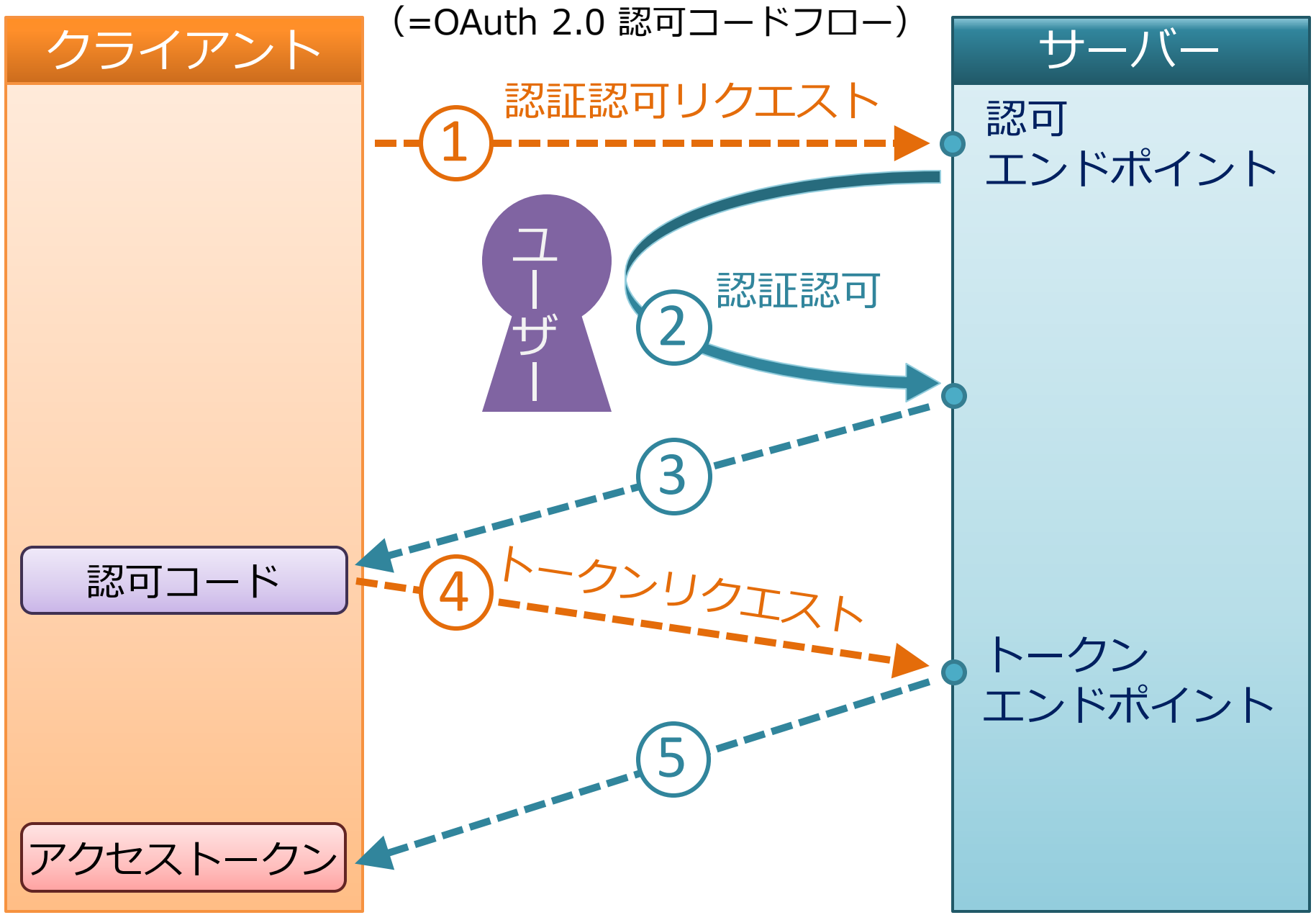
（scope に openid が含まれるかどうかをチェックする必要がなくなってしまう。）

Authlete の実装：

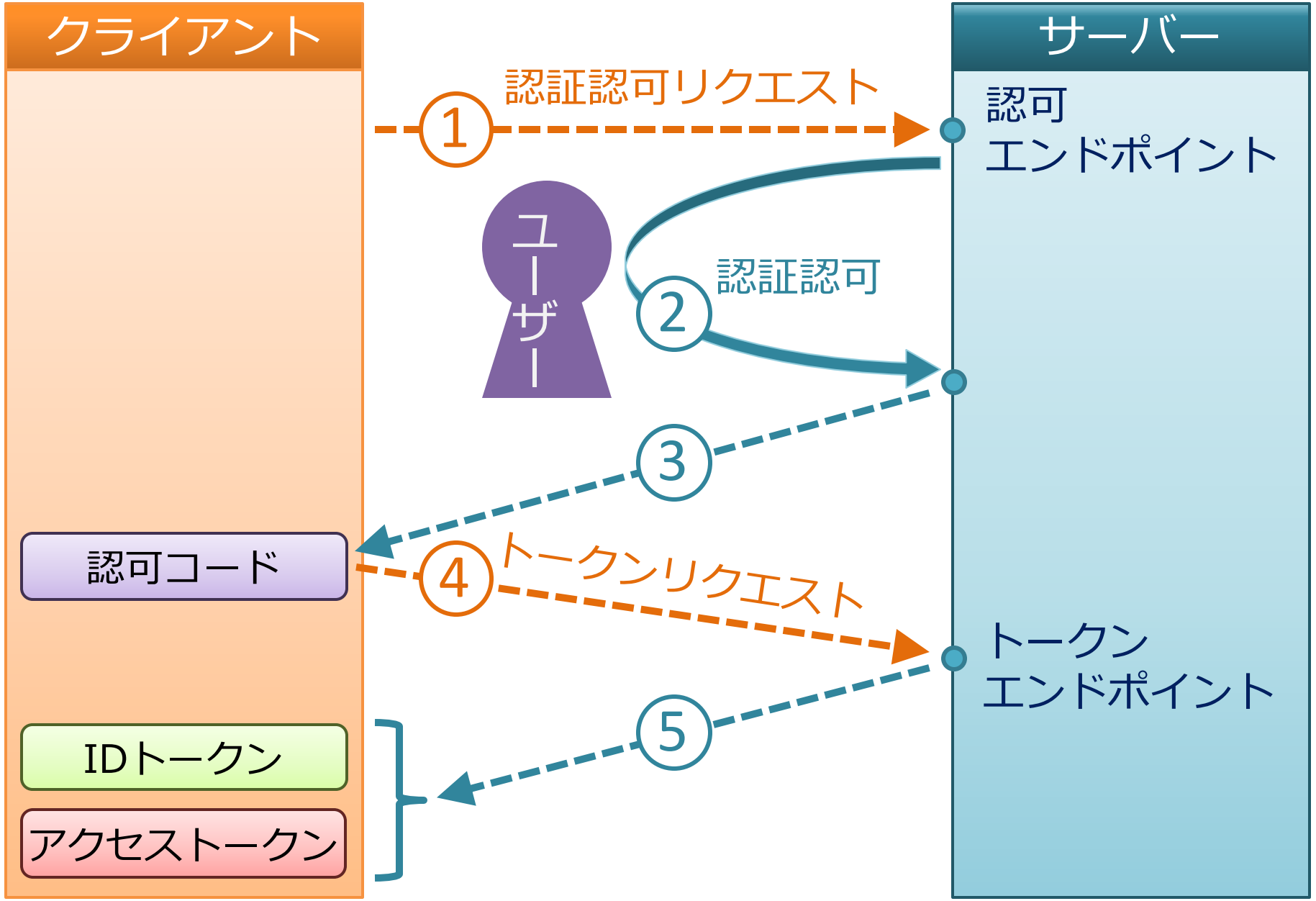
1. response_type が id_token を含む場合、scope に openid が含まれていなければ必ずエラーとする。
2. response_type が "code", "token", "code token", "none" の場合、scope に openid が含まれていなくてもエラーにならない。

	response_type				scope
	code	id_token	token	none	openid
1a	✔				
1b	✔				✔
2			✔		
3		✔			✔
4		✔	✔		✔
5	✔	✔			
6a	✔		✔		
6b	✔		✔		✔
7	✔	✔	✔		✔
8				✔	

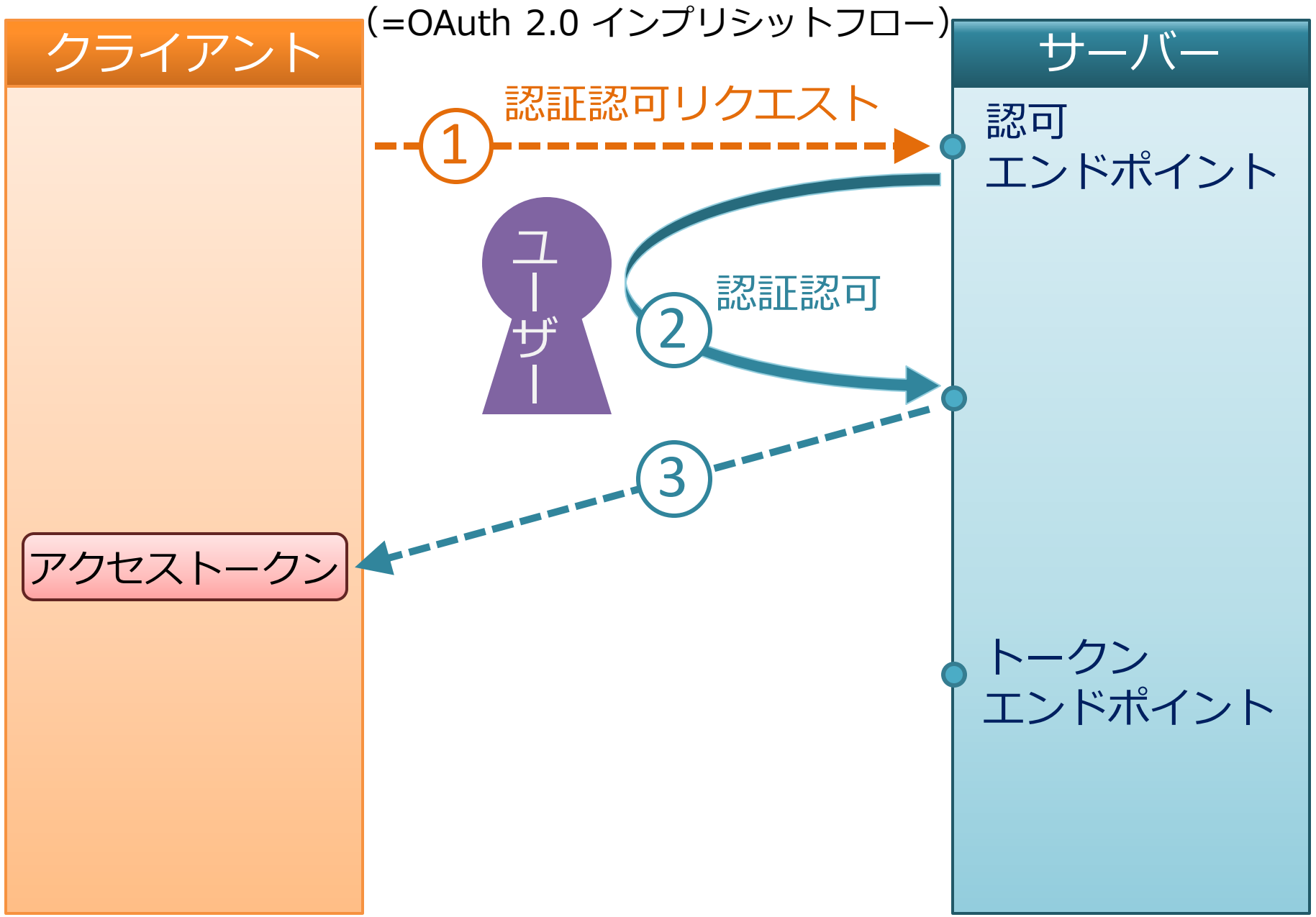
1a. response_type=code (openid を含まない)



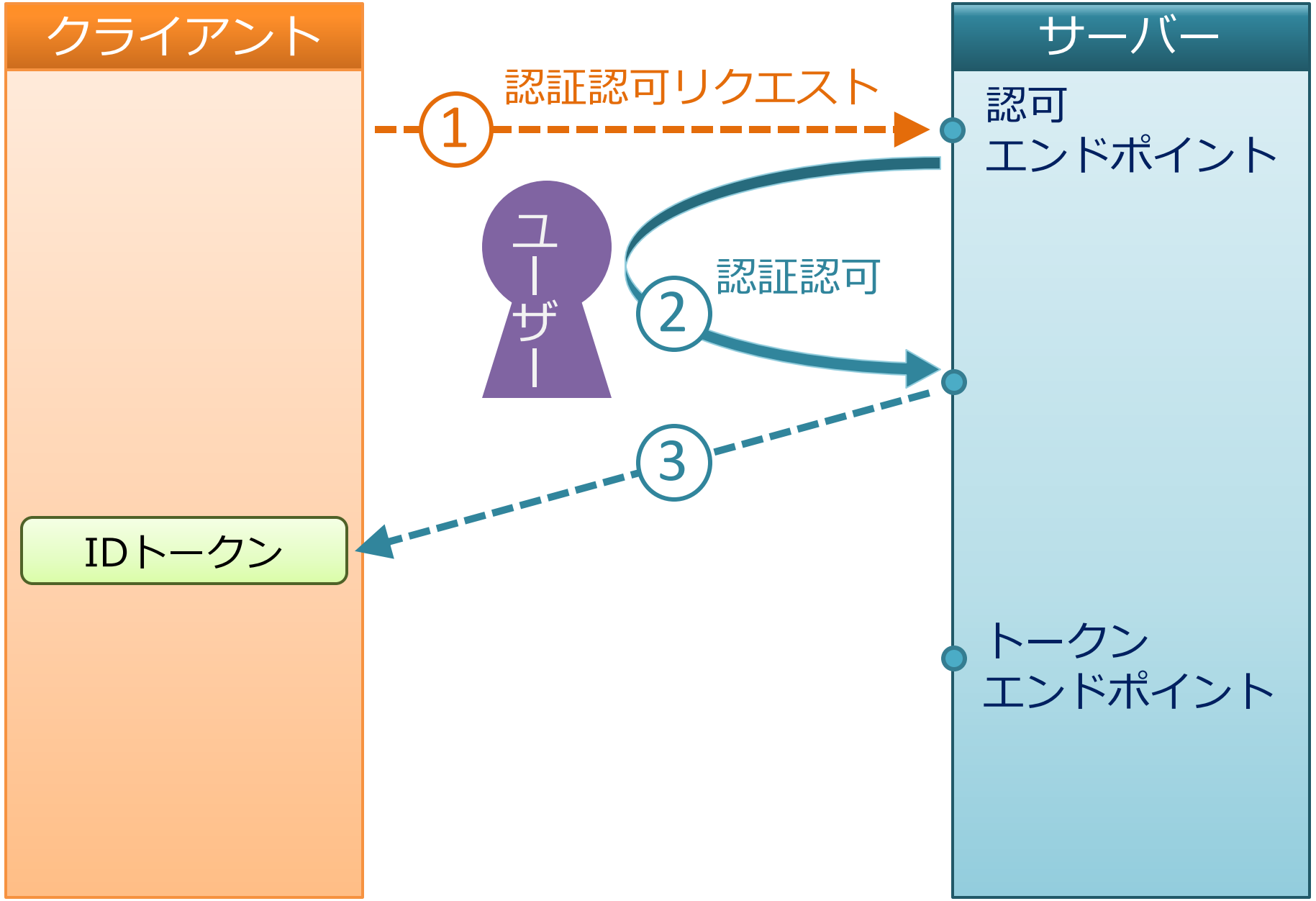
1b. response_type=code (openid を含む)



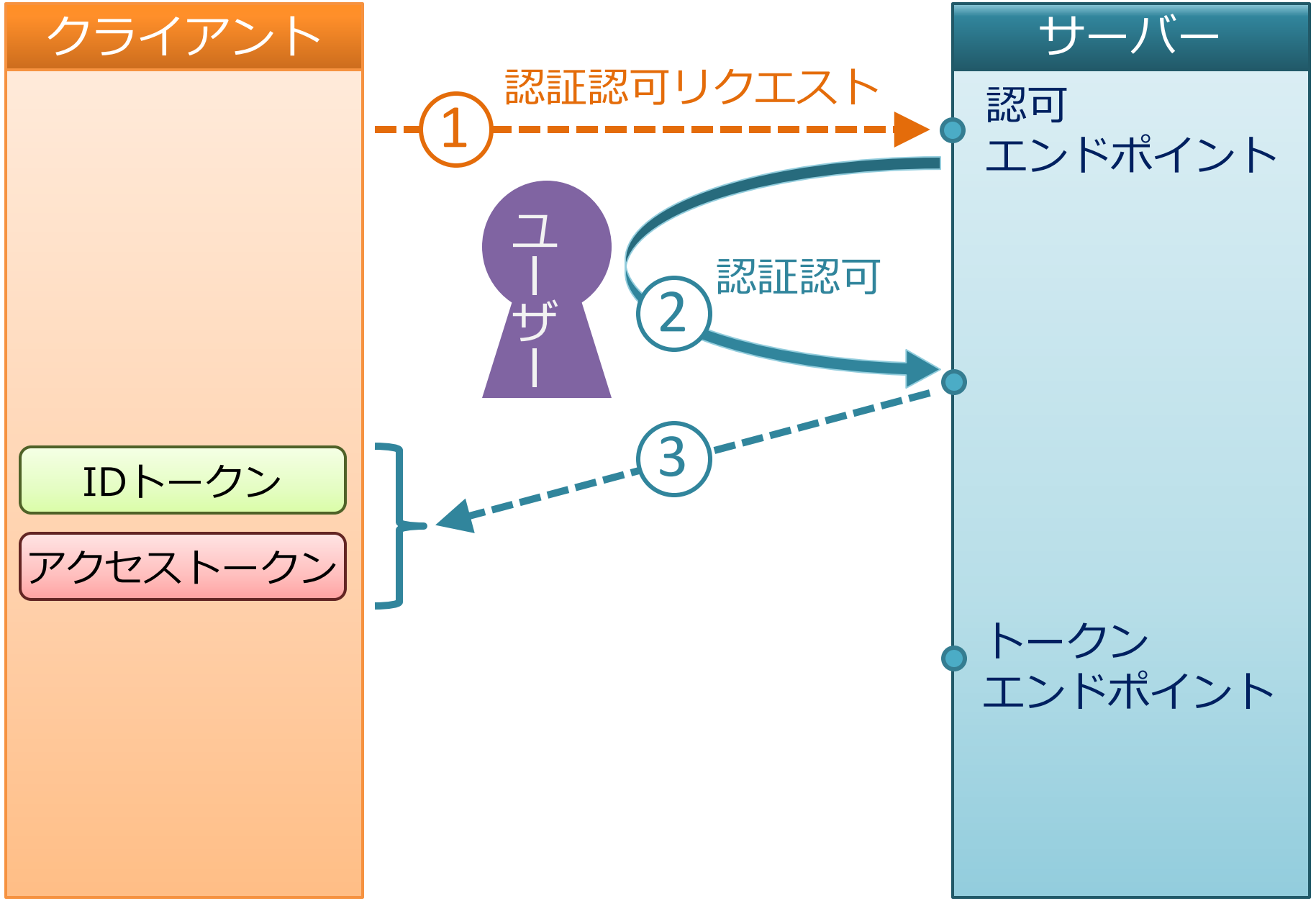
2. response_type=token (openid 不要)



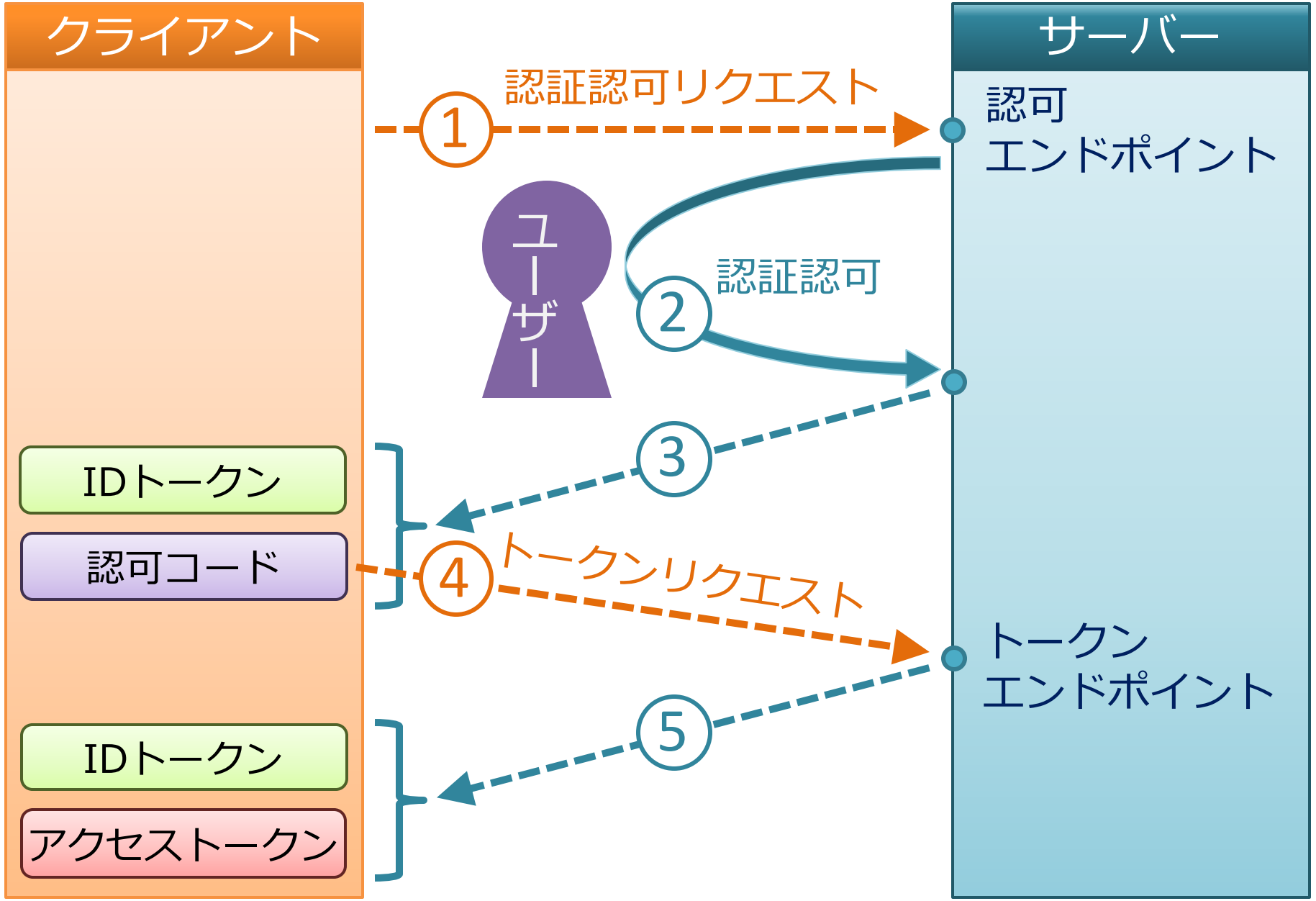
3. response_type=id_token (openid 必須)



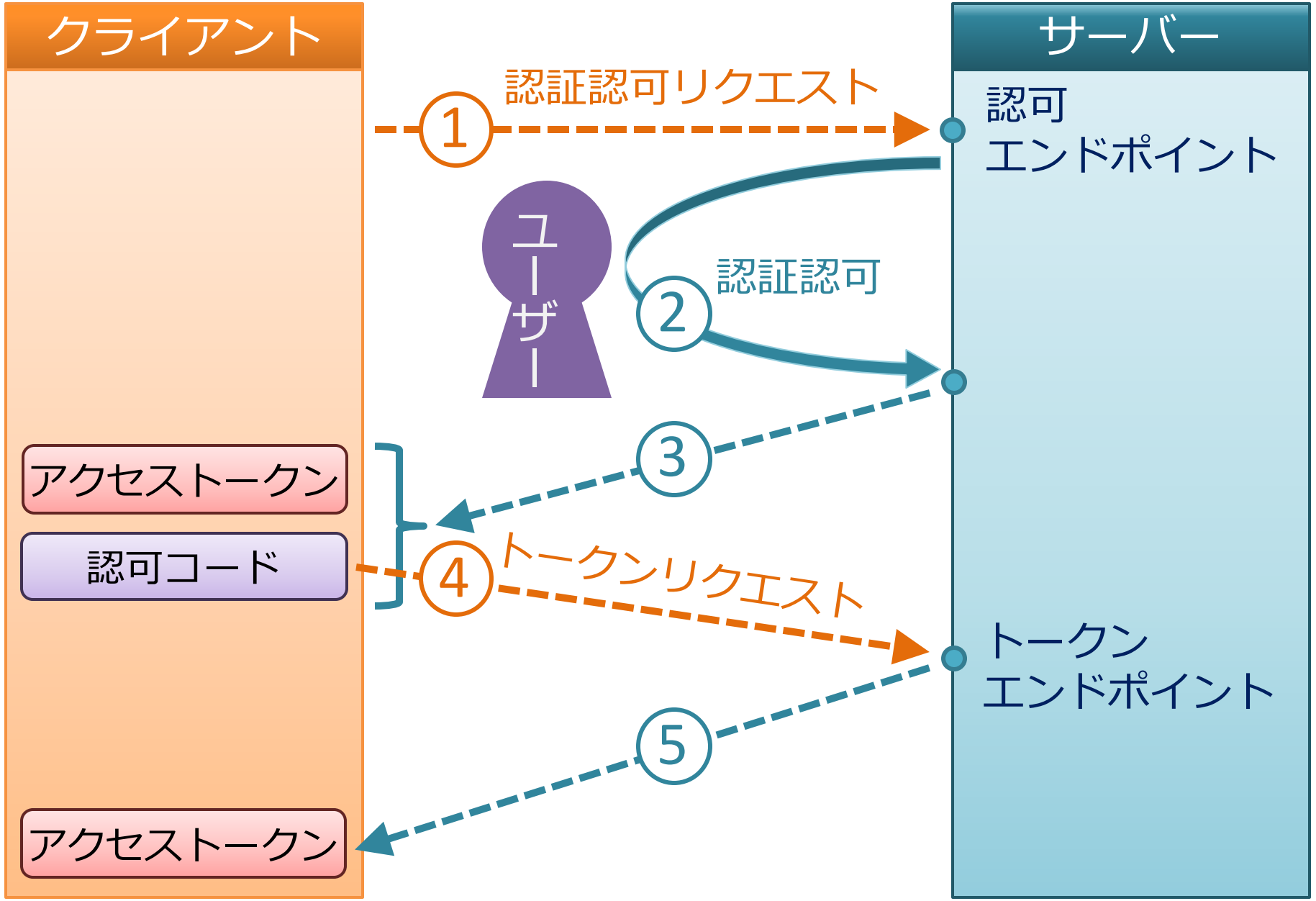
4. response_type=id_token token (openid 必須)



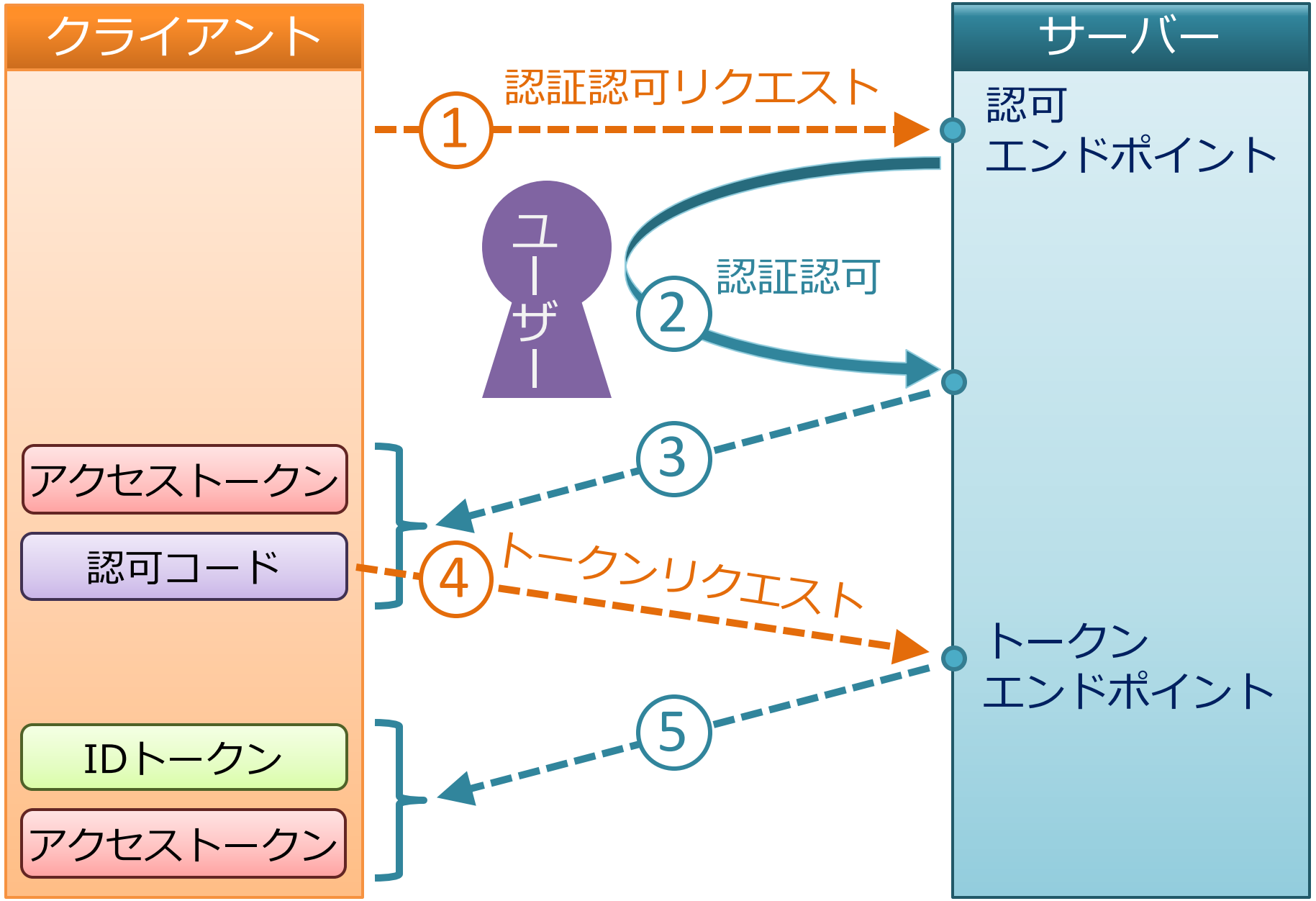
5. response_type=code id_token (openid 必須)



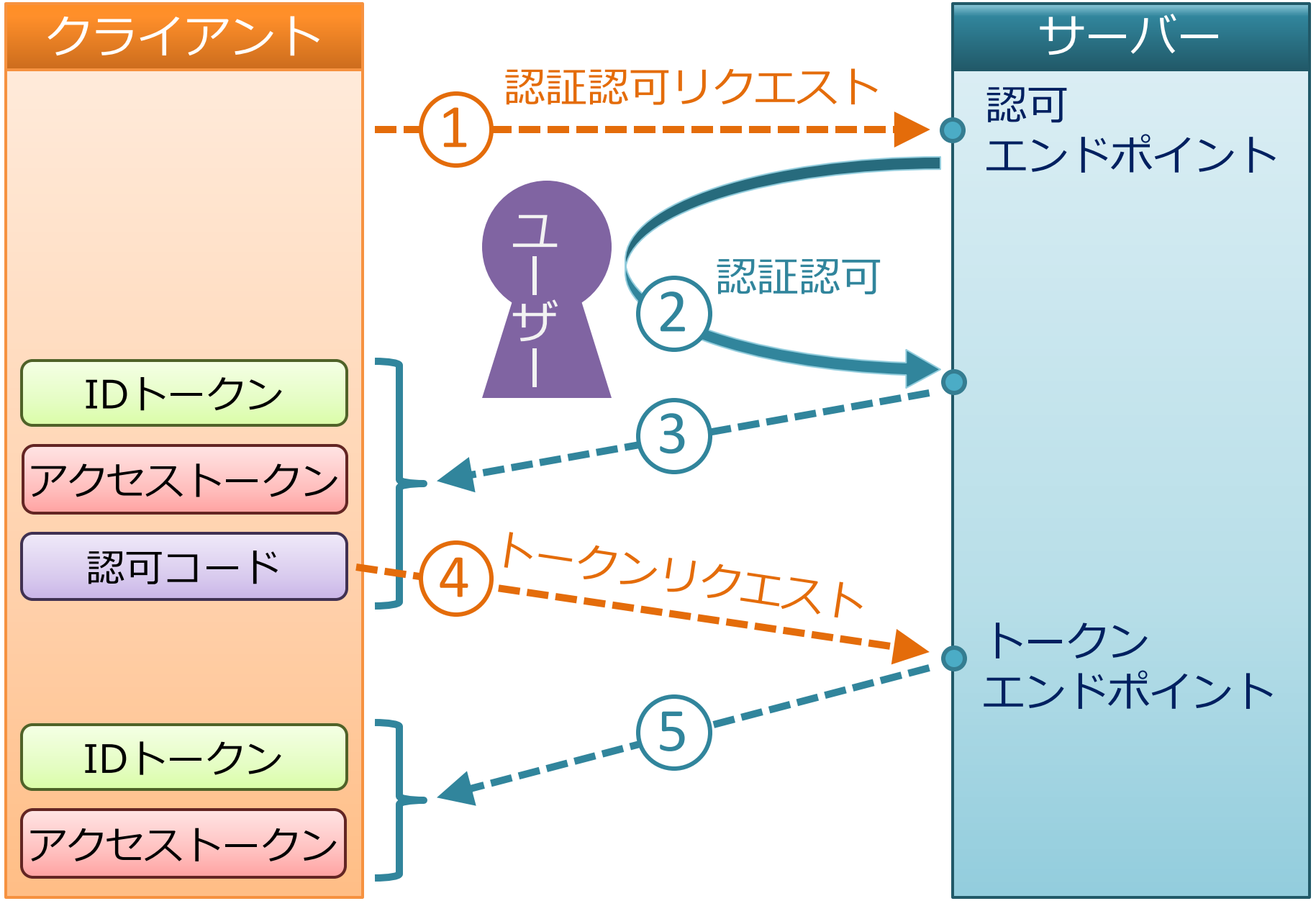
6a. response_type=code token (openid を含まない)



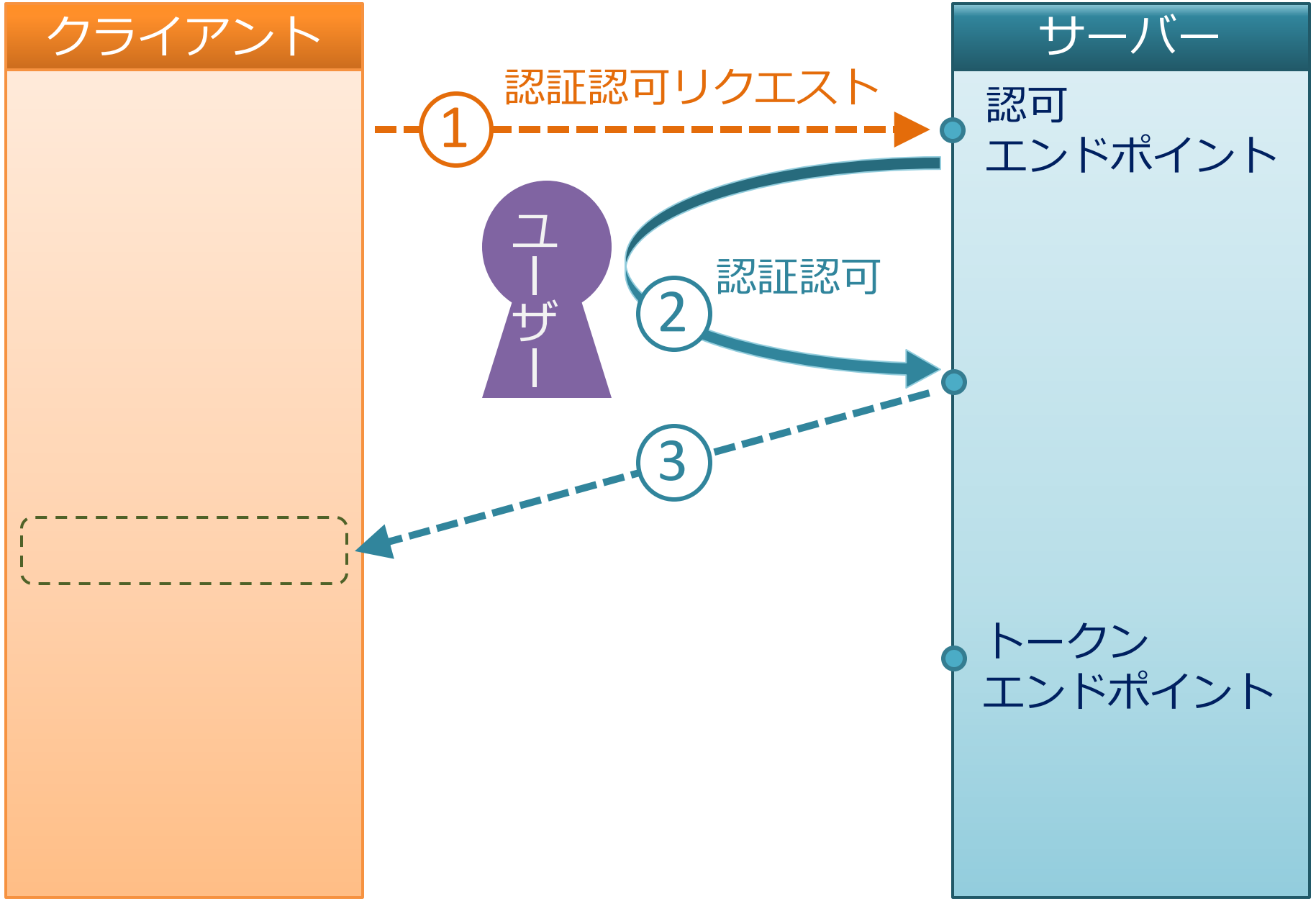
6b. response_type=code token (openid を含む)



7. response_type=code id_token token (openid 必須)



8. response_type=none (openid 不要)



OpenID Connect フローまとめ

1. response_type に **code** が含まれていれば、
   認可エンドポイントから**認可コード**が発行され、
   トークンエンドポイントから**アクセストークン**が発行される。
   - 加えて、scope に **openid** が含まれていれば、
     トークンエンドポイントから**IDトークン**が発行される。
2. response_type に **token** が含まれていれば、
   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。
3. response_type に **id_token** が含まれていれば、
   認可エンドポイントから**IDトークン**が発行される。
4. response_type が **none** ならば、何も発行されない。

JWK


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

JWK Set Document ⟶ *JWK* (JSON Web Key) の集合

JWK Set Document

```
{"keys":[
   { "kty" : "EC",
     "crv" : "P-256",
     "x"   :
   "MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
     "y"   :
   "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
     "use" : "enc",
     "kid" : "1" },
   { "kty" : "RSA",
     "n"   : "0vx7agoebG(中略)pJzKnqDKgw",
     "e"   : "AQAB",
     "alg" : "RS256",
     "kid" : "2011-04-29" }
]}
```

JWK

→Key Type は Elliptic Curve (楕円曲線)

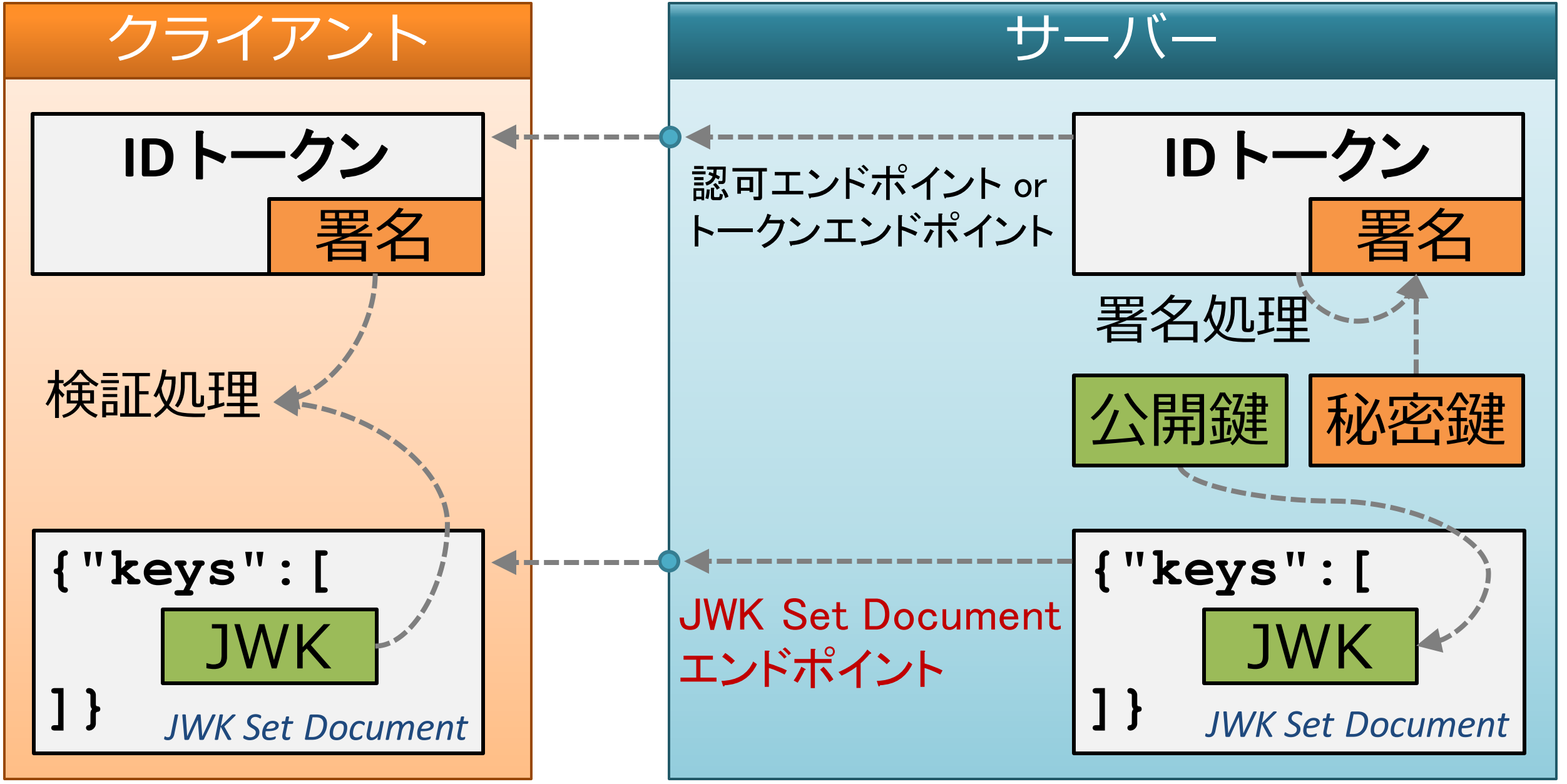
→ Elliptic Curve 固有のパラメーター群

→用途は暗号処理に限る

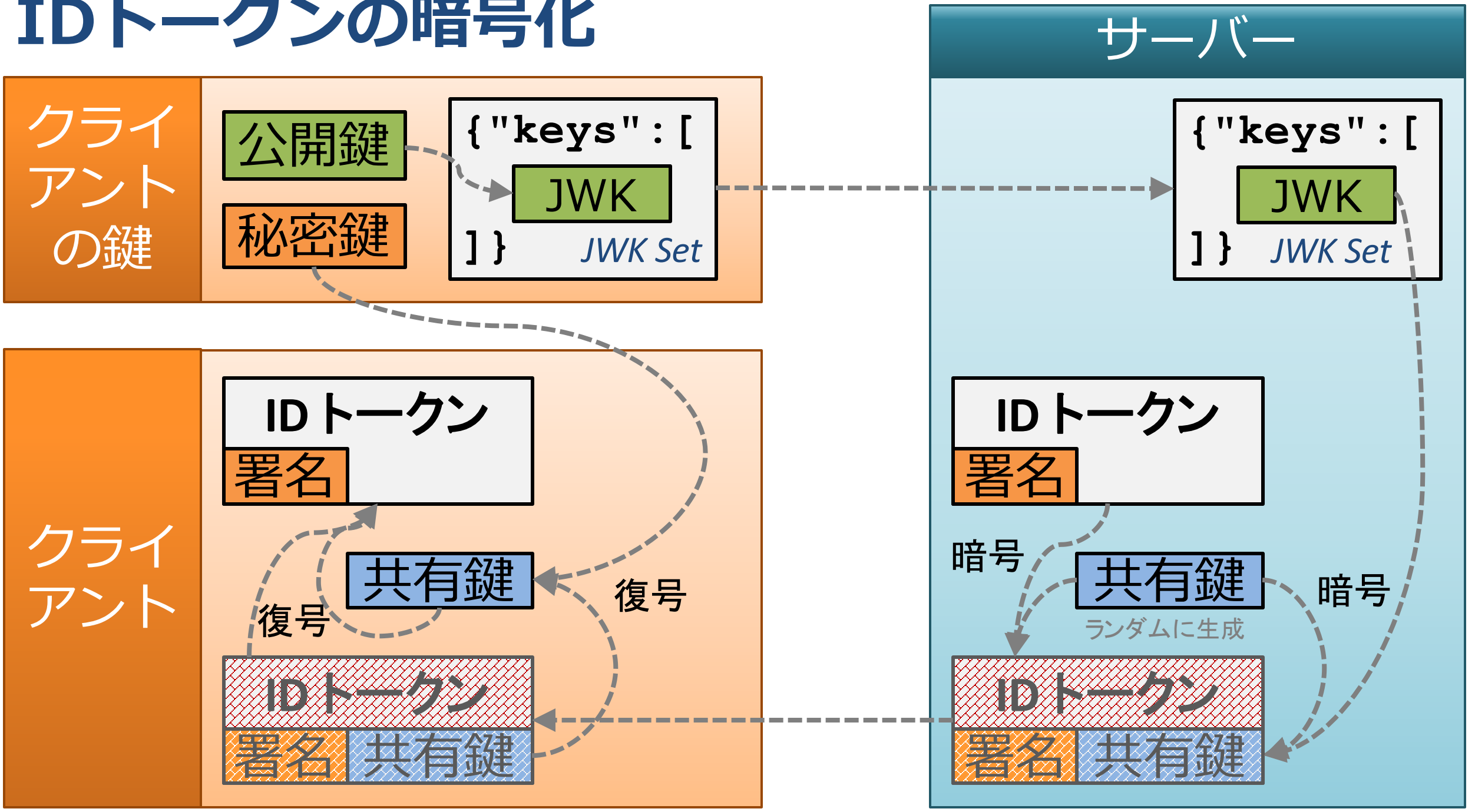
→ Key ID は 1

JWK

IDトークンの署名


クライアント
サーバー
IDトークン
署名
認可エンドポイント or
トークンエンドポイント
IDトークン
署名
署名処理
検証処理
公開鍵
秘密鍵
{"keys":[
JWK
]}
JWK Set Document
JWK Set Document
エンドポイント
{"keys":[
JWK
]}
JWK Set Document

IDトークンの暗号化


サーバー
クライアントの鍵
公開鍵
秘密鍵
{"keys":[
JWK
]}
JWK Set
{"keys":[
JWK
]}
JWK Set
クライアント
IDトークン
署名
共有鍵
復号
復号
IDトークン
署名
共有鍵
IDトークン
署名
暗号
共有鍵
ランダムに生成
暗号
IDトークン
署名
共有鍵

サーバーは、クライアントの JWK Set Document の在り処をどうやって知るのか？

1. クライアントアプリの属性 **`jwks`** の値として、
   JWK Set Document そのものが
   サーバーに登録されている。

2. クライアントアプリの属性 **`jwks_uri`** の値として、
   JWK Set Document の置かれている場所が
   サーバーに登録されている。

***OpenID Connect Dynamic Client Registration 1.0**, 2. Client Metadata*

ID トークンの署名・暗号で使うアルゴリズムは、どうやって決められるのか？

- クライアント属性 **id_token_signed_response_alg** で署名に使うアルゴリズムを指定する。

  HS256, HS384, HS512, RS256, RS384, RS512, ES256, ES384, ES512, PS256, PS384, PS512, none

- クライアント属性 **id_token_encrypted_response_alg** で共有鍵暗号化に使うアルゴリズムを指定する。

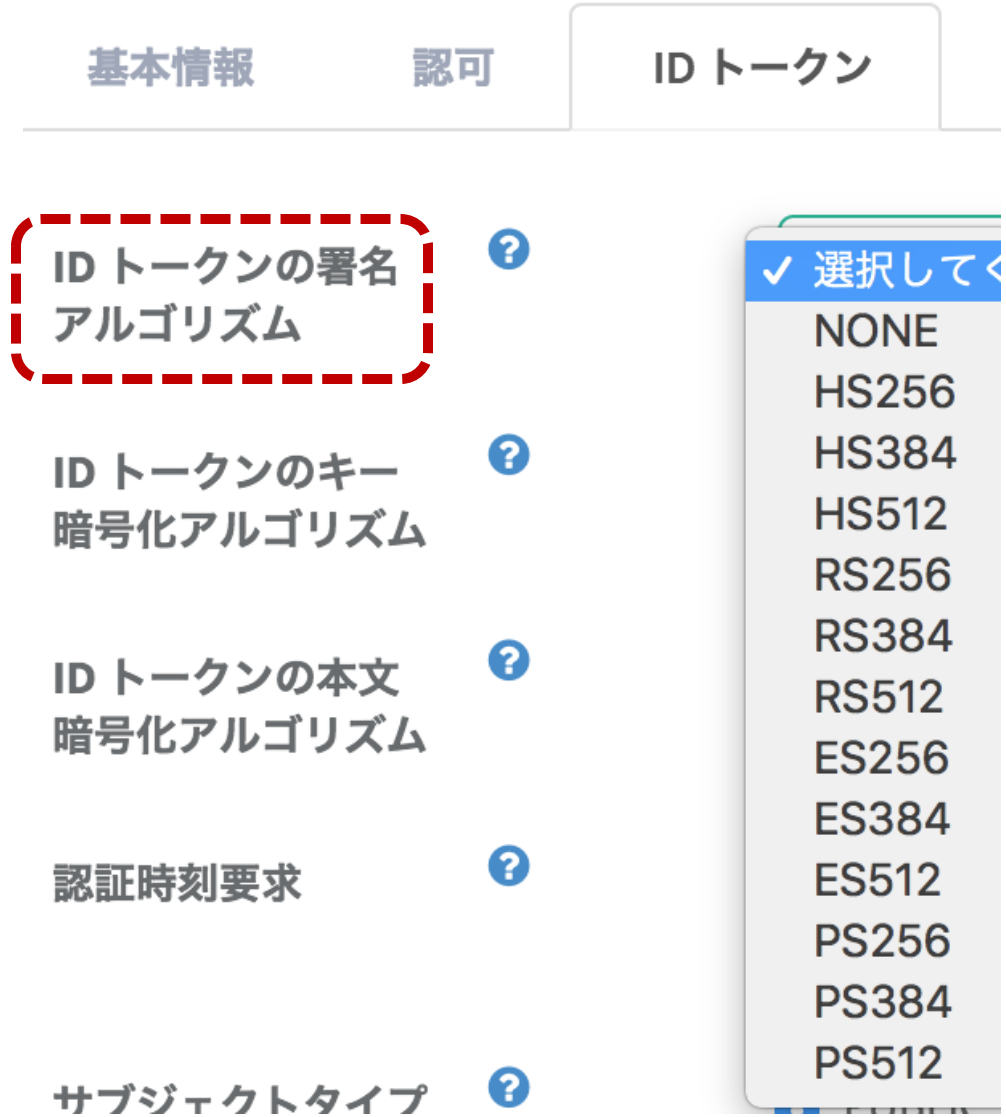
  RSA1_5, RSA-OAEP, RSA-OAEP-256, A128KW, A192KW, A256KW, dir, ECDH-ES, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, A128GCMKW, A192GCMKW, A256GCMKW, PBES2-HS256+A128KW, PBES2-HS384+A192KW, PBES2-HS512+A256KW

- クライアント属性 **id_token_encrypted_response_enc** で本文暗号化に使うアルゴリズムを指定する。

  A128CBC-HS256, A192CBC-HS384, A256CBC-HS512, A128GCM, A192GCM, A256GCM

参考：Authlete の Developer Console（アプリ管理画面）

id_token_signed_response_alg

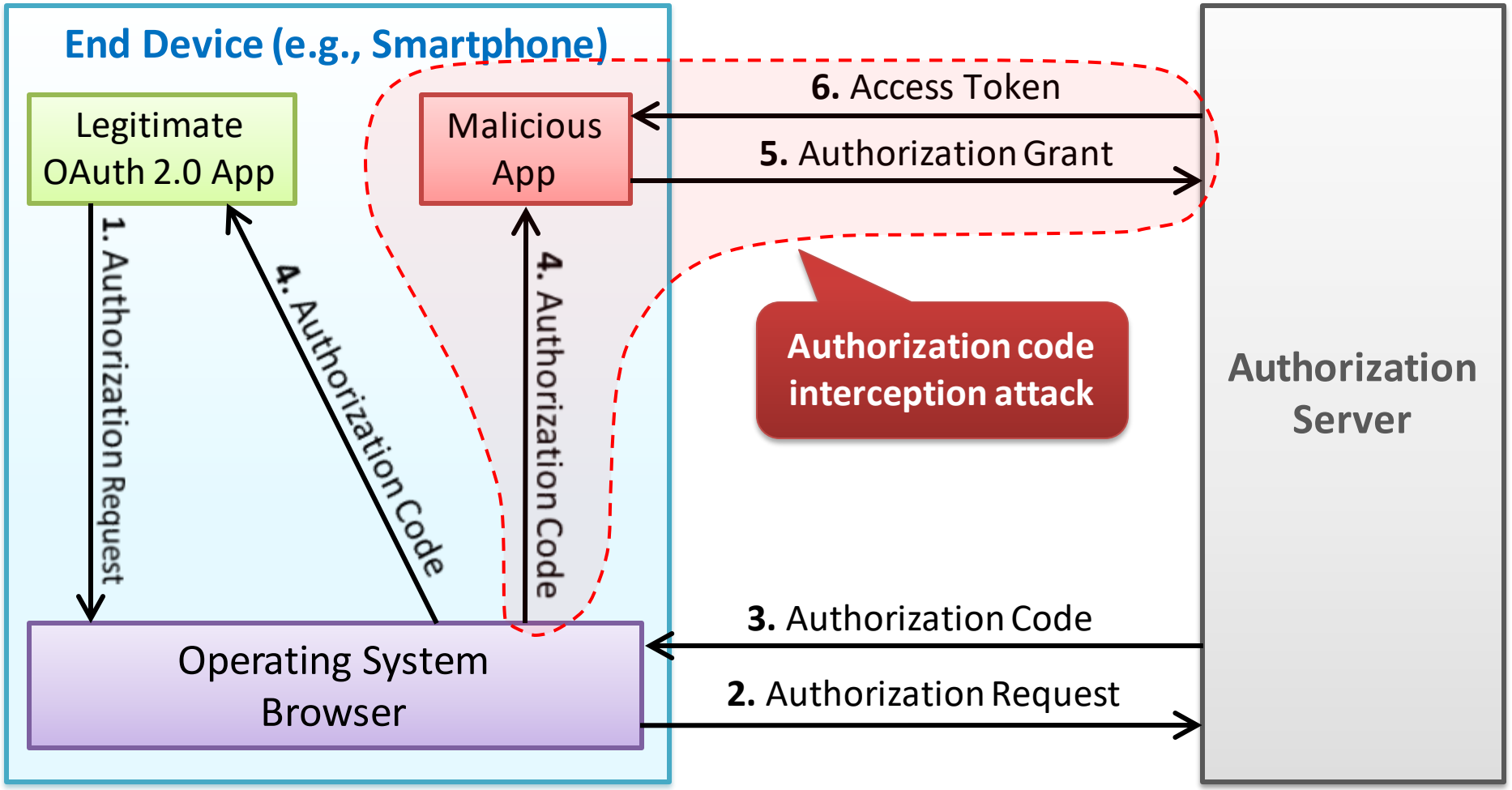


id_token_encrypted_response_alg

PKCE


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

Authorization Code Interception Attack



① アプリが認可リクエストを投げます。
② 認可サーバーに認可リクストを投げます。
③ 認可サーバーが認可コードを発行します。
④ アプリに認可コードが渡されます。

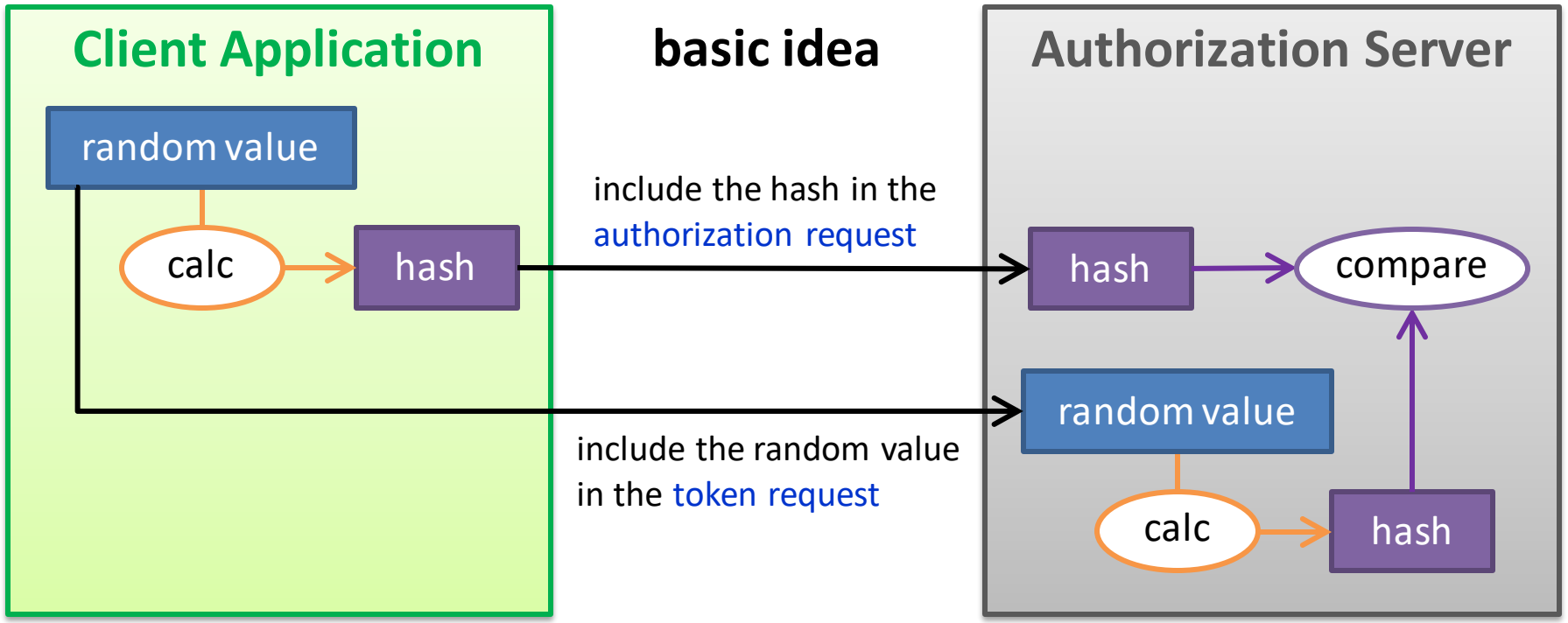
④ 悪意のあるアプリが認可コードを横取りします。
⑤ 正当なアプリのふりをしてアクセストークンを要求します。
⑥ 悪意のあるアプリにアクセストークンが発行されます。

これが『認可コード横取り攻撃』です。

RFC 7636 (Proof Key for Code Exchange by OAuth Public Clients)

認可コード横取り攻撃への対応策として、2015 年 9 月、RFC 7636 (Proof Key for Code Exchange), 通称 PKCE (ピクシー) が策定された。

認可リクエストを発行したアプリとアクセストークンを要求してきたアプリが同一であることを認可サーバー側で確認し、同一である場合のみアクセストークンを発行する。



① ランダム値を生成します。

② ハッシュを計算します。

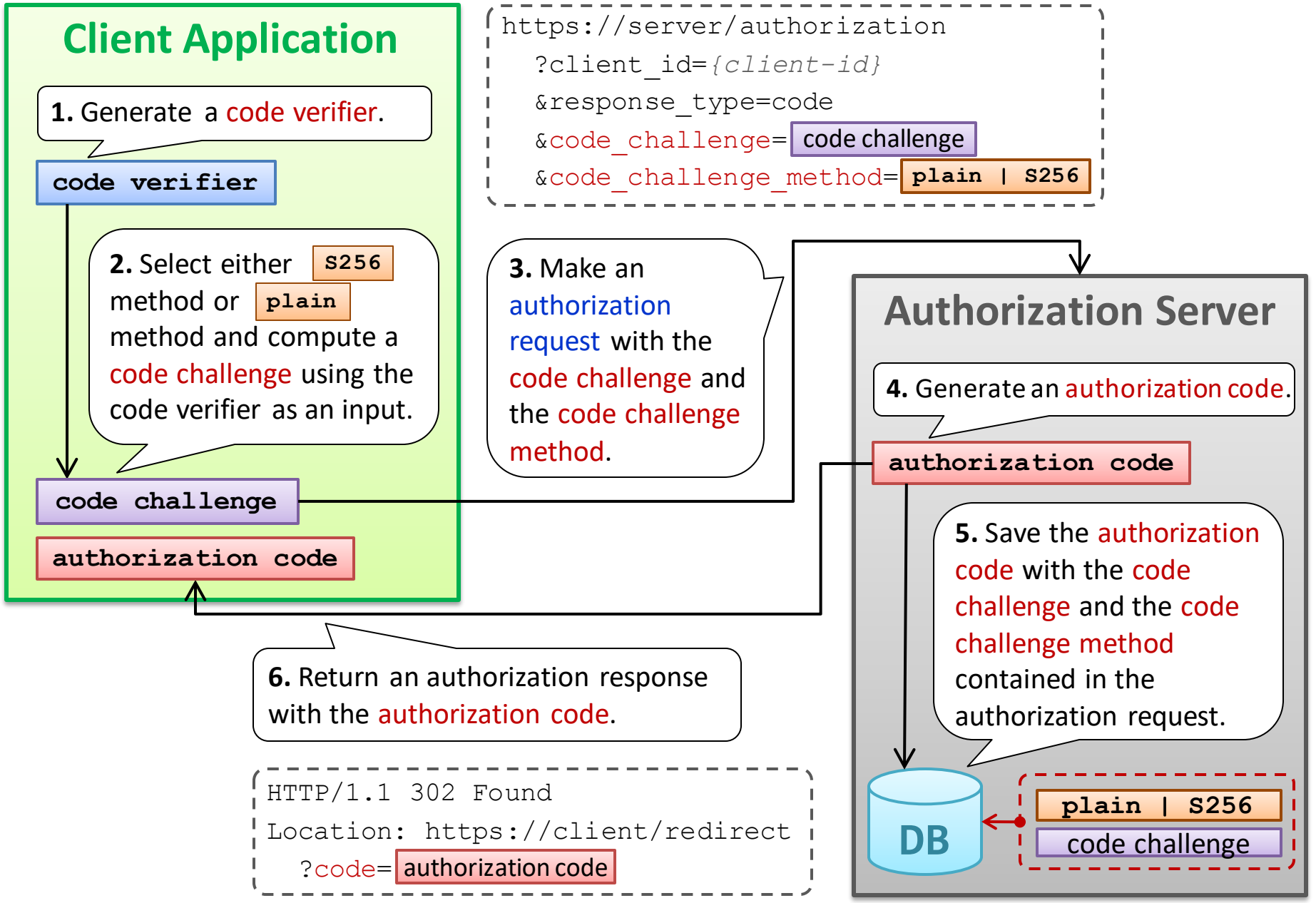
③ 認可リクエストにハッシュを含めます。

④ トークンリクエストにランダム値を含めます。

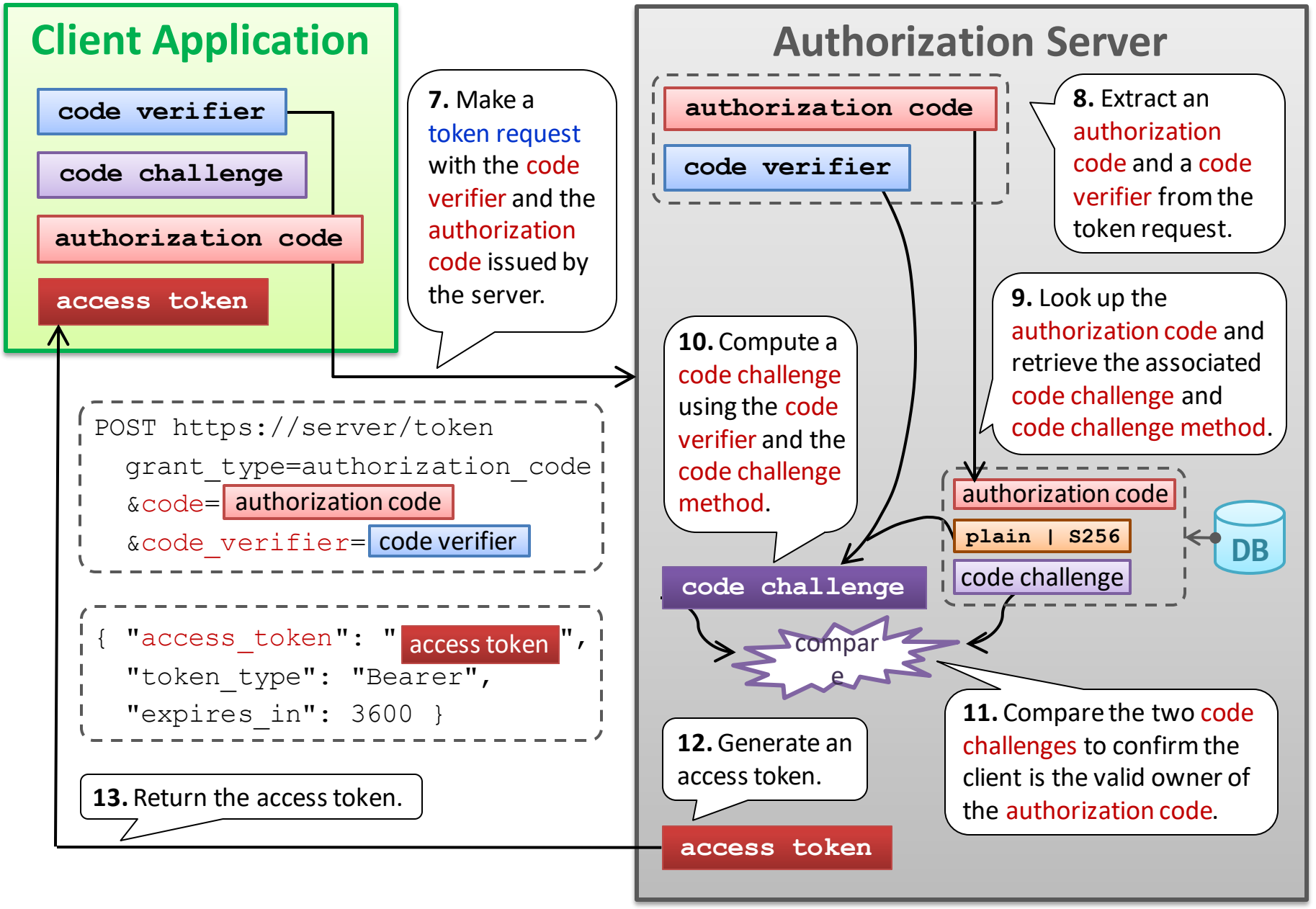
⑤ トークンリクエストに含まれているランダム値からハッシュを計算します。

⑥ 認可リクエストに含まれていたハッシュと比較します。

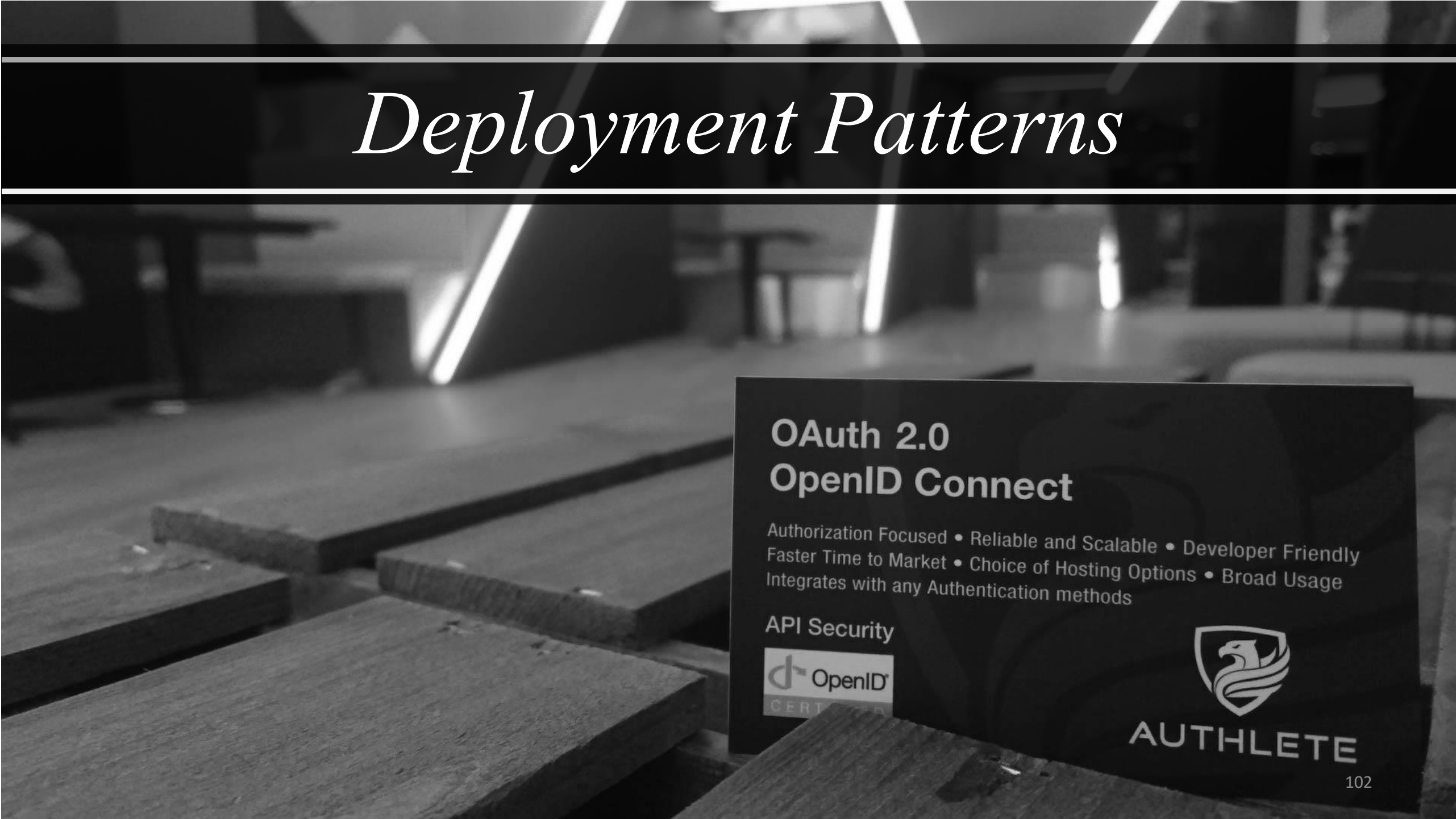
PKCE, Authorization Request and Response

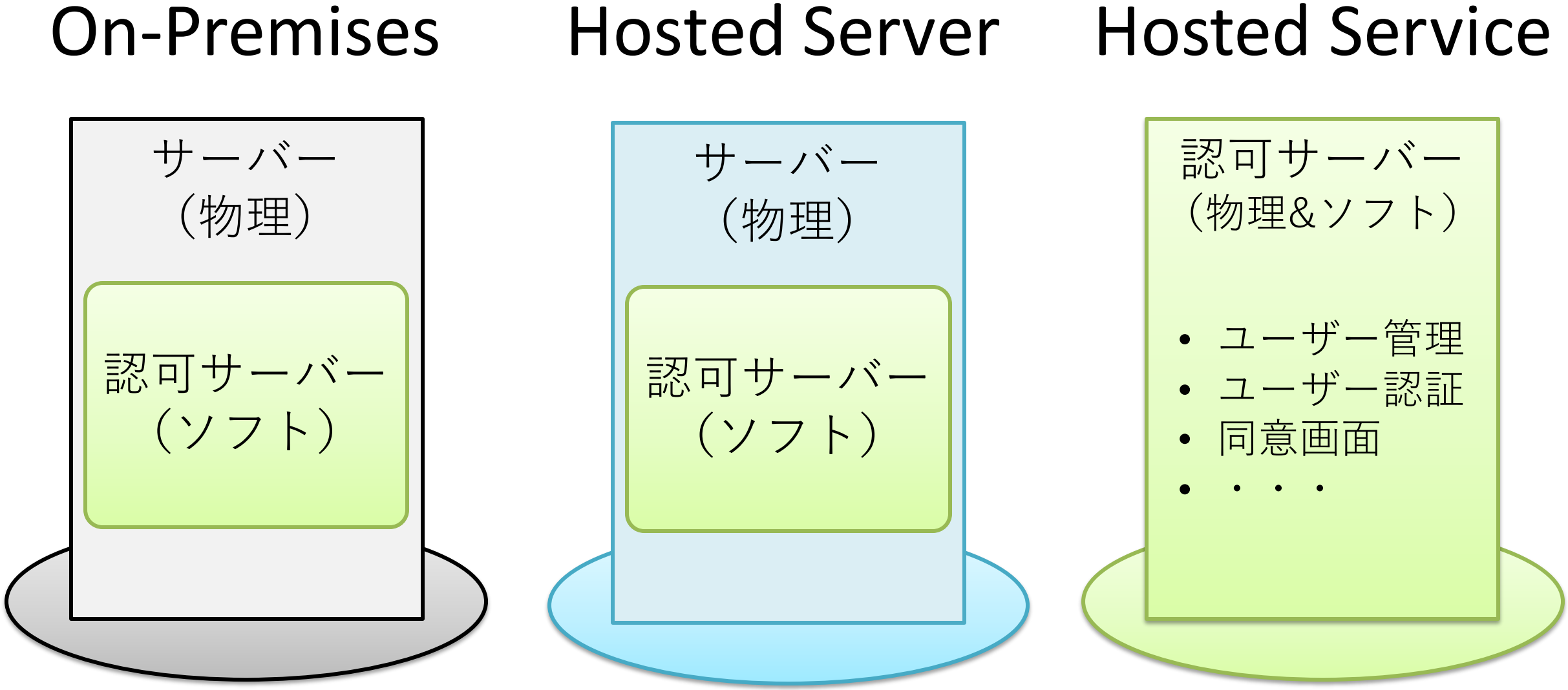


PKCE, Token Request and Response



Deployment Patterns



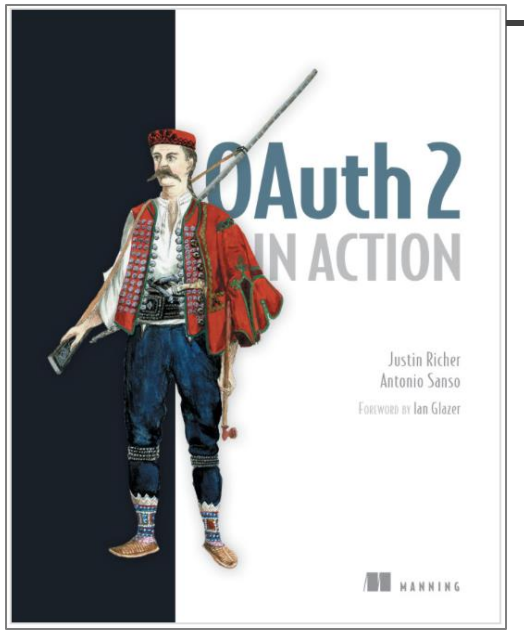

On-Premises
サーバー
（物理）
認可サーバー
（ソフト）
Hosted Server
サーバー
（物理）
認可サーバー
（ソフト）
Hosted Service
認可サーバー
（物理&ソフト）
• ユーザー管理
• ユーザー認証
• 同意画面
• ・・・
オンプレミス環境
クラウド
IaaS
クラウド
SaaS

Medium

Justin Richer Follow

Jul 27 · 9 min read

Deployment and Hosting Patterns in OAuth

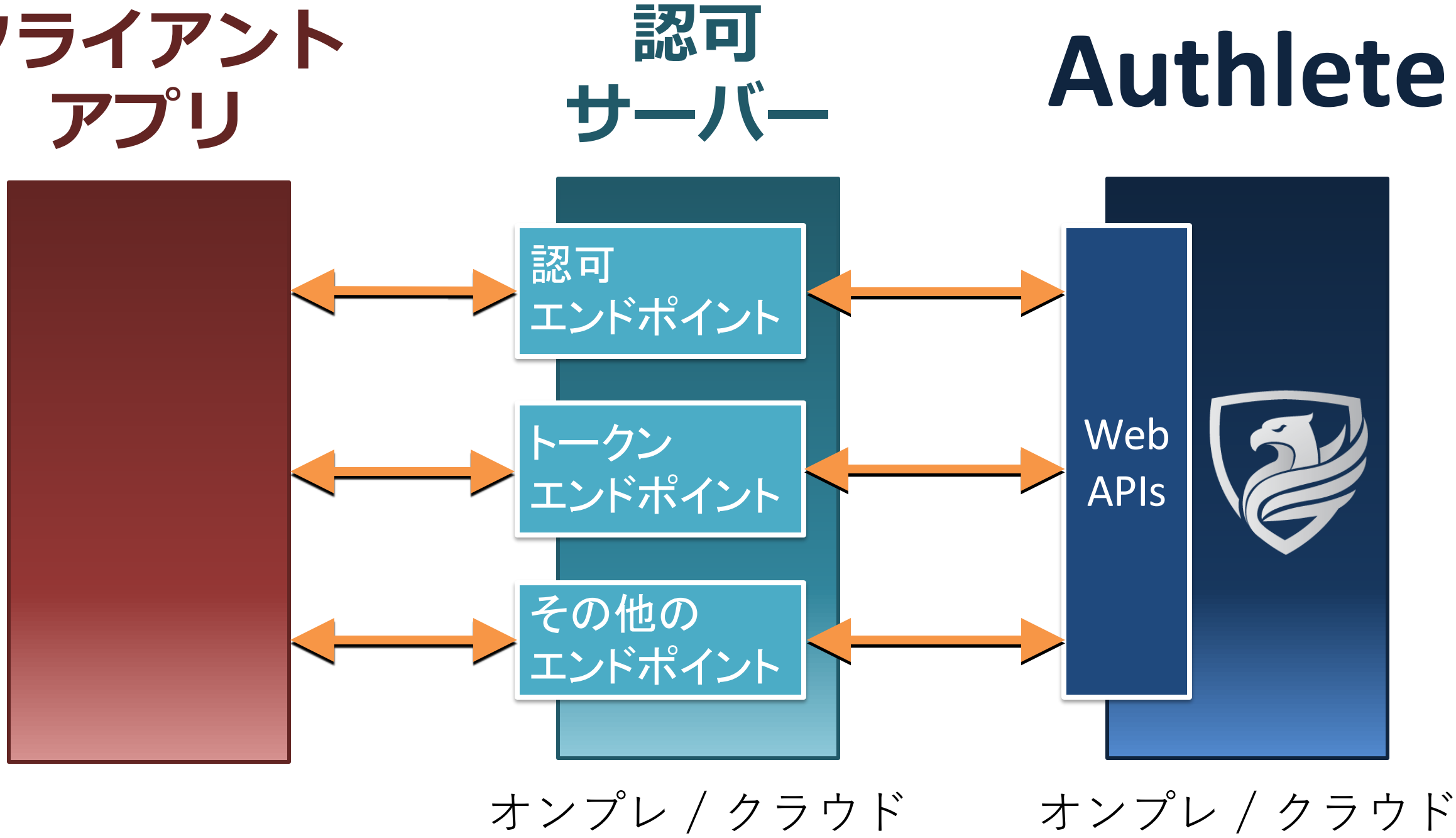


日本語訳

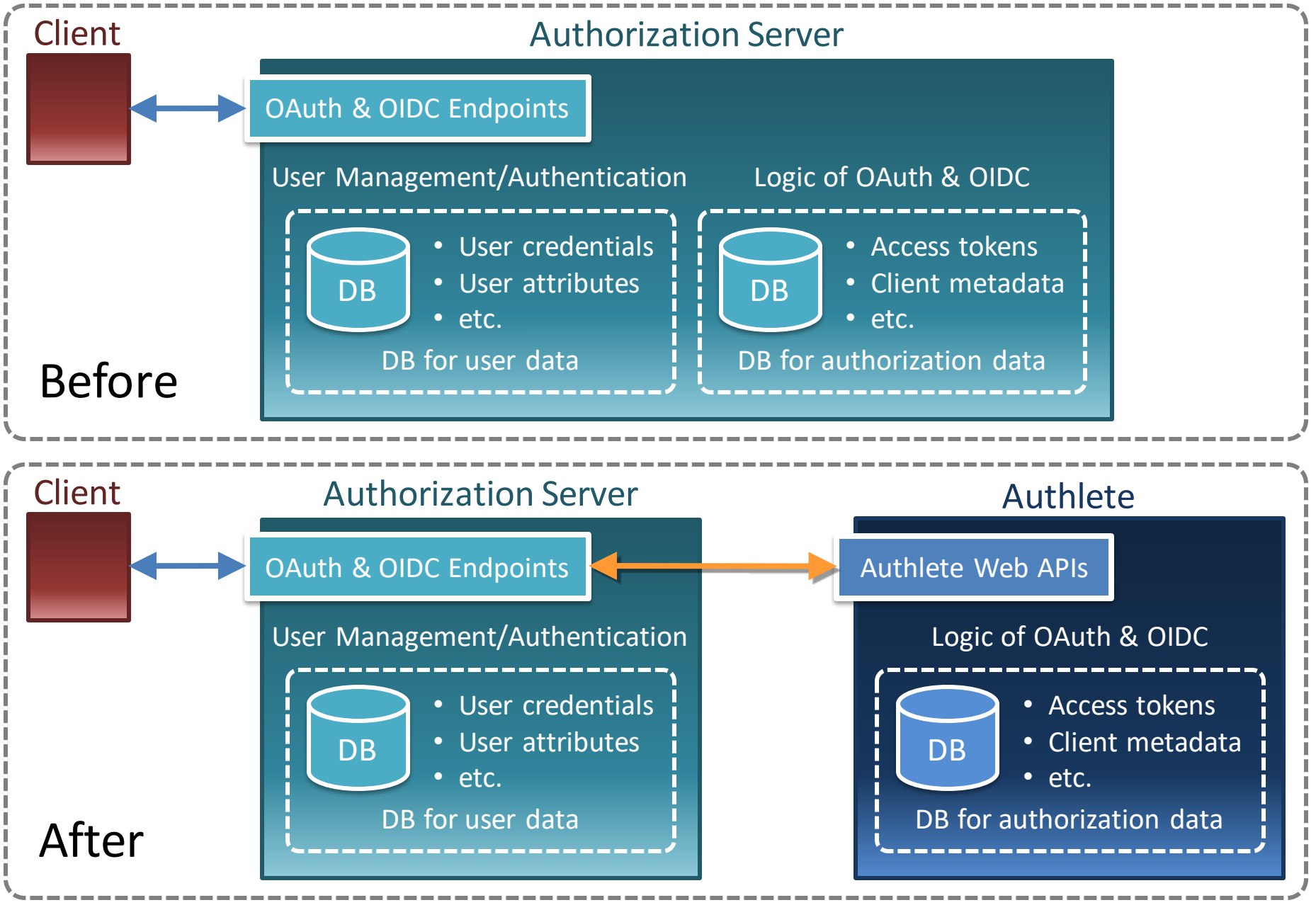
Authlete社監修

1. On-Premises
2. Hosted Server
3. Hosted Service
4. Semi-Hosted Service

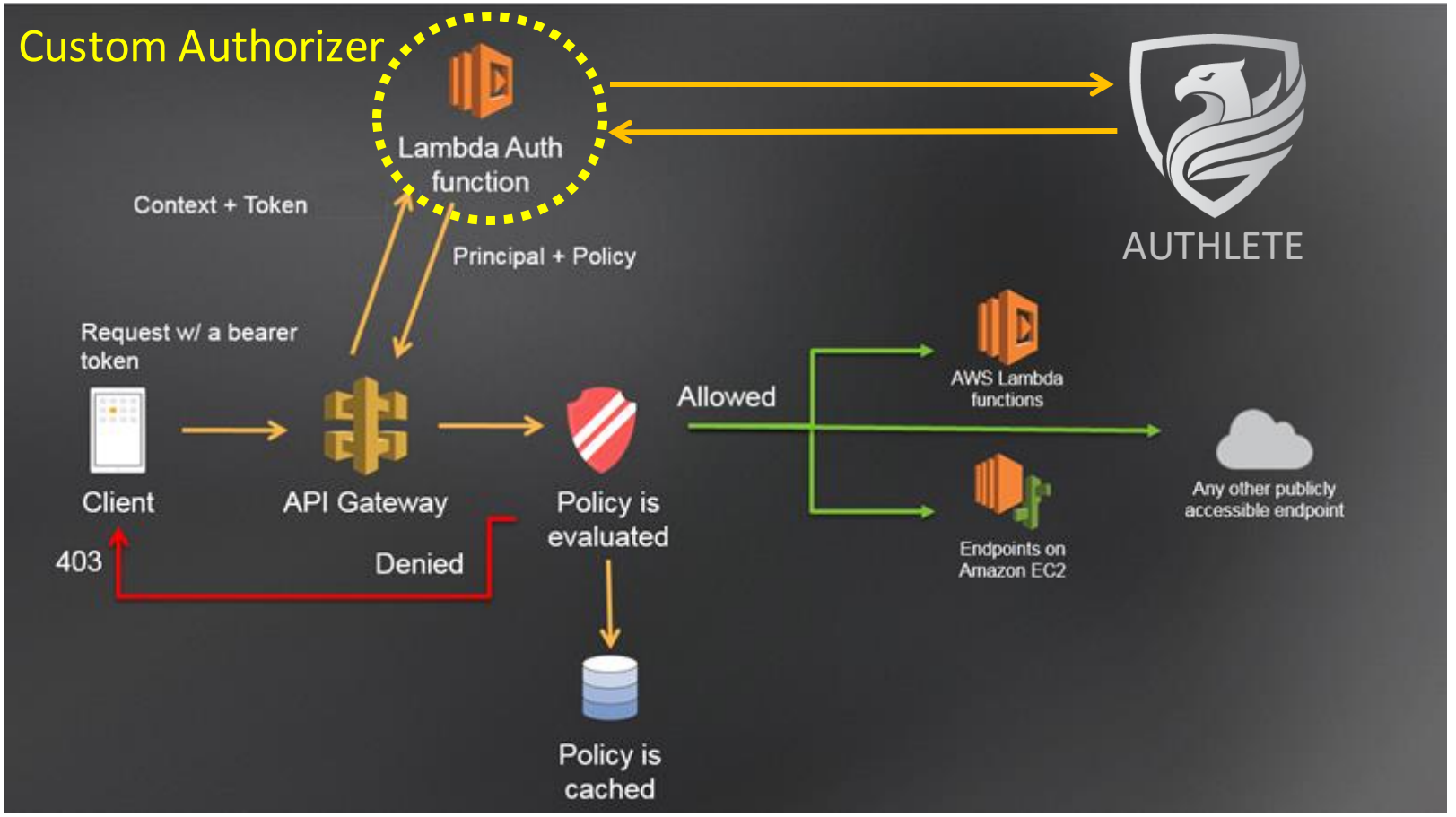
AUTHLETE


クライアント
アプリ
認可
サーバー
Authlete
認可
エンドポイント
トークン
エンドポイント
その他の
エンドポイント
Web
APIs
オンプレ / クラウド
オンプレ / クラウド

Authlete 以前と Authlete 以後



API 管理と OAuth/OIDC 実装の分離



Amazon API Gateway は、クライアントが API に提示してきたアクセストークンの検証を外部に移譲する仕組み (Custom Authorizer) を提供している。

Custom Authorizer の実装の中からさらに、外部の認可サーバーに作業を移譲することで、アクセストークンの検証を簡単に行うことができる。

MTLS (RFC 8705) 実装の分離

RFC 8705 : OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens

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

OAuth 2.0 Flows + Authlete

Authlete の背景にある哲学

エンドユーザー認証

認証方法には様々なものがあるが、いずれも、

エンドユーザーの一意識別子を特定する処理

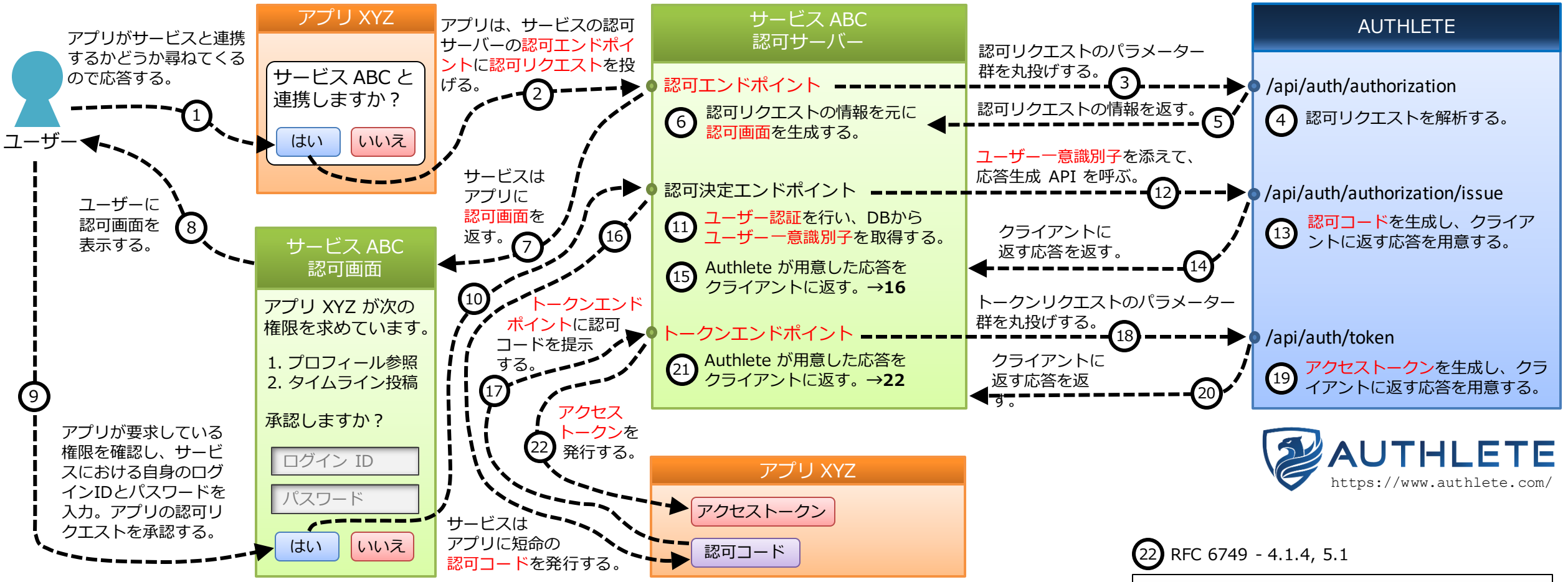
アクセストークンを格納する DB テーブル

どの**ユーザー**が、どの**クライアントアプリ**に、どの**権限**を与えたか、
を管理するため、ユーザー一意識別子を表すカラムを持っているが、

ユーザー 一意識別子	クライアント アプリ識別子	許可された スコープ群	有効 期限	表現 文字列	他
user001	client001	email profile	2018-04-18	abc...	...

エンドユーザー一意識別子は外部から提供してもらう。

認可コードフロー + AUTHLETE



AUTHLETE
https://www.authlete.com/

(2) RFC 6749 - 4.1.1, RFC 7636 - 4.3

```
GET {認可エンドポイント}
  ?response_type=code
  &client_id={クライアントID}
  &redirect_uri={リダイレクトURI}
  &scope={スコープ群}
  &state={任意文字列}
  &code_challege={チャレンジ}
  &code_challenge_method=S256
  HTTP/1.1
HOST: {認可サーバー}
```

(16) RFC 6749 - 4.1.2

```
HTTP/1.1 302 Found
Location: {リダイレクトURI}
  ?code={認可コード}
  &state={任意文字列}
```

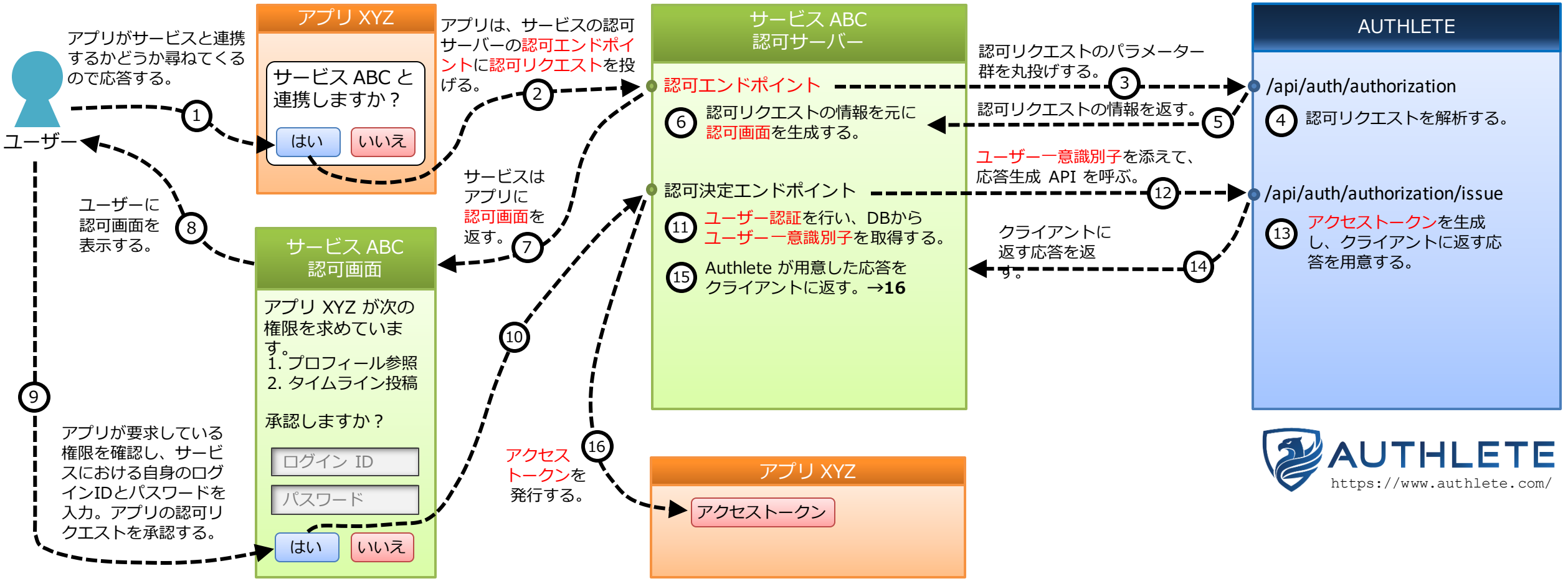
(17) RFC 6749 - 4.1.3, RFC 7636 - 4.5

```
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

grant_type=authorization_code&
code={認可コード}&
client_id={クライアントID}&
redirect_uri={リダイレクトURI}&
code_verifier={ベリファイヤー}
```

(22) RFC 6749 - 4.1.4, 5.1

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

{
  "access_token": "{アクセストークン}",
  "token_type": "{トークンタイプ}",
  "expires_in": {有効秒数},
  "refresh_token": "{リフレッシュトークン}",
  "scope": "{スコープ群}"
}
```

インプリシットフロー + AUTHLETE



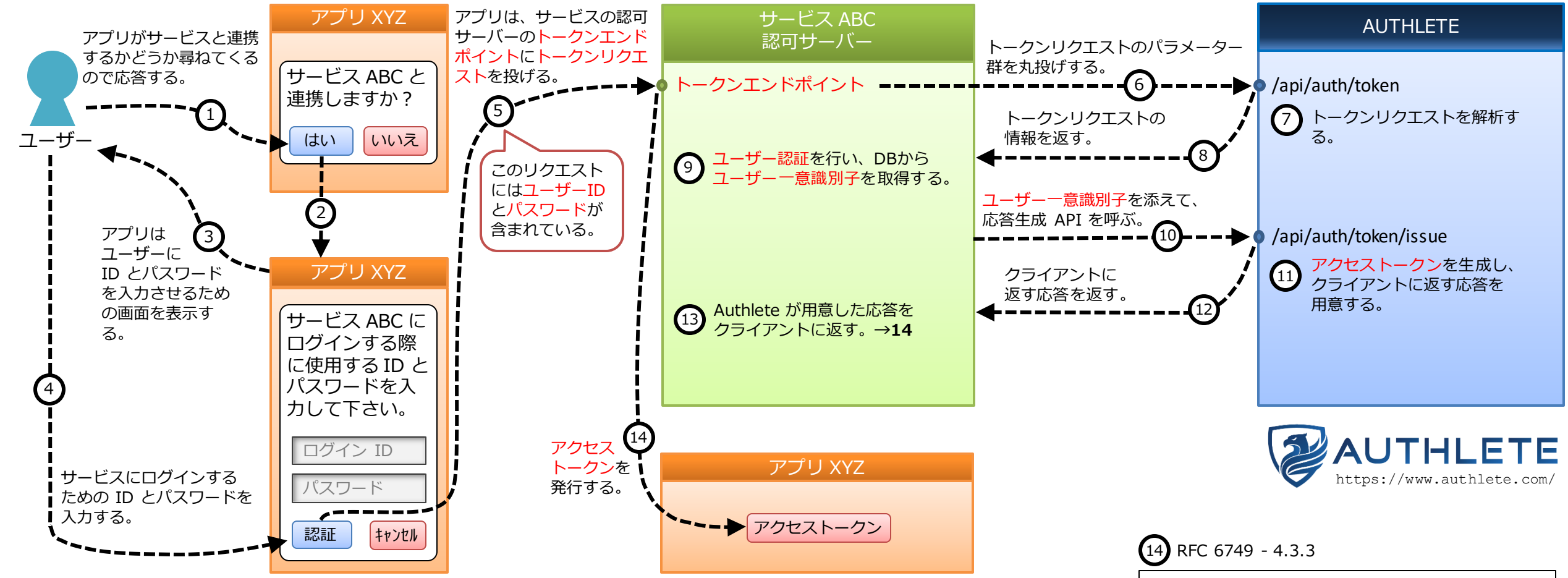
② RFC 6749 - 4.2.1

```
GET {認可エンドポイント}
  ?response_type=token
  &client_id={クライアントID}
  &redirect_uri={リダイレクトURI}
  &scope={スコープ群}
  &state={任意文字列}
  HTTP/1.1
HOST: {認可サーバー}
```

⑯ RFC 6749 - 4.2.2

```
HTTP/1.1 302 Found
Location: {リダイレクトURI}
  #access_token={アクセストークン}
  &token_type={トークンタイプ}
  &expires_in={有効秒数}
  &scope={スコープ群}
  &state={任意文字列}
```

リソースオーナー・パスワード・クレデンシャルズフロー + AUTHLETE



AUTHLETE
https://www.authlete.com/

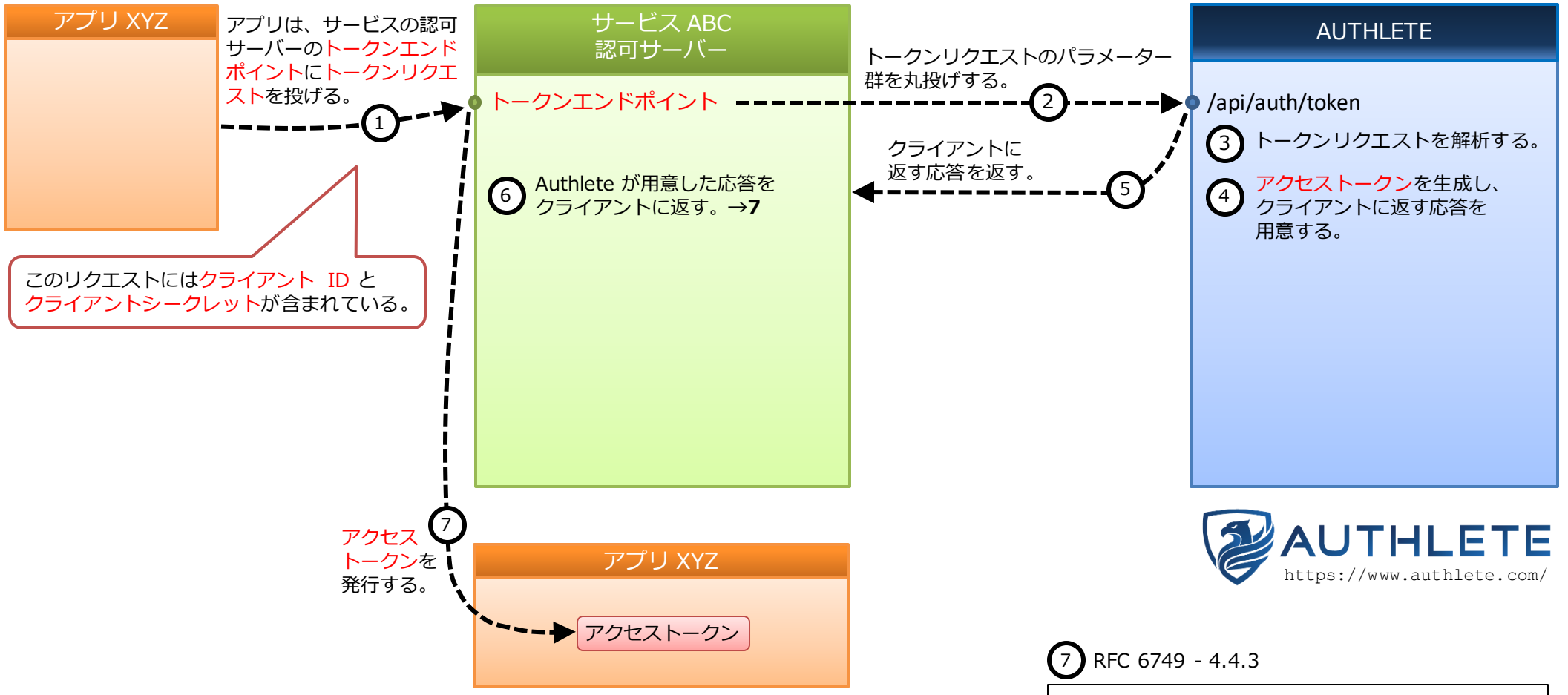
5 RFC 6749 - 4.3.2

```
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

grant_type=password&
username={ユーザーID}&
password={パスワード}&
client_id={クライアントID}&
scope={スコープ群}
```

14 RFC 6749 - 4.3.3

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

{
  "access_token": "{アクセストークン}",
  "token_type": "{トークンタイプ}",
  "expires_in": {有効秒数},
  "refresh_token": "{リフレッシュトークン}",
  "scope": "{スコープ群}"
}
```

クライアント・クレデンシャルズフロー + AUTHLETE



AUTHLETE
https://www.authlete.com/

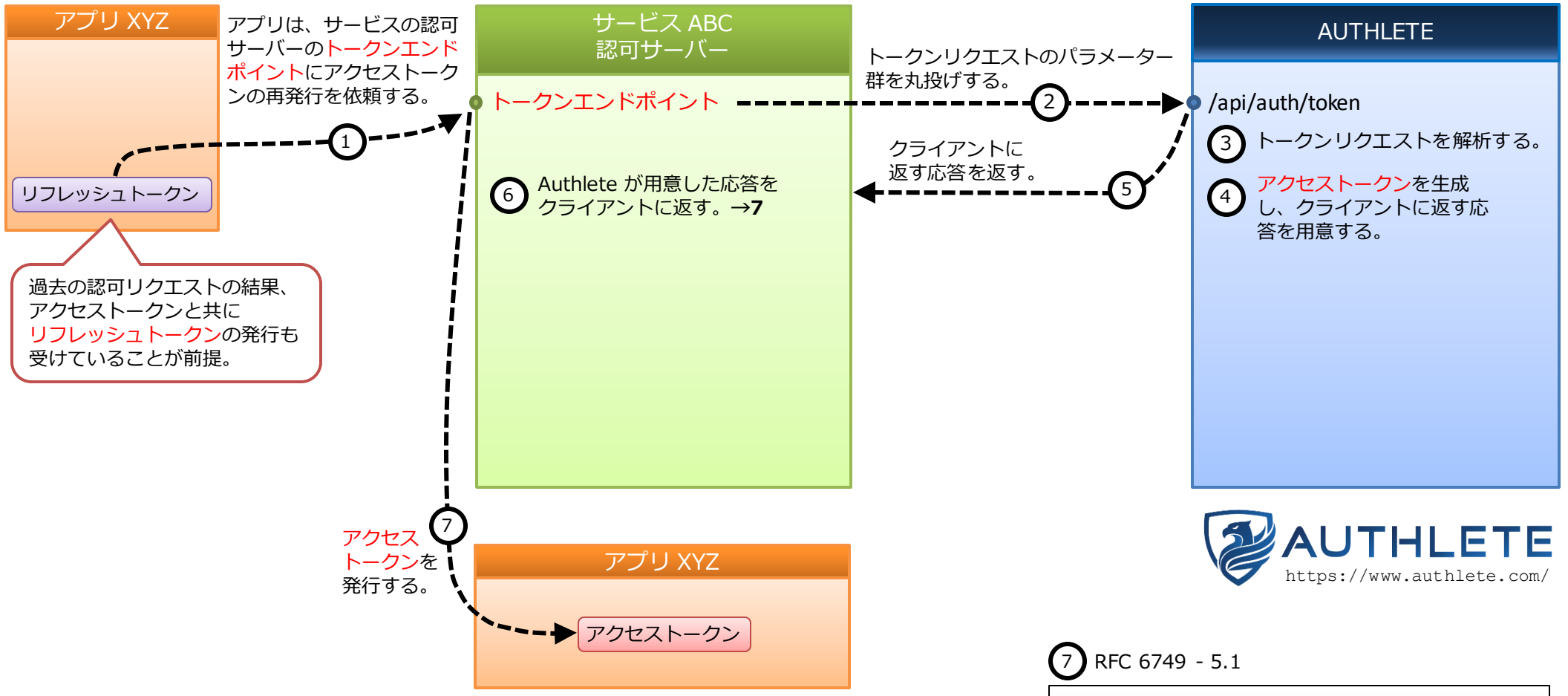
1 RFC 6749 - 4.4.2

```
POST {トークンエンドポイント} HTTP/1.1
HOST: {認可サーバー}
Authorization:
  Basic {クライアントクレデンシャルズ}
Content-Type:
  application/x-www-form-urlencoded

grant_type=client_credentials&
scope={スコープ群}
```

7 RFC 6749 - 4.4.3

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

{
  "access_token": "{アクセストークン}",
  "token_type": "{トークンタイプ}",
  "expires_in": {有効秒数},
  "scope": "{スコープ群}"
}
```

リフレッシュトークンフロー + AUTHLETE



AUTHLETE
https://www.authlete.com/

1 RFC 6749 - 6

```
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

grant_type=refresh_token&
refresh_token={リフレッシュトークン}&
client_id={クライアントID}&
scope={スコープ群}
```

7 RFC 6749 - 5.1

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

{
  "access_token": "{アクセストークン}",
  "token_type": "{トークンタイプ}",
  "expires_in": {有効秒数},
  "refresh_token": "{リフレッシュトークン}",
  "scope": "{スコープ群}"
}
```

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