Name	Presence	Value	Description
type	REQUIRED	“PasswordOOB”	
generator	OPTIONAL	String	If a client side device or algorithm is needed to derive the password, it can be referenced by this property. E.g. a trust service provider can have issued multiple tokens and allows the user to select one of them using this property.

Authentication object properties:

Name	Presence	Description
-		

An empty authentication object is required to indicate to the server that some out-of-band data must be acquired for this authorization.

Example authentication type:

```
{
    "type": "PasswordOOB",
    "id": "PIN2",
    "label": "PIN2"
}
```

Example authentication object:

```
{
    "id": "PIN2"
}
```

8.3.1.5 ChallengeResponse, in band response

The authorization process may be based on a challenge response protocol where the response is created by a client side mechanism. The mechanism itself is out of scope for this specification. The response is then sent via *credentials/authorize* in-band.

This is typically used where - the user is in possession of a token that requires input of a challenge and provides a OTP that needs to be sent to the service as a response. - A literal challenge is sent to the user via SMS, email or other out of band channel to be sent to the service as a response.

Authentication type properties:

Name	Presence	Value	Description
type	REQUIRED	“ChallengeResponse”	
format	OPTIONAL	“A”|“N”	Specifies the format of the password: - “A”: alphanumeric text; allowed characters: A-Z | a-z | 0-9 - “N”: numeric text If omitted, any character is allowed.
generator	OPTIONAL	String	If a client side device or algorithm is needed to derive the password, it can be referenced by this property. E.g. a trust service provider may have issued multiple tokens and allows the user to select one of them using this property.

Authentication object properties:

Name	Presence	Description

Name	Presence	Description
value	REQUIRED	The concrete response value.

This authentication object type requires the signature application to request a challenge from the service provider using *credentials/getChallenge*. The *credentials/getChallenge* method needs the id of the authentication object to decide which challenge to generate. The reply is either - the challenge itself, using a HTTP status code 200. In this case, the signature application is required to display the challenge in order to inform the user and prepare her to derive the response. - A HTTP status code 201. The challenge is sent out of band to the user. The signature application only provides means to enter the response for the user.

Example authentication type

```
{
    "type": “ChallengeResponse”,
    "id": “OTP”.
    "label": “OTP”
}
```

Example authentication object

```
{
    "id": “OTP”,
    "value": “sadf8aef”
}
```

8.3.1.6 ChallengeResponse, out of band response

The authorization process is based on a challenge response protocol where the response is created by a client-side mechanism. The mechanism itself is out of scope for this specification. The response is then sent via some out of band mechanism that is again outside the scope of this specification.

Authentication type properties:

Name	Presence	Value	Description
type	REQUIRED	“ChallengeResponseOOB”	
generator	OPTIONAL	String	If a client side device or algorithm is needed to derive the password, it can be referenced by this property. E.g. a trust service provider may have issued multiple tokens and allows the user to select one of them using this property.

Authentication object properties:

Name	Presence	Description
		There is no data sent in band.

This authentication object type requires the signature application to request a challenge using credentials/getChallenge. The credentials/getChallenge method needs the id of the authentication object to decide which challenge to generate. The reply is either - the challenge itself, using a HTTP status code 200. In this case, the signature application is required to display the challenge in order to inform the user and prepare him to derive the response. - A HTTP status code 201 The challenge is sent out of band to the user. The signature application only provides means to enter the response for the user.

Example authentication type

```
{
    "type": “ChallengeResponseOOB”,
    "id": “SMS”.
    "label": “SMS”
}
```

Example authentication object

```
{
    "id": “SMS”
}
```

An empty authentication object is required to indicate to the server that some out-of-band data must be acquired for this authorization.

8.4 OAuth 2.0 Authorization

OAuth 2.0 is an authorization framework that enables applications to obtain access to HTTP based services. It provides client applications a “secure delegated access” to server resources on behalf of a resource owner. In the context of this specification, the signature application is the client application. This allows resource owners to authorize third-party access to their server resources without sharing their credentials.

Using the OAuth 2.0 authorization scheme, the signature application will use the remote service’s authorization server for user authentication and access authorization. After a successful authentication and authorization, the authorization server of the remote service will provide the signature application with an access token that the signing application will use to authorize access to the remote service’s resources.

The following OAuth 2.0 grant types as defined in RFC 6749 [11] MAY be used:

- Authorization Code
- Client Credentials
- Refresh Token

The implicit grant SHALL NOT be used, due to security flaws.

Any provider implementing an OAuth 2.0 authorization flow SHALL follow the recommendations from OAuth 2.0 Security Best Current Practice [20]. The OAuth 2.0 authorization mechanisms can be used for different use cases, determined by the respective scope.

The following scopes are defined by this specification:

- “service” - used to request service authorization
- “credential” - used to request authorization for creating one or more signatures with a certain credential or fufiling the requirements of a certain signature qualifier.

An access token with the “credential” scope can be used instead of a classical “SAD” as obtained via **credentials/authorize** or **credentials/extendTransaction**. Such an access token will be sent to the remote signing API in the AUTHORIZATION header. For backward compatibility, it can also be sent as “SAD” parameter value.

An access token with the “credential” scope also includes the service authorization for the requests **credentials/info**, **signatures/signHash**, **signatures/signDoc** in conjunction with the respective credential or for the request **signatures/signDoc** in conjunction with the respective credential qualifier. As a consequence, an application that has obtained an access token for scope “credential” does not need an additional access token with scope “service” in order to use these requests.

This is useful if the application already has all the information required by **signature/signDoc** or **signature/signHash** and wants to save the additional roundtrip for service authorization. Using **signature/signDoc** with a signature qualifier to create signatures is one example. In this case, the signing application does not need to lookup the available certificates before starting the credential authorization process.

Note 12: In the course of authorizing the “credential” scope, the authorization server authenticates the client and conveys the client identity in the respective access token (which is equivalent to the service authorization).

A remote service can implement a single OAuth 2.0 authorization server supporting all beforementioned scopes (and possibly more) or just some of them.

In order to be able to use an OAuth 2.0 authorization mechanism, the signing application needs to be in possession of an OAuth `client_id` valid for the respective OAuth authorization server and corresponding credentials. The way this `client_id` is setup and the client authentication mechanism used is out of scope for this specification. Implementations can utilize any of the client authentication methods defined in the IANA “OAuth Token Endpoint Authentication Methods” registry established by IETF RFC 7591 [24].

The following sections describe the OAuth 2.0 endpoints supported by this specification and how to invoke them. Notice that the Client Credential flow is not described separately because it can be invoked by means of the **oauth2/token** endpoint, as defined in oauth2/token, using a *grant_type* with value “client_credentials”.

Tokens issued by OAuth 2.0 authorization endpoints SHOULD be revoked by using the authorization server’s revocation endpoint **oauth2/revoke**, as defined in oauth2/revoke, if supported. Tokens MAY also be revoked by calling the remote service’s **auth/revoke** method, as defined in auth/revoke, if supported.

The **info** method, as defined in info, provides the signing application with the OAuth endpoints location information. There are two options for the remote service:

- the parameter `oauth2` provides a base URL for all OAuth 2.0 endpoints. The URI path components of the supported OAuth 2.0 endpoints specified in oauth2/authorize, oauth2/pushed_authorize, oauth2/token, and oauth2/revoke SHALL be concatenated to the OAuth 2.0 base URI.
- the parameter `oauth2Issuer` provides the issuer URL of authorization server. The signing application SHALL obtain all endpoint URLs and further metadata about the OAuth authorization server as specified in IETF RFC 8414 “OAuth 2.0 Authorization Server Metadata” [23]. This prevents security (trustworthiness of endpoints) and operational (endpoints change) issues.

Note 13: OAuth in conjunction with the authorization code flow gives the authorization server full screen control in the course of the authorization process. This allows the authorization server to utilize user authentication means at its own discretion without the need for this specification to

cater for certain authentication means. This, for example, allows authorization servers to utilize FIDO/WebAuth [i.11] for strong and (optionally) password less authentication. The authorization server may use the WebAuthn API as exposed by the user agent to authenticate the user based on the keys maintained in the platform or external authenticator.

8.4.1 Restricted access to authorization servers

OAuth 2.0 authorization frameworks typically offer an open and unrestricted authorization endpoint. In the context of the authorization server of a remote service, this means that a user will have no restrictions while accessing the **oauth2/authorize** endpoint, as defined in oauth2/authorize.

However, a remote service may need to restrict users from accessing its authorization server. There are two common cases when a restriction would be desirable: with remote services connected to Corporate Identity Management services or connected to public Electronic Identity (eID) frameworks. In the former case, the remote service may be required to prevent access to users that are not affiliated with the Corporate, in the latter the remote service may be restricted to avoid abuse by unauthorized users.

To restrict access to the authorization server of a remote service, this specification introduces the additional *account_token* parameter to be used when calling the **oauth2/authorize** endpoint. This parameter contains a secure token designed to authenticate the authorization request based on an *Account ID* that SHALL be uniquely assigned by the signature application to the signing user or to the user's application account.

In case a RSSP wants to provide restricted access to its authorization server, it SHOULD register in advance the *Account ID* of the authorized users that need to have access to the **oauth2/authorize** endpoint.
The means and actions required to exchange and register an *Account ID* between users and the RSSP are out of the scope of this specification.

The *account_token* parameter is based on a JSON Web Token (JWT), defined as follows, according to the RFC 7519 [16]:

```
account_token = base64UrlEncode(<JWT_Header>) + "." +
                base64UrlEncode(<JWT_Payload>) + "." +
                base64UrlEncode(<JWT_Signature>)
```

JWT_Header

```
<JWT_Header> = {
   "typ": "JWT",
   "alg": "HS256"
}
```

JWT_Payload

```
<JWT_Payload> = {
   "sub": <Account_ID>,                  //Account ID
   "iat": <Unix_Epoch_Time>,             //Issued At Time
   "jti": <Token_Unique_Identifier>,     //JWT ID
   "iss": <Signature_Application_Name>,  //Issuer
   "azp": <OAuth2_client_id>             //Authorized presenter
}
```

JWT_Signature

```
<JWT_Signature> = HMACSHA256(
   base64UrlEncode(<JWT_Header>) + "." +
   base64UrlEncode(<JWT_Payload>),
   SHA256(<OAuth2_client_secret>)
)
```

Parameters

Parameter	Presence	Value	Description
typ	REQUIRED	*String* JWT	The Header Parameter used to indicate that this object is a JSON Web Token (JWT) according to RFC 7519 [16] Section 5.1.
alg	REQUIRED	*String* HS256	The Header Parameter used to indicate that the algorithm of the signature of the JWT is HMAC using SHA-256 according to RFC 7518 [15] Section 3.1.
sub	REQUIRED	*String*	The client-defined Account ID that allows the RSSP to identify the account or user initiating the authorization transaction.
iat	REQUIRED	*Number*	The Unix Epoch time when the *account_token* was issued. The value is used to determine the age of the JWT. The RSSP SHOULD define the lifetime of the JWT and SHALL accept or reject an *account_token* based on its own expiration policy.
jti	REQUIRED	*String*	A unique identifier for the JWT. This protects from replay attacks performed by reusing the same *account_token* .
iss	OPTIONAL	*String*	Contains the name of the issuer of the token (e.g. the commercial name of the signature application).
azp	REQUIRED	*String*	Contains the unique “client ID” previously assigned to the sgnature application by the remote service.

Implementation notes

- The RSSP SHALL securely share the OAuth 2.0 *client_id* and *client_secret* with the signature application as part of the OAuth 2.0 configuration (see OAuth 2.0 Authorization).
- The *JWT_signature* required to generate the *account_token* SHALL be calculated with the HMAC function, using as shared secret the SHA256 hash of the OAuth 2.0 *client_secret*.
- The signature application SHOULD register in advance with the RSSP the list of *Account ID* parameters associated with those users that are authorized to access a restricted authorization server.

Example

```
…?
account_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI3S1lCckpBLWtCOTF5T1Rld1JZRz
h5SGdzN3EtbzR1NiIsImlhdCI6MTUzNzAxMjgwMCwianRpIjoiYjgzZmY4OWEtZWQzZi00NjgxLTgyOGQtNzE2MGI
5MTNjYTcyIiwiaXNzIjoiQ1NDIFNpZ25hdHVyZSBBcHBsaWNhdGlvbiIsImF6cCI6ImE4NzliNDE5LThmZWQtNDcy
ZS05Yzk3LTJmODk3NTIxODU3ZSJ9.SEwD3KGDPFX-8IIJE7pC_RJ-0wdOVinEPTHmKKVQb6E&…
```

8.4.2 oauth2/authorize

Description

This is the OAuth 2.0 authorization endpoint. It SHALL process OAuth 2.0 authorization requests using the Authorization Code flow as described in Section 1.3.1 of RFC 6749 [11].

This endpoint can be used in two modes. The application either sends all authorization request parameters to this endpoint, which is the classical mode as defined in RFC 6749 [11]. Alternatively, the application can first push the authorization request payload to the authorization server via the "pushed authorization request endpoint" as defined in IETF Draft draft-ietf-oauth-par [28] and use the request URI produced as parameter to the authorization endpoints. This section describes the authorizaton request parameters for both modes. The pushed authorization endpoint and the use of the pushed authorization request mode is described in **oauth2/pushed_authorize**.

The authorization server MAY support scopes "service" or "credential" for service and credential authorization, respectively. The authorization server MAY also support use of "authorization_details" as defined in IETF Draft draft-ietf-oauth-rar [27] in conjunction with the authorization detail type "credential" for credential authorization. In case of "credential" authorization, a signing application SHALL either use the scope value "credential" (in conjunction with the details of the transaction in the URI parameters) or the authorization details type "credential" in a certain transaction (in which case the transaction details are contained in the authorization details object).

Note 14: Be aware that **oauth2/authorize** is designed as an unauthenticated endpoint. A provider offering this endpoint SHOULD protect the service from abuse and customer's risk. This is especially true when used for credential authorization. The authorization server MAY need to (re-)authenticate the user through the user agent before establishing a different, potentially cost-generating channel to the user (e.g. sending a push notification). A provider MAY apply practices like session cookies or HTML5 session storage in order to retain a good user experience, while addressing and mitigating related security issues. A provider MAY also implement individual access authorization mechanisms on the **oauth2/authorize** endpoint. The means for achieving this are beyond the scope of this specification.

Input

Note 15: Although RFC 3986 [3] doesn't define length limits on URIs, there are practical limits imposed by browsers and web servers. It is RECOMMENDED not to exceed an URI length of 2083 characters for maximum interoperability.

Input parameters defined in OAuth 2.0

Parameter	Presence	Value	Defined by	Description
response_type	REQUIRED	*String*	RFC 6749 [11]	see RFC 6749 [11], section 4.1.1. The value SHALL be "code".
client_id	REQUIRED	*String*	RFC 6749 [11]	see RFC 6749 [11], section 4.1.1.
redirect_uri	REQUIRED Conditional	*String*	RFC 6749 [11]	The URL where the user will be redirected after the authorization process has completed. The authorization is reqired to exactly match the parameter value with the pre-registered values. Only a valid URI pre-registered with the remote service SHALL be passed. If omitted, the remote service will use the default redirect URI pre-registered by the signature application.

Parameter	Presence	Value	Defined by	Description
scope	OPTIONAL	*String*	RFC 6749 [11]	The scope of the access request as described by Section 3.3 of RFC 6749 [11]. This specification defines the following scopes: • "service": it SHALL be used to obtain an authorization code suitable for service authorization. • "credential": it SHALL be used to obtain an authorization code suitable for credentials authorization. The scope of the request might be further detailed using request parameters as defined below. The parameter is OPTIONAL. If neither the "scope" nor the "authorization_details" parameter is provided, the authorization server SHALL use a default scope of "service".
authorization_details	OPTIONAL	*String*	IETF Draft-ietf-oauth-rar [27]	The details of the access request as described in IETF Draft-ietf-oauth-rar [27]. This specification defines the following authorization details type: • "credential": it SHALL be used to obtain an authorization code suitable for credentials authorization. The parameter is OPTIONAL. If this parameter is used, all values relevant for credential authorization SHALL be passed in this object. The scope "credential" as well as any request parameter relevant for credential authorization SHALL NOT be used in this case.
code_challenge	REQUIRED	*String*	RFC 7636 [25]	Cryptographic nonce binding the transaction to a certain user agent, used to detect code replay and CSRF attacks. See IETF RFC 7636 [25] and the IETF OAuth Security BCP [20], section 2.2, for details.
code_challenge_method	OPTIONAL	*String*	RFC 7636 [25]	Code verifier transformation method as defined in IETF RFC 7636 [25], defaults to `plain`. The recommended value is `S256`.
state	OPTIONAL	*String*	RFC 6749 [11]	see RFC 6749 [11], section 4.1.1.
request_uri	REQUIRED Conditional	*String*	IETF Draft-ietf-oauth-par [28]	URI pointing to a pushed authorization request previously uploaded by the client. This parameter SHALL only be used in conjunction with the `client_id`. All other parameters SHALL NOT be combined with this parameter.

Input parameters defined in this specification

This specification defines the following additional parameters:

Parameter	Presence	Value	Description
lang	OPTIONAL	*String*	Request a preferred language according to RFC 5646 [9]. If specified, the authorization server SHOULD render the authorization web page in this language, if supported. If omitted and an Accept-Language header is passed, the authorization server SHOULD render the authorization web page in the language declared by the header value, if supported. The authorization server SHALL render the web page in its own preferred language otherwise.
credentialID	REQUIRED Conditional	*String*	The identifier associated to the credential to authorize. It SHALL be used only if the scope of the OAuth 2.0 authorization request is "credential". Be aware that this parameter value may contain characters that are reserved, unsafe or forbidden in URLs and therefore SHALL be url-encoded by the signature application.

Parameter	Presence	Value	Description
signatureQualifier	REQUIRED Conditional	*String*	This parameter contains the symbolic identifier determining the kind of signature to be created as defined in signatures/signDoc. It SHALL be used only if the scope of the OAuth 2.0 authorization request is "credential" and if there is no parameter "credentialID" present.
numSignatures	REQUIRED Conditional	*Number*	The number of signatures to authorize. Multi-signature transactions can be obtained by using a combination of array of hash values and by calling multiple times the **signatures/signHash** method, as defined in signatures/signHash. It SHALL be used only if the scope of the OAuth 2.0 authorization request is "credential".
hashes	REQUIRED Conditional	*String*	One or more base64url-encoded hash values to be signed. It allows the server to bind the access token to the hash, thus preventing an authorization to be used to sign a different content. It SHALL be used only if the scope of the OAuth 2.0 authorization request is "credential". It SHALL be used if the *SCAL* parameter returned by **credentials/info** method, as defined in credentials/info, for the current *credentialID* is "2", otherwise it is OPTIONAL. Multiple hash values can be passed as comma separated values, e.g. `oauth2/authorize?hash=dnN3ZX.. .ZmRm,ZjIxM3… Z2Zk,…` The order of multiple values does not have to match the order of hashes passed to **signatures/signHash** method, as defined in signatures/signHash.
hashAlgorithmOID	REQUIRED Conditional	*String*	String containing the OID of the hash algorithm used to generate the hashes.
description	OPTIONAL	*String*	A free form description of the authorization transaction in the *lang* language. The maximum size of the string is 500 characters. It can be useful to provide some hints about the occurring transaction.
account_token	OPTIONAL	*String*	An account_token as defined in Restricted access to authorization servers. It MAY be required by a RSSP if their authorization server has a restricted access. The value is a JSON Web Token (JWT) according to RFC 7519 [16].
clientData	OPTIONAL	*String*	Arbitrary data from the signature application. It can be used to handle a transaction identifier or other application-spe cific data that may be useful for debugging purposes. WARNING: this parameter MAY expose sensitive data to the remote service. Therefore it SHOULD be used carefully.

Authorization details type "credential"

The authorization details type `credential` allows applications to pass the details of a certain credential authorization in a single JSON object. It consists of the following field:

Field	Presence	Value	Description
type	REQUIRED	*String*	authorization details type identifier. It must be set to `credential`.
credentialID	REQUIRED Conditional	*String*	see definition above (Input parameters).
signatureQualifier	REQUIRED Conditional	*String*	see definition above (Input parameters).
documentDigests	REQUIRED	*JSON array*	An array composed of entries for every document to be signed. This applies for both cases, where are document is signed or a digest is signed. Every entry is composed of the following elements: • "hash": REQUIRED Conditional String containing the actual Base64-encoded octet-representation of the hash of the document. • "label": String containing a human-readable description of the respective document. The AS will use the label element in the user consent to designate the document.
hashAlgorithmOID	REQUIRED	*String*	String containing the OID of the hash algorithm used to generate the hashes listed in documentDigests.

Field	Presence	Value	Description
locations	OPTIONAL	*JSON array*	Element as defined in IETF Draft-ietf-oauth-rar [27] designating the locations of the API the access token issued in a certain OAuth transaction shall be used. Might be used by deployments to identify the RSSP.

If the credential authorization values are provided via this authorization details, then they SHALL NOT be provided within the other request parameters. The authorization details SHOULD be used, since it allows a more detailes information on the documents to be signed.

Output

After a successful authorization, the authorization server SHALL redirect the user-agent by sending the HTTP/1.1 302 Found response with a Location header containing the URI specified by the *redirect_uri* parameter and adding the following values as query component using the "application/x-www-form-urlencoded" format.

Attribute	Presence	Value	Description
code	REQUIRED	*String*	The authorization code generated by the authorization server. It SHALL be bound to the client identifier and the redirection URI. It SHALL expire shortly after it is issued to mitigate the risk of leaks. The signature application cannot use the value more than once.
state	REQUIRED Conditional	*String*	Contains the arbitrary data from the signature application that was specified in the state attribute of the input request. It SHALL be returned when specified in the request.
error	REQUIRED Conditional	*String* invalid_request | access_denied | unsupported_response_type | invalid_scope | server_error | temporarily_unavailable	A single error code string from the following list: • "invalid_request": it SHALL be used if the request is missing a required parameter. • "access_denied": it SHALL be used if the server denied the request. • "unsupported_response_ty pe": it SHALL be used if the server does not support the required response type. • "invalid_scope": it SHALL be used if the requested scope is invalid, unknown, or malformed. • "server_error": it SHALL be used if the server encountered an unexpected condition that prevented it from fulfilling the request. • "temporarily_unavailable" : it SHALL be used if the server is currently unable to handle the request due to temporary overload or maintenance . It SHALL be returned only in case of an error.
error_description	OPTIONAL	*String*	Human-readable text providing additional error information. It MAY be returned only in case of an error.
error_uri	OPTIONAL	*String*	A URI identifying a human-readable web page with information about the error. It MAY be returned only in case of an error.

Sample Request (Service authorization)

```
GET https://www.domain.org/oauth2/authorize?
  response_type=code&
  client_id=<OAuth2_client_id>&
  redirect_uri=<OAuth2_redirect_uri>&
  scope=service&
  code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256&
```

```
  lang=en-US&
  state=12345678
```

Sample Response (Service authorization)

```
HTTP/1.1 302 Found
Location: <OAuth2_redirect_uri>?
  code=FhkXf9P269L8g&
  state=12345678
```

Sample Request (Credential authorization)

```
GET https://www.domain.org/oauth2/authorize?
  response_type=code&
  client_id=<OAuth2_client_id>&
  redirect_uri=<OAuth2_redirect_uri>&
  scope=credential&
  code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256&
  credentialID=GX0112348&
  numSignatures=1&
  hashes=MTIzNDU2Nzg5MHF3ZXJ0enVpb3Bhc2RmZ2hqa2zDtnl4&
  hashAlgorithmOID=2.16.840.1.101.3.4.2.1&state=12345678
```

Sample Response (Credential authorization)

```
HTTP/1.1 302 Found
Location: <OAuth2_redirect_uri>?code=HS9naJKWwp901hBcK348IUHiuH8374&
  state=12345678
```

Sample Request (Credential authorization with signature qualifier)

```
GET https://www.domain.org/oauth2/authorize?
  response_type=code&
  client_id=<OAuth2_client_id>&
  redirect_uri=<OAuth2_redirect_uri>&
  scope=credential&
  code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256&
  signatureQualifier=eu_eidas_qes&
  numSignatures=1&
  hashes=MTIzNDU2Nzg5MHF3ZXJ0enVpb3Bhc2RmZ2hqa2zDtnl4&
  hashAlgorithmOID=2.16.840.1.101.3.4.2.1&state=12345678
```

Sample Response (Credential authorization with signature qualifier)

```
HTTP/1.1 302 Found
Location: <OAuth2_redirect_uri>?code=HS9naJKWwp901hBcK348IUHiuH8374&
  state=12345678
```

Sample Request (Credential authorization with signature qualifier via authorization_details)

```
GET https://www.domain.org/oauth2/authorize?
  response_type=code&
  client_id=<OAuth2_client_id>&
  redirect_uri=<OAuth2_redirect_uri>&
  code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256&
  &state=12345678
```

```
&authorization_details=%5B%7B%22type%22:%22credential%22,%22signatureQualifier%22:%22eu_e
idas_qes%22,%22documentDigests%22:%5B%7B%22hash%22:%22sTOgwOm+474gFj0q0x1iSNspKqbcse4Ieiq
lDg/HWuI=%22,%22label%22:%22Example%20Contract%22%7D,%7B%22hash%22:%22HZQzZmMAIWekfGH0/ZK
W1nsdt0xg3H6bZYztgsMTLw0=%22,%22label%22:%22Example%20Terms%20of%20Service%22%7D%5D,%22ha
shAlgorithmOID%22:%222.16.840.1.101.3.4.2.1%22%7D%5D
```

Decoded `authorization_details` parameter

```
[
   {
      "type":"credential",
      "signatureQualifier":"eu_eidas_qes",
      "documentDigests":[
         {
            "hash":"sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
            "label":"Example Contract"
         },
         {
            "hash":"HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
            "label":"Example Terms of Service"
         }
      ],
      "hashAlgorithmOID":"2.16.840.1.101.3.4.2.1"
   }
]
```

Sample Response (Credential authorization with signature qualifier)

```
HTTP/1.1 302 Found
Location: <OAuth2_redirect_uri>?code=HS9naJKWwp901hBcK348IUHiuH8374&
  state=12345678
```

Error Response

```
HTTP/1.1 302 Found
Location: <OAuth2_redirect_uri>?error=invalid_request&
  error_description=Invalid%20Authorization%20Code&state=12345678
```

8.4.3 oauth2/pushed_authorize

This is the OAuth 2.0 pushed authorization endpoint as defined in IETF Draft draft-ietf-oauth-par [28]. It allows clients to push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and provides them with a request URI that is used as reference to the data in a subsequent call to the authorization endpoint (**oauth2/authorize**).

This mechanisms protects the contents of the authorization request from modification and eavesdropping, allows for practically arbitrary request sizes, and enables the authorization server to authenticate the signing application in advance of the authorization process.

The application sends the parameters as defined in **oauth2/authorize** (except the `request_uri` parameter) to the pushed authorization endpoint using a HTTP POST request. The application is required to authenticate towards the authorization server using the mechanism used in the context of token requests (see **oauth2/token**). The authorization server will respond with a request URI that the application sends to the authorization endpoint along with its `client_id` instead of the authorization parameters.

Sample Pushed Authorization Request (Service authorization)

```
POST oauth2/pushed_authorize HTTP/1.1
Host: www.domain.org
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

response_type=code&
client_id=<OAuth2_client_id>&
redirect_uri=<OAuth2_redirect_uri>&
scope=service&
code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
code_challenge_method=S256&
lang=en-US&
state=12345678
```

Sample Pushed Authorization Request (Credential authorization with authorization details)

```
POST oauth2/pushed_authorize HTTP/1.1
Host: www.domain.org
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

response_type=code&
client_id=<OAuth2_client_id>&
redirect_uri=<OAuth2_redirect_uri>&
code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
code_challenge_method=S256&
&state=12345678
&authorization_details=%5B%7B%22type%22:%22credential%22,%22signatureQualifier%22:%22eu_e
idas_qes%22,%22documentDigests%22:%5B%7B%22hash%22:%22sTOgwOm+474gFj0q0x1iSNspKqbcse4Ieiq
lDg/HWuI=%22,%22label%22:%22Example%20Contract%22%7D,%7B%22hash%22:%22HZQzZmMAIWekfGH0/ZK
W1nsdt0xg3H6bZYztgsMTLw0=%22,%22label%22:%22Example%20Terms%20of%20Service%22%7D%5D,%22ha
shAlgorithmOID%22:%222.16.840.1.101.3.4.2.1%22%7D%5D
```

Sample Pushed Authorization Response (Service authorization)

```
HTTP/1.1 201 Created
Cache-Control: no-cache, no-store
Content-Type: application/json

{
   "request_uri": "urn:example:bwc4JK-ESC0w8acc191e-Y1LTC2",
   "expires_in": 90
}
```

Sample authorization Request (with request_uri)

```
GET /authorize?client_id=<OAuth2_client_id>
       &request_uri=urn%3Aexample%3Abwc4JK-ESC0w8acc191e-Y1LTC2 HTTP/1.1
     Host: as.example.com
```

8.4.4 oauth2/token

Description

This is the OAuth token endpoint. It is used to obtain an OAuth 2.0 bearer access token from the authorization server by passing either the client credentials pre-assigned by the authorization server to the signature application, or the authorization code or refresh token returned by the

authorization server after a successful user authentication, along with the client ID and client secret in possession of the signature application. This method SHALL be used only in case of an Authorization Code flow as described in Section 1.3.1 of RFC 6749 [11], in case of Client Credential flow as described in Section 1.3.4 of RFC 6749 [11] or in case of Refresh Token flow as described in Section 1.5 of RFC 6749 [11]. Notice that the Client Credential flow and Refresh Token flow can be used only for service authorization.

For confidential clients, implementations MAY utilize any of the client authentication methods defined in the IANA "OAuth Token Endpoint Authentication Methods" registry established by IETF RFC 7591 [24].

This is a non-exhaustive list of options:

- Passing a pre-issued client secret as a parameter in the request body as described in Section 2.3.1 of RFC 6749 [11].
- Applying a pre-issued client secret within the HTTP Basic authentication scheme as described in Section 2.3.1 of RFC 6749 [11].
- Passing a client assertion as defined in section 4.2 of RFC 7521 [14].
- Using TLS Client authentication as defined in RFC 8705.

Note 16: **oauth2/token** does not specify a regular CSC API method, but rather the URI of the OAuth 2.0 Token endpoint. Depending on the discovery method, this URL is either determined by adding **oauth2/token** to the authorization server's base URI or from the authorization server's configuration.

Input

In order to maintain full compatibility with the OAuth 2.0 standard, the following parameters SHALL be passed in the HTTP request entity-body using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8.

Note 17: The list of parameters is split between standard parameters that are defined by the OAuth 2.0 framework (see RFC 6749 [11] and RFC 7521 [14]) and parameters that are defined in this specification. These parameters SHALL be combined in a single query string.

Input parameters defined in OAuth 2.0

Parameter	Presence	Value	Description
grant_type	REQUIRED	*String* authorization_code | client_credentials | refresh_token	The grant type, which depends on the type of OAuth 2.0 flow: • "authorization_code": SHALL be used in case of Authorization Code Grant. • "client_credentials": SHALL be used in case of Client Credentials Grant. • "refresh_token": SHALL be used in case of Refresh Token flow.
code	REQUIRED Conditional	*String*	The authorization code returned by the authorization server. It SHALL be bound to the client identifier and the redirection URI. This SHALL be used only when *grant_type* is "authorization_code".

Parameter	Presence	Value	Description
refresh_token	REQUIRED Conditional	*String*	The long-lived refresh token returned from the previous session. This SHALL be used only when the scope of the OAuth 2.0 authorization request is "service" and *grant_type* is "refresh_token" to reauthenticate the user according to the method described in Section 1.5 of RFC 6749 [11].
client_id	REQUIRED	*String*	The *client_id* as defined in the Input parameter table in oauth2/authorize.
client_secret	REQUIRED Conditional	*String*	This is the "client secret" previously assigned to the signature application by the remote service. It SHALL be passed if the client is setup to authenticate with a client secret and does not use an authorization header. Note: According to RFC 6749 [11] section 2.3.1., including the client credentials in the request-body is NOT RECOMMENDED and SHOULD be limited to clients unable to directly utilize the HTTP Basic authentication scheme.
client_assertion	REQUIRED Conditional	*String*	The assertion being used to authenticate the client. Specific serialization of the assertion is defined by profile documents. It SHALL be passed if the client is setup for authentication with client assertions.
client_assertion_type	REQUIRED Conditional	*String*	The format of the assertion as defined by the authorization server. The value will be an absolute URI. It SHALL be passed if a client assertion is used.
redirect_uri	REQUIRED Conditional	*String*	The URL where the user was redirected after the authorization process completed. It is used to validate that it matches the original value previously passed to the authorization server. This SHALL be used only if the *redirect_uri* parameter was included in the authorization request, and their values SHALL be identical.
authorization_details	REQUIRED Conditional	*String*	MUST be present if the `authorization_details` parameter was used in the authorization request. It contains the authorization details as approved during the authorization process. In case a signature qualifier was used in the request and resolved for a credential ID in the course of the authorization process, this object will contain the credential ID.

Input parameters defined in this specification

Parameter	Presence	Value	Description
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.

Output

This method returns the following values using the "application/json" format:

Output parameters defined in OAuth 2.0

Attribute	Presence	Value	Description
access_token	REQUIRED	*String*	The short-lived access token to be used depending on the *scope* of the OAuth 2.0 authorization request. This access token as the value of the "Authorization: Bearer" in the HTTP header of the subsequent API requests within the same session. A signing application MAY also pass an access tokens with scope "credential" as the value of the SAD parameter when invoking the **signatures/signHash** or **signatures/signDoc** methods, as defined in signatures/signHash.

Attribute	Presence	Value	Description
refresh_token	OPTIONAL	*String*	The long-lived refresh token used to re-authenticate the user on the subsequent session based on the method described in Section 1.5 of RFC 6749 [11]. The presence of this parameter is controlled by the user and is allowed only when the scope of the OAuth 2.0 authorization request is "service". In case *grant_type* is "refresh_token" the authorization server MAY issue a new refresh token, in which case the client SHALL discard the old refresh token and replace it with the new refresh token.
token_type	REQUIRED	*String*	Access token type as defined in RFC 6749 [11]. Default is "Bearer", other token types are defined in the "OAuth Access Token Types" established by RFC 6749 [11].
expires_in	OPTIONAL	*Number*	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 sec. (1 hour).

Note 18: The lifetime of the refresh token is determined by the RSSP.

Output parameters defined in this specification

Attribute	Presence	Value	Description
credentialID	OPTIONAL	*String*	The identifier associated to the credential authorized in the corresponding authorization request. This response parameter MAY be present in case the scope `credential` is used in the authorization request along with the parameter "signatureQualifier" and the authorization server determined a credentialID in the authorization process to be used in subsequent signature operations.

Error Case	Status Code	Error	Error Description
Missing "client_id" parameter	400 (bad request)	invalid_request	Missing parameter client_id
Missing "grant_type" parameter	400 (bad request)	invalid_request	Missing parameter grant_type
Invalid parameter "grant_type"	400 (bad request)	invalid_request	Invalid parameter grant_type
Missing "code" parameter	400 (bad request)	invalid_request	Missing parameter code
Missing "refresh_token" parameter	400 (bad request)	invalid_request	Missing parameter refresh_token
Invalid "client_id" parameter	400 (bad request)	invalid_request	Invalid parameter client_id
Invalid "code" parameter	400 (bad request)	invalid_grant	Invalid parameter code
The "redirect_uri" parameter does not match the redirection URI in the authorization request	400 (bad request)	invalid_grant	redirect_uri parameter does not match redirect_uri parameter of authorization request
Invalid "refresh_token" parameter	400 (bad request)	invalid_grant	Invalid parameter refresh_token
Refresh token expired	400 (bad request)	invalid_grant	Refresh token expired
Authorization code invalid or expired	400 (bad request)	invalid_grant	Authorization code is invalid or expired
Missing "client_secret" parameter and no authorization header provided	400 (bad request) | 401 (unauthorized)	invalid_request	Client authorization required

Error Case	Status Code	Error	Error Description
Invalid “client_secret” parameter	400 (bad request)	invalid_request	Invalid parameter client_secret

Sample Request (Authorization code flow)

```
POST oauth2/token HTTP/1.1
Host: www.domain.org
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=FhkXf9P269L8g&
client_id=<OAuth2_client_id>&
client_secret=<OAuth2_client_secret>&
redirect_uri=<OAuth2_redirect_uri>
```

cURL example

```
curl -i -X POST
     -H "Content-Type: application/x-www-form-urlencoded"
     -d 'grant_type=authorization_code&
         code=FhkXf9P269L8g&
         client_id=<OAuth2_client_id>&
         client_secret=<OAuth2_client_secret>&
         redirect_uri=<OAuth2_redirect_uri>'
    https://www.domain.org/oauth2/token
```

Sample Response (for service scope)

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
   "access_token": "4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA",
   "refresh_token": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
   "token_type": "Bearer",
   "expires_in": 3600
}
```

Sample Response (for credential scope)

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
   "access_token":
   "3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5H3XlFQZ3ndFhkXf9P2",
   "token_type": "Bearer",
   "expires_in": 300
}
```

Sample Response (for credential scope with signature qualifier and AS selected credential)

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
   "access_token":
   "3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5H3XlFQZ3ndFhkXf9P2",
   "token_type": "Bearer",
   "expires_in": 300,
```

```
   "credentialID": "GX0112348"
}
```

Sample Response (for credential authorization details with signature qualifier and AS selected credential)

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
   "access_token":"3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5H3XlFQZ3ndFhkXf9P2",
   "token_type":"Bearer",
   "expires_in":300,
   "authorization_details":[
      {
         "type":"credential",
         "credentialID":"GX0112348",
         "documentDigests":[
            {
               "hash":"sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
               "label":"Example Contract"
            },
            {
               "hash":"HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
               "label":"Example Terms of Service"
            }
         ],
         "hashAlgorithmOID":"2.16.840.1.101.3.4.2.1"
      }
   ]
}
```

Sample Request (Refresh token flow)

```
POST oauth2/token HTTP/1.1
Host: www.domain.org
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&
refreshToken=_TiHRG-bA+H3XlFQZ3ndFhkXf9P24%2FCKN69L8gdSYp5_pw&
client_id=<OAuth2_client_id>&
client_secret=<OAuth2_client_secret>&
redirect_uri=<OAuth2_redirect_uri>
```

cURL example

```
curl -i -X POST
     -H "Content-Type: application/x-www-form-urlencoded"
     -d 'grant_type=refresh_token&
         refreshToken=_TiHRG-bA+H3XlFQZ3ndFhkXf9P24%2FCKN69L8gdSYp5_pw&
         client_id=<OAuth2_client_id>&
         client_secret=<OAuth2_client_secret>&
         redirect_uri=<OAuth2_redirect_uri>'
     https://www.domain.org/oauth2/token
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
   "access_token": "K7x-0Lj7Wwdt4pwH3XlFQZ3ndFhkXf9P2_TiHRQaxZ9kJ0",
   "token_type": "Bearer",
```

```
   "expires_in": 3600
}
```

8.4.5 oauth2/revoke

Description

Revoke an access token or refresh token that was obtained from the authorization server, as described in RFC 7009 [13]. This method may be used to enforce the security of the remote service. When the signature application needs to terminate a session, it is RECOMMENDED to invoke this method to prevent further access by reusing the token.
This method allows the signature application to invalidate its tokens according to the following approach:

- If the token passed to the request is a *refresh_token*, then the authorization server SHALL invalidate the refresh token and it SHOULD also invalidate all access tokens based on the same authorization grant.
- If the token passed to the request is an *access_token*, then the authorization server SHALL invalidate the access token and it SHALL NOT revoke any existing refresh token based on the same authorization grant.

The invalidation of the token takes place immediately, and the token cannot be used again after its revocation. As a token issued in the process of credential authorization is automatically invalidated as soon as its usage limit is reached, a client does not have to revoke the corresponding token after use. However, a provider SHOULD support the revocation of such a token before reaching the usage limit.

A confidential client SHALL authenticate with the authorization server using its client authentication method.

Note 19: **oauth2/revoke** does not specify a regular CSC API method, but rather the URI of the OAuth 2.0 Token endpoint. Depending on the discovery method, this URL is either determined by adding **oauth/revoke** to the authorization server's base URI or from the authorization server's configuration.

8.4.5.1 Input

In order to maintain full compatibility with the OAuth 2.0 standard, the following parameters SHALL be passed in the HTTP request entity-body with the authorization endpoint URI using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8.

Note 20: The list of parameters is split between standard parameters that are defined by the OAuth 2.0 framework (see RFC 6749 [11] and RFC 7521 [14]) and parameters that are defined in this specification. These parameters SHALL be combined in a single query string.

Input parameters defined in OAuth 2.0

Parameter	Presence	Value	Description
token	REQUIRED	*String*	The token that the signature application wants to get revoked.

Parameter	Presence	Value	Description
token_type_hint	OPTIONAL	*String* access_token | refresh_token	Specifies an optional hint about the type of the token submitted for revocation. If the parameter is omitted, the authorization server SHOULD try to identify the token across all the available tokens.
client_id	REQUIRED Conditional	*String*	The *client_id* as defined in the Input parameter table in oauth2/authorize. It SHALL be passed if no authorization header is used.
client_secret	REQUIRED Conditional	*String*	The *client_secret* as defined in the Input parameter table in oauth2/token.
client_assertion	REQUIRED Conditional	*String*	The *client_assertion* as defined in the Input parameter table in oauth2/token.
client_assertion_type	REQUIRED Conditional	*String*	The *client_assertion_type* as defined in the Input parameter table in oauth2/token.

Input parameters defined in this specification

Parameter	Presence	Value	Description
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.

Output

This method has no output values and the response returns “No Content” status.

Error Case	Status Code	Error	Error Description
Missing “token” parameter	400 (bad request)	invalid_request	Missing parameter token
“token_hint” parameter present, not equal to “access_token” nor “refresh_token”	400 (bad request)	invalid_request	Invalid parameter token_type_hint
Invalid access_token or refresh_token	400 (bad request)	invalid_request	Invalid string parameter token
Unsupported token type	400 (bad request)	unsupported_token_type	The authorization server does not support the revocation of the presented token type. That is, the client tried to revoke an access token on a server not supporting this feature.
Missing “client_id” parameter and no authorization header provided	400 (bad request) | 401 (unauthorized)	invalid_request	Missing parameter client_id
Invalid “client_id” parameter	400 (bad request)	invalid_request	Invalid parameter client_id
Missing “client_secret” parameter and no authorization header provided	400 (bad request) | 401 (unauthorized)	invalid_request	Client authorization required
Invalid “client_secret” parameter	400 (bad request)	invalid_request	Invalid parameter client_secret
Invalid Authorization header	401 (unauthorized)	invalid_client	Invalid authorization header

Sample Request